Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Anders Rantala Hunderi

Supporting and Improving the Extensibility of the "Odin" system

Master's thesis in Computer Science
Supervisor: Hallvard Trætteberg
January 2022

NTNU
Norwegian University of
Science and Technology

Anders Rantala Hunderi

Supporting and Improving the Extensibility of the “Odin” system

Master’s thesis in Computer Science
Supervisor: Hallvard Trætteberg
January 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Sammendrag

Som en del av en masteroppgave i 2021 ble “Odin” systemet utviklet for å hjelpe fagstab ved NTNU administrere og synkronisere student-grupper på tvers av den digitale læringsplattformen Blackboard og kildekodehåndterings-tjenesten Gitlab, samt gi innsikt i studentenes arbeidsmetodikk i form av et metrikk-dashbord.

Dashbordet som har blitt utviklet eksisterer i dag mer som et proof-of-concept, ettersom det fortsatt er uklart hvilke datasett det er viktig å analysere. Dette betyr at fremtidig utvikling av systemet vil kreve en dels eksperimentering av hvilke metrikker som vises – ved å modifisere eksisterende elementer, eller utvikle nye. For å forsikre at slikt implementasjonsarbeid kan gjennomføres på en effektiv og organisert måte, analyserer denne studien “utvidbarheten” (extensibility) av dagens system og implementerer endringer i arkitekturen for å forbedre utvidbarheten som nødvendig. Ettersom uttrykket “utvidbarhet” per nå ikke har en standardisert definisjon ble et kortere litteraturstudium gjennomført, som samlet inn definisjoner og uttrykk som kan brukes for å beskrive et systems utvidbarhet.

Etter å ha sett en mangel på veldefinert utvidbarhet i det eksisterende systemet, ble et internt plugin-system implementert. Dette systemet eksponerer deler av den interne metrikklogikken og tillater en å legge til nye data og visualiseringer i systemet, på en måte som fullt skiller disse tilleggene fra den eksisterende kildekoden. I tillegg tillater plugin-systemet (delvis) at plugin-artefakter blir bundet senere i systemets livssyklus, hvilket reduserer driftsstans når ny logikk legges til. De metrikkene som nå vises i Odin-systemet ble trukket ut, og omskrevet til selvstendige plugins. Dette gir både et eksempel på hvordan plugins kan utvikles for det nye systemet, men viser også at systemet tillater utvikling av metrikk-plugins som er minst like kraftige som de eksisterende metrikkene.

Acknowledgements

This thesis marks the end of my now six years of studies at NTNU. First, I would like to thank my supervisor Hallvard Trætteberg for dedicating his time and knowledge to guide me throughout this project.

Secondly, I would like to thank the Master's students Petter Grø Rein and Tore Stensaker Tefre, who created the "Odin" web-application which this thesis builds upon. I would especially extend my thanks to Rein for being available and willing to answer any questions I had about the Odin system during my project.

Lastly, I would like to thank my friends and family for their support and interest in the project, and for helping me keep my sanity during an arduous period of pandemic waves, home-offices, and technical difficulties. I would especially like to thank my friend, Scott Mitchell, for his invaluable feedback and motivational support during this project.

Abstract

As part of a Master's Thesis from 2021, the "Odin" system was created to help NTNU course staff manage and synchronize student groups across the Blackboard Learning Management System (LMS) and Gitlab Source Code Management (SCM) Service, and provide insight into the students' coding process by means of a metrics dashboard.

Today, the metric dashboard more-so exists as a proof-of-concept, since there is still uncertainty as to which data is important to analyze. Future work on this system will therefore require experimentation with which metrics are shown in the dashboard – modifying existing ones, or developing new ones. To ensure that these implementations can be done in an effective and organized manner, this study analyzes the "extensibility" of the current system and implements architectural changes to improve the extensibility as needed. As the term "extensibility" is currently not a standardized expression, a shorter literary review was also done in order to gather definitions and key phrases one can use to describe a system's extensibility.

Seeing a lack of well-defined extensibility in the current system, a plugin sub-system was implemented. This system exposes the internal metric logic to allow new data and visualizations to be added, and allows new metric capabilities to be added into the system in a way that fully encapsulates these additions from the existing core. Furthermore, the plugin system (partially) allows plugins to be bound later in the system's life-cycle, lessening downtime whenever new logic is added. The current metrics shown in the dashboard were extracted into plugins, serving as both examples of how plugins can be made, but also showcasing that the system allows the development of metric plugins at least as complex as the existing metrics.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API Application Programming Interface

CSS Cascading Style Sheets

DOM Document Object Model

HTML Hyper-Text Markup Language

IDE Integrated Development Environment

JSON JavaScript Object Notation

JSX JavaScript Syntax Extension

LMS Learning Management System

MVP Minimum Value Product

NTNU Norwegian University of Science and Technology

PAT Personal Access Token

RCP Rich Client Platform

SCM Source Code Management

UI User Interface

URL Unified Resource Locator

VCS Version Control System

XUL XML User Interface Language

XML Extensible Markup Language

1 | Introduction

1.1 Context

It should come as no surprise that developing software is a major aspect of the software engineering courses students partake in at the Norwegian University of Science and Technology (NTNU). As with real life software development, group based coding assignments are a key factor in the teaching methodology. One crucial activity for the academic staff is to manage these groups, and ensure they are working as intended. This can become cumbersome in a course where several hundred students are admitted yearly, so some systems to ease this activity is needed. Currently, NTNU's staff and students use the Blackboard Learn Learning Management System (LMS) to exchange information, manage students and groups, and handle assignment deliveries.

When managing the development of a software product, the issue of tracking and merging changes is usually handled with the help of some form of a Version Control System (VCS). Of these systems, Git is arguably one of the most widely used (RhodeCode 2016). The use of the Git VCS is facilitated in coding assignments at NTNU by allowing – and often requiring – projects to be developed and delivered as a Git repository. As hosting and managing Git repositories is outside the scope of the Blackboard LMS, NTNU has created their own instance of the Gitlab Source Code Management (SCM) service, integrating the user authentication with the same user database as many other institute systems use.

Unfortunately these systems do not communicate with each-other, meaning there is no direct link between data defined on Blackboard and data defined on Gitlab. Because of this, course staff must exert extra effort when performing tasks that requires data from both services. For instance, this becomes a prevalent issue during group assignments, where groups must be managed and kept synchronized across the two systems.

In addition to group administration, an important part of course staff's work in these subjects is following up on the groups' development process throughout the course, often to make sure the group is working well as a team. With the separated systems used today, this process requires extra effort as course staff must, step-by-step, go through student repositories, searching

through files, commits[1], issue listings, and so on – manually analyzing this data to get a sense of the group's performance.

In response to this issue, a study was performed in the previous year (spring 2021)– as part of a Master's Thesis – which aimed to create a software solution able to merge these systems in a more seamless manner (Rein & Tefre 2021). The end result of this thesis was the development of a Minimum Value Product (MVP) realized in the form of the "Odin" web application, which collects and manages data from both systems. This semester (autumn 2021) the management system will be put through a trial run where it will be used to manage coding assignments for students in the IT1901 course.

1.2 Motivation

As the aforementioned system is an MVP, only the most important functionality has been prioritized and developed. Since the functional requirements are likely to evolve as time passes, it is important for the course staff – the main stakeholders – that the system is in a state where extending the functionality can be done in an efficient manner. One way to realize such a requirement is to ensure that the systems architecture – or specifically the set of software elements and relations that comprise the system (Bass et al. 2012) – are designed in such a way that it supports this. More precisely, we want an architecture that satisfies the *extensibility* quality goal.

As discussed later on in Chapter 3, the original project did not directly state extensibility as a quality goal. The architecture of a system is formed by it's overarching business and quality goals (Bass et al. 2012), and one can therefore argue that if the preliminary design discussions and decisions did not focus on a given quality, the resulting design is unlikely to support the quality in a valuable manner. Any indirect support would be the result of either the architect's own preferences or an overlap from other, similar qualities.

To ensure that the system is as extensible as desired, this thesis will aim to analyse the extensibility of current architecture, and implement architectural changes required to make the system support the desired degree of extensibility.

[1] A "commit" is Git terminology for "an identifiable set of changes to one or more project files"

1.3 Research Questions

This thesis will attempt to answer the following research questions:

- **Research Question 1 (RQ1):**
  What characterizes “Extensible" software? What makes some software be regarded as more extensible than others?
- **Research Question 2 (RQ2):**
  To what extent does the current system support the extensibility quality goal?
- **Research Question 3 (RQ3):**
  How can the current system’s architecture be modified to better support extensibility?

1.4 Report Outline

Chapter 1 – Introduction – presents the context and motivation for this thesis, and presents the research questions that drives the study.

Chapter 2 – Theory – gives a shorter presentation of academic literature surrounding the extensibility quality attribute, and presents a definition of the term itself, along with key terms that will be used in later discussions. Additionally, this chapter presents three examples of extensible architectures seen in popular extensible software.

Chapter 3 – Supporting Extensibility in “Odin” – analyses the architecture of the existing Odin system, discusses how well the current system supports extensibility, and proposes a design of a plugin system that aims to make the system more extensible.

Chapter 4 – Plugin System Implementation – showcases the implementation work done to realize the design proposed in Chapter 3.

Chapter 5 – Results and Discussion – puts forth answers to the thesis’ research questions, and discusses the validity of these results.

Chapter 6 – Conclusion and Further Work concludes the thesis: summarizing the findings and proposing possible directions for future work and research.

2 | Theory

In order to evaluate the extensibility of the current system, and reason about architectural design decisions whilst improving the system, it will be necessary to begin mapping some definitions and phrases in regards to extensibility. This chapter will put forth a general definition and characteristics of extensibility and the different ways the quality can be supported by a software's architecture. It will also discuss some common methods and patterns we can see utilized in existing software and frameworks that are generally regarded as "extensible".

2.1 Defining Extensibility

As a starting point to find a definition, the ISO/IEC 25010 standard was looked at (ISO 2011). This standard defines eight high-level quality attributes, which is further separated into a large set of sub-groups. This standardized set is shown in 2.1. As one can see here, the current standard for quality attributes does *not* provide a definition of the term "extensibility", meaning it will be necessary to look for a definition elsewhere.

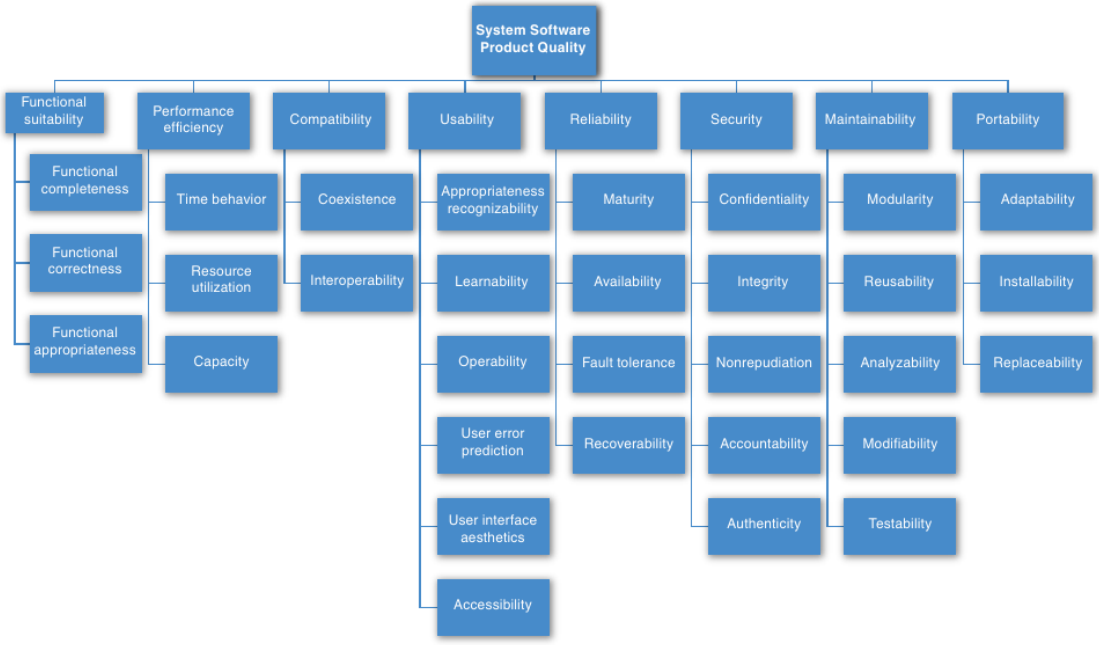


Figure 2.1: Quality attributes defined in ISO 25010 (Figure from Bass et al. (2012)).

In a doctoral thesis about component-based software, Zenger (2004) states that software is extensible if it can be adapted to unanticipated changes in it's specification. Designing for extensibility is a means to reduce the cost of introducing new – or similar – functionality into a system.

In Hillard (2020), we also see extensibility discussed, and here Hillard claims that software is *ideally* extensible if new behaviour can be added without any changes to existing code, and without affecting existing functionality. Hillard further notes that since real systems are rarely ideal, you will still need to change existing code on occasion while extending the software. Therefore, supporting extensibility also means you must support some degree of flexibility, to allow such needed changes to be made. This signifies that there is some general relation between flexibility and extensibility.

In the 4th European Conference on Software Architecture, Bode & Riebisch (2010) discusses supporting the ongoing evolution of a system, an puts forth how to measure and support the "evolvability" quality goal. Here, evolvability is defined as follows:

> *Evolvability is the ability of a software system throughout it's life-cycle to accommodate to changes and enhancements in requirements and techonologies, that influence the system's architectural structure, with the least possible cost while maintaining the architectural integrity*

In this definition, Bode & Riebisch mentions *changes in requirements*, matching Zenger's definition about changes in specification, but this quality goal is still very broad, and is more considered with systems of very long lifetimes.

Bode & Riebisch goes on to define evolvability as a super-set of several more precises qualities that affect the properties of the system in different ways: Analyzability/Understandability, Changeability/Modifiablitiy[1], Re-usability, Testability, Traceability, Compliance to standards, and Process Qualities. These qualities are then divided further, and Bode & Riebisch defines modifiability to consist of: **Extensibility**, Variability, and Portability. While many of the qualities discussed here are outside of the scope of this project, it is still worth noting the relationship between extensibility and modifiability.

In Breivold et al. (2007) – from which Bode & Riebisch derives their definition – extensibility is defined as as the system's ability to enable *the implementation of extensions* to expand or enhance the system with new features and capabilities, with minimal impact to the existing

[1]Changeability and modifiability seemingly gets used interchangeably in some architecture literature(Bode & Riebisch 2010). Others define modifiabilty as a subset of changeability(Adams 2015), and some put changeablitiy as a subset of Modifiability(ISO 2011). Since most other qualities mentioned together here are outside of our scope, we're seeing them as more or less interchangeable.

system. Furthermore, they specify that extensibility is a system design principle where the implementation explicitly takes future growth into consideration.

Returning to the connection between modifiability and extensibility, it might be too simple to conclude that it is a direct subset of modifiability, but Breivold et al. (2007) notes that they have a fairly strong correlation, as any change made when improving extensibility will be justified through modifiability.

From these definitions, we define modifiability and extensibility as follows:

- **Modifiability:** The degree to which a system or program enables a modification to be quickly and cost-effectively implemented, without introducing defects or degrading the existing product quality (based on ISO (2011) and Bode & Riebisch (2010)).
- **Extensibility:** The degree to which a system or program enables the implementation of *new* capabilities, with little to no changes to existing code and without affecting existing capabilities (based on Bode & Riebisch (2010) and Hillard (2020)).

2.2 Supporting Extensibility

2.2.1 Characteristics of Extension Mechanisms

With a definition in place, it is now necessary to define a general body of knowledge to better understand how to develop an architecture to support the quality. When discussing extensibility in a system, there are three key terms that can be used: “extension points”, “extensions”, and “extension mechanisms” (Klatt & Krogmann 2008). An extension point is some interface defining how an extension interacts with the system. An extension is an implementation conforming to the extension point. Lastly, an extension mechanism denotes some explicit extensible part of the system.

Klatt & Krogmann further states that extension mechanisms can be characterized by 12 top-level characteristics, as shown in Figure 2.2. While this give a nice baseline for ways to discuss a systems extensibility, I’ve decided to exclud the Selection, Certification, and Repository characteristics in the further discussion, as I would argue these characteristics are strictly relevant to the external distribution of extension resources, and not relevant for the mechanism’s technical implementation. Briefly, the included characteristics are defined as follows:

- **System Access:** The mechanism *must* define the degree to which extensions has access the system.

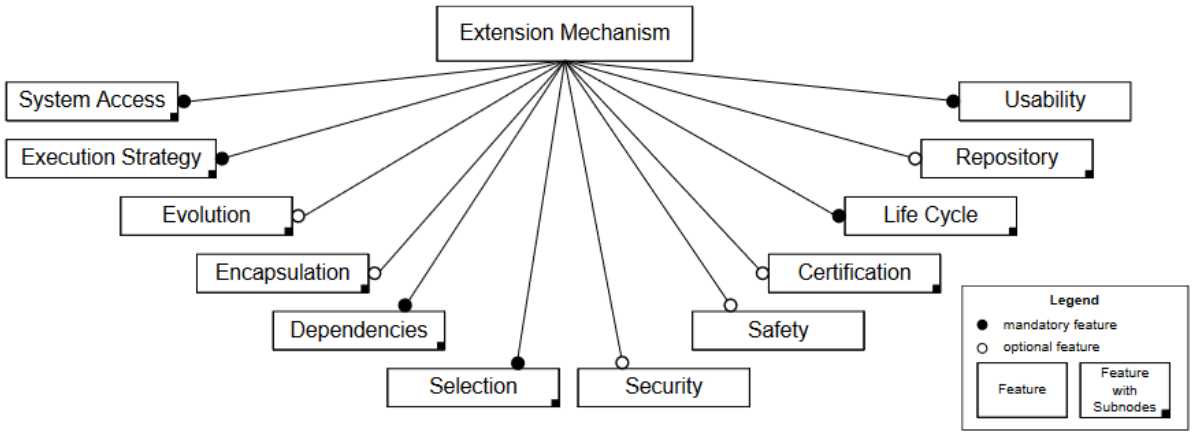


Figure 2.2: Key characteristics of extensibility mechanisms (Klatt & Krogmann 2008)

- **Execution Strategy:** The mechanism *must* either have extension be executed by the system (event calls), or have the extensions themselves execute the system (APIs).
- **Evolution:** As the system evolves, the mechanisms *can* impose rules to ensure that modifications to extension points do not invalidate existing extensions (e.g. versioning rules).
- **Encapsulation:** The mechanism *can* have extensions be separated from each-other and the core system, where extension points define the interface between the extensions and core system.
- **Dependencies:** The mechanism *must* implement logic to avoid complex dependency issues when extensions use the same resources.
- **Security:** The mechanism *can* impose rules to avoid bad extension code crashing the core system.
- **Safety:** The mechanism *can* impose rules to limit what internal data is accessible to extensions.
- **Life Cycle:** The mechanism *must* consider various states of an extensions life-cycle, including installation, loading, execution, updating, unloading, and installation.
- **Usability:** The mechanism's usage *must* attempt to be as intuitive as possible. If developers can use and understand the mechanism with little training effort, it is more likely to be used.

In addition to these key characteristics, extension mechanisms can also be classified based on which artifacts are changed and how they are changed, distinguished as three different

categories of extensibility: White-box, gray-box, and black-box extensibility (Zenger 2004, Aly et al. 2012).

With white-box extensibility, extension developers are provided full access to the core system's source code. This category is further divided into Open-box and Glass-box, based on whether the extension developer is allowed to modify the provided source. With open-box, the extension developer is can modify the source code as needed to accommodate their new extension, whereas with glass-box, the source code is visible, but not modifiable.

In contrast to white-box extensibility, black-box extensibility refers to a system being extended without any knowledge about the internals of the target system. Any extensions are developed against the original system's interface specifications. In designing such a system, one often needs to anticipate all possible extension scenarios, causing black-box approaches to often be more limited than white-box ones. However, since less knowledge about the internal system details is required, extension development is often easier to perform.

Lastly, Gray-box extensibility exists as a middle ground between black-box and white-box extensibility. In a gray-box system, developers of extensions can work from a more abstract system documentation, which lists artifacts available for refinement, and how extensions interact with the original system. Unlike white-box, gray-box systems do not rely on a full exposure of the source code, but unlike black-box, it still provides some internal details about the implementation and execution of the system.

2.2.2 Extensibility and the project life-cycle

Due to its close relation to extensibility, it can also be relevant to look at architectural approaches pertaining to modifiability. Being a more broader term, there is generally more literature discussing this quality, but we must keep in mind that the degree to which these mechanisms will affect extensibility can be limited. Bass et al. (2012) shows two areas one can improve modifiability: reducing complexity – by splitting modules, reducing coupling, and increasing cohesion; and having changes occur later in the life-cycle – by defering bindings.

In regards to binding, the thought here is that later in the life-cycle values are bound, the cheaper changes will be over time. However, creating these mechanisms has a higher cost the later in the life-cycle they occur, creating a trade-off. One can consider there to be five stages of the project life-cycle, each with different supporting mechanisms (Bachmann et al. 2007, Bass et al. 2012):

- **Code time** using aspect-oriented programming, polymorphism, and module parameterization

- **Compile/Build time** using component replacement, compile parameters, and aspects.
- **Deployment time** using configuration bindings.
- **Initialization time** using resource files.
- **Run-time** using run-time registration, plugins, publish subscriber, parameters, and so on.

Knowing that extensibility is strongly correlated to modifiability, it stands to reason that the same life-cycle binding categorization should apply to mechanisms that affect extensibility, as well as the same trade-off – higher up-front cost, lower cost down the line. This means that we must take care to avoid spending time and resources on late binding modifiability mechanisms that end up not having a valuable affect on extensibility.

2.2.3 Separating Extensibility from Modifiability

As discussed previously, one could arguably view extensibility as a subset of modifiability: An addition of new capabilities will be realized by some modification to the resources of the application, be it modifying the source files directly during code-time, or adding plugin resources during run-time. If the aim is to explicitly improve a system's extensibility, we need to more concretely define how one can separate a modifiability mechanism from an extensibility mechanism.

Szyperski (1996) and Hillard (2020) shows that one can claim that even simple open-box modifications of source files can be extensible, since it is – per the definition – an extension as long as a *new* capability is added. But for such extensions, the difference between modifiability and extensibility becomes negligible, and therefore not of particular interest as an explicit extension mechanism.

Klatt & Krogmann (2008) argues that if an extension mechanism does not express the encapsulation characteristic, extensions are just direct code manipulations. From this, I'd argue that the most concrete manner one can separate general modifications from an explicit extension is if the mechanism implements this key characteristic in some form.

The Encapsulation Characteristic

Klatt & Krogmann defines the encapsulation characteristic by three main aspects – namespace, extension points, and location – as shown in Figure 2.3. The namespace aspect provides an identifier for the extension's artifacts, such as objects, configurations and other elements. The location aspect also supports identification, but more-so in the physical sense, as the extensions resources should be located together in the system.

The extension point is arguably the most significant of these three. The extension point repre-

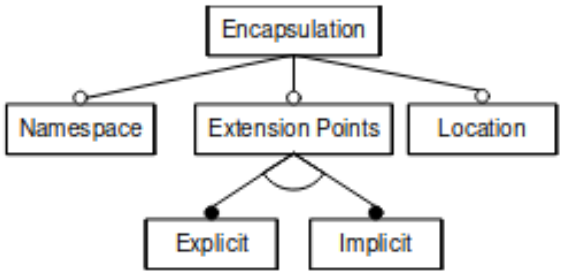


Figure 2.3: The elements of the Encapsulation characteristic (Klatt & Krogmann 2008).

sents a well defined interface between the extended system and it's extensions. This aspect is the main way in which the extension code is separated from the core system. If the extension points are explicitly defined, the core system provides facilities one must use to introduce and connect to an extension point. On the other hand, extension points can be implicitly defined, where the points are described informally in documentation, but not directly enforced. Additionally, an interesting note regarding extension points presented by Rytter & Jørgensen (2010), is that while a system with some set of extension points can be considered extensible, a system's extensibility is also affected by how difficult it is to introduce *new* extension points: if a new extension point *cannot* be introduced, the system has in a sense failed to be extensible.

An example of encapsulation

To showcase how encapsulation helps us separate modifiability and extensibility, imagine the following example: We have a very basic, single-file, calculator application where two numbers can be added together. To extend the functionality of this calculator, you want to implement logic to support subtractions.

If there is no extension mechanism, the new logic is added by writing the needed code directly into the source file. This *has* added new capability to the application, but as there is no encapsulation of the extension code whatsoever, and no identifiable extension point between the new and old code. As this extension has no discernible difference from a general code modification, you would not claim that this version of the program is extensible (it might be modifiable and structured well, but not noticeably extensible).

To improve the extensibility of our calculator, we now want to introduce some extension points, to allow some clearer distinctions between extensions and the system itself. If our application is written in some object oriented language (e.g. Java), we can make use of polymorphic inheritance. By creating an abstract base class that all calculator operations *must* implement, the abstract methods of this class provides the explicit extension points where extension code

can affect the system (Szyperski 1996). This is of course crude, but some encapsulation is now present, as it is at least *implied* that extension developers should encapsulate their code within this class. This encapsulation is still fairly weak, and the core application must still be modified to make use of the new class.

While allowing extension developers to directly change the system's source code allows for more flexibility in what an extension can contribute to the system, it generally introduces a lot of overhead: extension developers are required to understand more of the whole system to understand how their code will run, bugs more likely to occur, changes might unexpectedly propagate to many parts of the code, and extension developers might cause changes that invalidate the capabilities of other existing extensions. While it might be more restrictive, implementing a mechanism that offers strong encapsulation doesn't only improve extensibility, it also mitigates these issues, lessening the overhead for extension developers.

As a last example of the calculator application, you have now decided that instead of *just* making a calculator, you want to create a framework[2] that can be used to make a wider range of advanced calculators. You framework provides a basic run-time, which loads operator classes (which extends the aforementioned base class) according to some configuration file. The framework could also handle all frontend logic, so that extension code is limited to only concrete mathematical logic. In this case, the configuration file (or a required method in the operator class) would simply provide a operation name, and an icon to show in it's respective button. This lessens the overhead for developers, as they only need to focus on their exact task (the mathematical operation), but do not need to know about the specifics of the User Interface (UI) logic. Here, the extension points are defined in the same way as the previous example, but extension developers do not touch the core run-time. The framework can also impose rules about namespaces and locations to further improve the encapsulations.

2.3 Extensibility in practice: Plugin Architecture

When looking for "extensible" applications in the current software eco-system, it seems as though much of the software discussed as "extensible" are applications that realizes extensibility by means of a plugin mechanism – seen in examples presented in Klatt & Krogmann (2008), but also in surface level conversations on various online articles and forums. This section will provide a definition of this mechanism, and take a look at it's implementation in some these applications.

It must be noted that when looking for a succinct definition of the plugin mechanism itself, I

[2]The term "Framework" can be somewhat ambiguous, but like Klatt & Krogmann we define it as a collection of libraries that provides a basic application on it's own.

struggled to find one in relevant academic literature. One definition was found in Rice & Foemmel (2012), but it did not quite fit the mechanism we see implemented in the example applications. There is however countless articles from both professional and unprofessional blogs by software industry experts, as well as descriptions in the documentation of systems that implement plugin mechanisms. As such, the definition presented in this section here will be based on commonalities from these non-academic sources, as opposed to an academic text.

Plugins are (commonly) a run-time mechanism, where the software itself consists of some core platform with a specific functionality, where additional functionality can be added to the system later as needed, without directly affecting the core platform. Sayfan (2017) even goes so far as to claim plugin-based extensibility as the best-practice to extend systems in a safe manner, whilst promoting separation of concerns, and allows system-extensions to be built without risking destabilizing the core itself. Many of systems using this mechanism also have it be part of the core user experience, with the end user able to select new capabilities from large collections of available plugins, as opposed to the application developers themselves selecting the plugins.

Some popular software where one sees this mechanism includes – among many other – most modern web-browsers, as-well as various code editors and Integrated Development Environment (IDE) software such as Visual Studio Code and Eclipse. The exact way these applications implement their plugin systems is covered later in this chapter.

2.3.1 Structure of a Plugin Architecture

The basis of a plugin architecture is to some degree, simple. It generally consists of three loosely coupled parts: The core application, a plugin manager, and the plugins themselves (Sayfan 2017).

At a high level, the core application defines how the system itself operates, controlling the dataflow between plugins, and other internal logic you would not want plugins to deal with. The core can generally be anything you want, the connections to the external plugins and what you allow them to extend are what defines the plugin architecture design, usually taking shape as some list of functions that plugin can – or must – implement to be allowed to be plugged in (Apple 2013). Usually, the core also consists of some common logic which plugins can use to lessen code duplication, such as a shared logger object or debug methods.

The plugin manager is the part of the core system that handles the plugins. It provides the logic to find and initialize plugins, and should separate the plugins from the core – in fact, the core should ideally not even be aware that any plugins exist. Preferably, the plugin manager should also have logic to validate that plugins are allowed to be plugged in, for example making sure

certain interfaces are implemented.

Lastly, we have the plugins themselves, which provide the core with new functionality. Generally, these can be considered as full sub-systems in their own right, and can be developed however the extension developer sees fit (Szyperski 1996). As long as the plugin adheres to the rules set by the plugin manager, the plugin internals are of no interest to the core system. There are of course ways one should approach plugin development to ensure it can be called a “good plugin". Simply writing solid and efficient code obviously provides a good plugin, but it’s also worth considering the plugins purpose: Plugins should preferably only serve a very specific purpose, the key idea being that that this provides extensibility, flexibility, and stronger cohesion to the overall system. Additionally, the system and plugins should be loosely coupled to other plugins. You might have *some* dependencies to other plugins, but in an implementation, we should be able to add and remove plugins at will, with little to no effect on other plugins or the core system (Elgabry 2019).

Being a pattern, the exact way a plugin architecture is implemented will vary from software to software. To provide some more concrete examples of how this can be done, we will look closer at three applications: The Firefox web-browser, The Visual Studio Code editor, and the Eclipse IDE. When looking at these examples, we will look at the plugin systems from three different perspectives: The end-user, who discovers and installs plugins; the plugin developer, who creates new functionality; and the system developer, who creates new extension points and maintains the plugin manager.

The end-user perspective is to some extent less interesting for the discussion of architecture, but the experience is still worth mentioning, as this will be how most IT1901 staff interacts with the system. New functionality will be developed as plugins, so naturally the plugin development experience is perhaps the most important to ensure that development is easy. During this thesis, I myself act as the system developer, so insight here is valuable. Furthermore, as it’s likely that new extension points are needed later, this perspective also relates to the course staff maintaining the system after I’ve stepped away from it.

2.3.2 Example 1: Firefox Browser Extensions

As of today, most modern web-browsers implement some mechanism that provides their users with a way to customize and extend their browser, often realized by a plugin architecture. In this text, we will specifically be looking at the plugin mechanism found in the Firefox browser. The Firefox browser is open-sourced which provides insight into the specifics of the core system, and also fairly well documented, Additionally, I myself have some experience writing extensions for Firefox, which should make it easier to understand the documentation and plugin developer perspective.

As expected of a browser, the core system of Firefox provides functionality to display and navigate web-pages, whilst also providing additional features such as a search bar, tabs, bookmark, and so on. As these core features themselves aren't that relevant, I will not delve further in to the details here. The interesting aspect of the core system is the plugin manager implementation. The general architectural structure can be found in Figure 2.4. Most of the program is written in C++, however a large portion of the user interface is written in JavaScript and XUL (Grosskurth & Godfrey 2006).

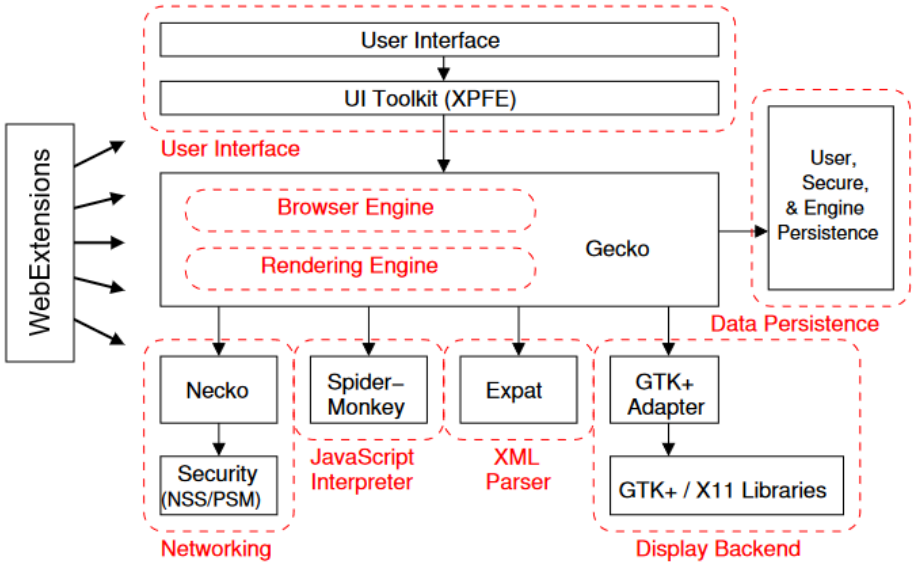


Figure 2.4: The general architecture of the Firefox Browser (Grosskurth & Godfrey 2006). Modified to showcase the WebExtensions implementation

Currently, Firefox supports "browser extensions[3]" with an implementation of the WebExtensions framework (Mozilla n.d.*b*). The WebExtensions framework accesses the internals of the browser at different levels of the architecture, and provides extensions with an abstracted access via various JavaScript Application Programming Interfaces (APIs). Currently there are 49 distinct APIs (Mozilla n.d.*a*), each providing extensions with access to a specific part of the browser's system (Tabs, window, storage, content scripts, and so on).

Starting at the perspective of the system developer, the plugin management system itself promotes extensibility to a certain degree, as adding new extension points has been made fairly easy. Every WebExtension API is represented by an instance of the JavaScript `ExtensionAPI` class, which contains all functions and events it makes available to their extensions. The frame-

[3]Firefox uses some different terminology regarding their plugin systems. They use the word "Plugin" when referring to the now deprecated system for adding support for new media-types. "Extensions" refer to the current system that we are looking at.

work keeps track of any available API using a simple list of JSON defined objects, and implementing a new API only requires you to add it's object definition to the list. See the example fragment in Listing 2.1. The API will then be lazily loaded into the system whenever an extensions that requests access to it is activated. Your new extension point will most likely require *some* modifications to the core, as you might need it to respond to some new event, or provide access to previously inaccessible data. Even so, the WebExtensions Framework does seem to offer extension points themselves a fairly strong separation from the core, potentially making these changes a low-effort job. In the ideal case, you might just be exposing some data or function that is already internally available, meaning adding the new extension point happens purely within the WebExtensions Framework configuration, and your API's own files.

```
{
  // ... other declarations
  "myapi": {
    "schema": "chrome://extensions/content/schemas/myapi.json",
    "url": "chrome://extensions/content/ext-myapi.js",
    "paths": [
      ["myapi"],
      ["anothernamespace", "subproperty"]
    ],
    "scopes": ["addon_parent"],
    "permissions": ["myapi"],
    "manifest": ["myapi_key"],
    "events": ["update", "uninstall"]
  }
}
```

Listing 2.1: Declaring a new Firefox JavaScript API

To show a real example of how an API can be implemented, see Listing 2.2. The listing shows the implementation for the Content Script API, which gives an extension the ability to inject JavaScript, HTML and CSS into web-pages. For brevity, only the `register()` function is shown. This registration function performs some validity checks, and then broadcasts a message to the core system that some new extension has scripts that it wants injected. The core system itself implements a handler for the `Extension:RegisterContentScript` event to perform the actual injection.

Lastly, we have the extensions themselves. An extension is a collection of CSS, JavaScript, and HTML files, along with various extra resource files. The extension itself is defined by a single JSON file – `manifest.json` – which must be present in every extension. This file contains metadata about the extension, such as it's name, version, and the permissions it requires. The

```
this.contentScripts = class extends ExtensionAPI {
  getAPI(context) {
    // ...
    // Internal logic and callbacks
    // ...
    return {
      contentScripts: {
        async register(details) {
          // ... Permission checks
          const contentScript = new ContentScriptParent({ context, details });
          const { scriptId } = contentScript;

          parentScriptsMap.set(scriptId, contentScript);

          const scriptOptions = contentScript.serialize();

          await extension.broadcast("Extension:RegisterContentScript", {
            id: extension.id,
            options: scriptOptions,
            scriptId,
          });

          extension.registeredContentScripts.set(scriptId, scriptOptions);
          extension.updateContentScripts();

          return scriptId;
        },
        async unregister(scriptId) {
          // ...
        }
      }
    }
  }
}
```

Listing 2.2: Registration of a content script in the Content Scripts API. Internal logic deeper in the system listens for the `"Extension:RegisterContentScript"` event, and performs the actual injection.

manifest also contains pointers to the extension's other files, which in turn informs what environments to load the files into. See Figure 2.5 for an overview of the different file environments, and see Listing 2.3 for the basic manifest example presented in the WebExtensions documentation.

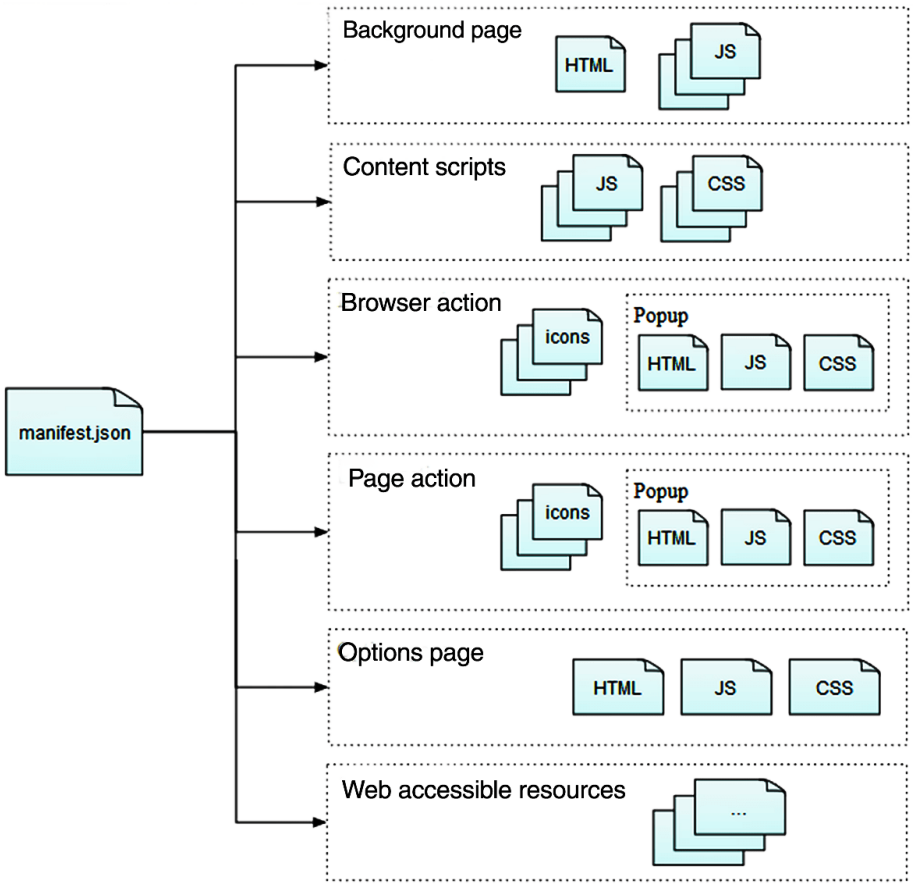


Figure 2.5: The anatomy of a Firefox extension

The different environments extension files can be in are:

- **Background:** Scripts that run in the background, independent from web-pages or browser window. Loaded with the extension, and stays loaded until the extension is disabled or uninstalled.
- **Content:** Injected into specified web-pages. Can be CSS or JavaScript files.
- **Page Action:** An icon added to the browser URL bar, with an optional popup. Appears only when visiting specified URLs. Supplies events to the background scripts.

```
{
  "manifest_version": 2,
  "name": "Borderify",
  "version": "1.0",

  "description": "Adds a red border to all webpages matching mozilla.org.",
  "icons": {
    "48": "icons/border-48.png"
  },

  "content_scripts": [
    {
      "matches": ["*://*.mozilla.org/*"],
      "js": ["borderify.js"]
    }
  ]
}
```

Listing 2.3: Example manifest file for a Firefox browser extension, which injects a script into `mozilla.org` pages

- **Browser Action:** Like page actions, but appears regardless of the current page.
- **Options Page:** A page for modifying the extension's preferences.
- **Web Accessible Resources:** Various resource files available for page and content scripts.

In the previous manifest example, the `"content_script"` key declares that the extension wishes to inject the `borderify.js` script file into any website matching the `"*://*.mozilla.org/*"` URL-pattern.

Extension scripts get access to the available WebExtensions APIs via an object called `browser` – similar to the `window` and `document` values available in regular JavaScript. Based on the environments, keys and permissions defined in the extension manifest, the WebExtensions Framework instantiates and appends the accessible APIs to the `browser` object. The APIs that should be made accessible is determined by fields used to register it in the Framework. Looking back at Listing 2.1 for an example, that API can be accessed by using `browser.myapi` or `browser.anothernamespace.subproperty`.

The `scope` and `permissions` keys adds additional restrictions to API access: `scope` defines which environments it can be accessed in, and `permissions` requires extensions to explicitly require access in their manifest.

For the MyAPI example, the `browser.myapi` path is available only for extensions listing `myapi` in their manifest permissions, and is only loaded if the extension script is running within the `addon_parent` scope. The `addon_parent` scope encapsulates the various extension pages (background, page, browser, and options environments), meaning the API is only available for scripts running in those pages.

Developing a Firefox extension also appears to be fairly easy – depending on the extensions purpose, of course. A developer will need to garner some understanding of the available APIs and the manifest structure, but apart from that, the extension content can be developed however the developer wants: There are no rules regarding the internal structure, the system just runs the scripts that the manifest points to. Of course, not making use of any internal browser functions will probably not make for an interesting plugin.

Distributing the extension after development can happen in two ways: Publicly available on the Firefox add-on website (commonly referred to as AMO), or distributing it yourself. The installation process for self-distribution is more complicated – mostly due to security concerns – but that is not really relevant for the architecture itself. The process for the end-user is generally the same regardless, the only difference being the location of the extensions installation package, and which buttons the user must click. In the end, the process for the end-user is straight-forward: The user finds the extension they want, installs it with the applicable menu, and confirms any permissions the extension demands.

2.3.3 Example 2: Visual Studio Code Extensions

Microsoft's Visual Studio Code is an open-sourced[4] code editor built upon the Electron Framework (OpenJS Foundation n.d.), and is written in TypeScript, JavaScript, HTML, and CSS. Strictly speaking, the application itself is fairly light-weight: the core features offered by the application include a text-editor with integrated IntelliSense[5], file search/navigation, a debugger, and built-in Version Control support. Additionally, the software has implemented a plugin architecture, which gives users the ability to find and install run-time Extensions (Microsoft 2021*b*). In-fact, most of the aforementioned core features are themselves developed and included as pre-installed extensions, utilizing the exact same API as any other external extension (Microsoft 2021*a*).

As with Firefox, the extension experience for the end-user is straight forward: The user looks for extension in the VS Code extension marketplace – which the program has a built-in view for – or they point the program to a local `.vsix` package-file. In either case, the system then validates

[4] To be precise, the version of VS Code released by Microsoft is in-fact *not* open-sourced, but the base code it's built upon *is*, so in the context of this thesis, it's practically open-sourced.

[5] IntelliSense is a broader term for many code editing features, such as: code completion, parameter info, etc.

and installs the extension, along with any potential dependencies. As extensions are run in separate processes, extensions can be dynamically installed and uninstalled without needing to restart the system.

The extensions themselves are developed in Typescript, and uses the VS Code API – a set of thirteen JavaScript APIs – to communicate with VS Code and other extensions. Similarly to Firefox, an extension defines their connection to VS Code using a single manifest file. However, the structure and manifest-keys differ. Some notable differences include:

- VS Code extensions do not need to request access to distinct parts of the API. Any and all JavaScript API is available to the extension.
- Whereas Firefox extensions define sets of scripts to be loaded in to different environments, VS Code's extensions more directly define exactly when certain callbacks should be run, in the form of Activation Events and Contribution points.
- VS Code extensions must point to a single main file – a JavaScript file with an `activate` and `deactivate` method, whereas Firefox extensions are an assorted set of files for different environments – none of which are required.

An extension's Activation Events is a list of platform events that will cause the extension to activate, such as when a specific command is run, or a file of a certain language is opened. When this event happens, the plugin system runs the main file's activation function. Note that this function only runs one time, once activated, the extension stays active until VS Code shuts down or the extension is disabled/uninstalled. Contribution Points is a map that defines which of the system's extension points the extension adds new functionality to, such as a new command or support for a new language.

As an example, the "Hello World" extension used in the VS Code API documentation is shown in Listing 2.4 and 2.5[6]. Notably, we see the use of the `command` contribution point, and the `onCommand` activation event. More precisely, this defines that the extension adds a new command named `helloworld.helloWorld`, which will be available for the user to invoke within the editor. The first time the user uses this command, the `onCommand:helloworld.helloWorld` activation event is triggered, causing the activation method in Listing 2.5 to be invoked, and the callback for the the command is then registered. After the activation method has finished, the command-callback invokes, and a message was displayed to the user via the `window` API.

Another thing worth noting, is the usage of the disposable pattern: the `helloworld.helloWorld` command is registered, but it needs to be removed from the global list of available commands if

[6]The whole extension structure consists of more files than these two, but these two files are the ones most relevant for this thesis' analysis.

```
{
   "name": "helloworld-sample",
   "displayName": "helloworld-sample",
   "description": "HelloWorld example for VS Code",
   "version": "0.0.1",
   "publisher": "vscode-samples",
   "repository":
   ↪  "https://github.com/microsoft/vscode-extension-samples/helloworld-sample",
   "engines": {
     "vscode": "^1.51.0"
   },
   "categories": ["Other"],
   "activationEvents": ["onCommand:helloworld.helloWorld"],
   "main": "./out/extension.js",
   "contributes": {
     "commands": [
       {
         "command": "helloworld.helloWorld",
         "title": "Hello World"
       }
     ]
   },
   "scripts": {
     "vscode:prepublish": "npm run compile",
     "compile": "tsc -p ./",
     "watch": "tsc -watch -p ./"
   },
   "devDependencies": {
     "@types/node": "^8.10.25",
     "@types/vscode": "^1.51.0",
     "tslint": "^5.16.0",
     "typescript": "^3.4.5"
   }
 }
```

Listing 2.4: Extension manifest for the VS Code “Hello World" example

the plugin is deactivated/uninstalled during run-time. The `context.subscriptions` array gives the VS Code platform a list of any so-called “disposables” that need to be cleaned up when an extension is deactivated (as opposed to having the extension manually clean it up within it’s deactivation method).

```
import * as vscode from 'vscode';

export function activate(context: vscode.ExtensionContext) {
  console.log('Congratulations, your extension "helloworld-sample" is now
  ↪  active!');

  let disposable = vscode.commands.registerCommand('helloworld.helloWorld', ()
  ↪  => {

    vscode.window.showInformationMessage('Hello World!');
  });

  context.subscriptions.push(disposable);
}

export function deactivate() {}
```

Listing 2.5: Extension source code for the VS Code “Hello World" example. (Comments omitted)

Lastly, we’ll look at the VS Code system itself. Simply put, VS Code consist of a layered and modular core that supplies certain extensions mechanisms, as shown in 2.6. The layers are defined as follows:

- **Base layer:** General utilities and UI building blocks.
- **Platform layer:** Service injection support, along with base services.
- **Editor layer:** The IDE’s text editor (Separated into the “Monaco" NodeJS project).
- **Workbench layer:** Main IDE view, loading in elements like the editor, status bar, etc.

Within each layer, the code is further organized by the target run-time environment (Microsoft 2020). The layered structure of the application, and strong rules for what layers each is allowed to communicate with implies both strong cohesion, lower coupling and defined separation of concerns. The different environments that can be accessed are:

- **Common:** Simple JavaScript APIs, and runs in all other environments.
- **Browser:** Browser APIs, such as the DOM.

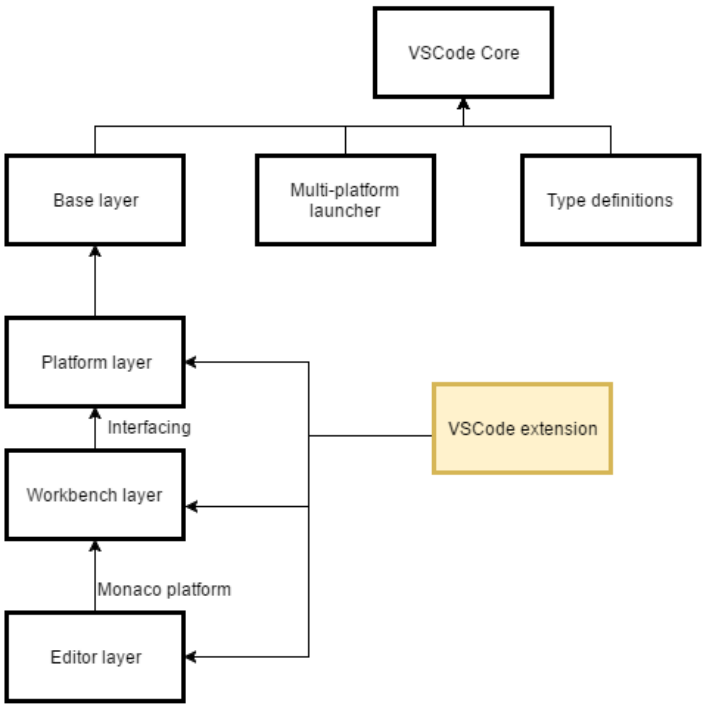


Figure 2.6: The module organization of Visual Studio Code (Schipper et al. 2017)

- **Node:** NodeJS APIs.
- **Electron-Sandbox:** APIs that communicate directly with select Electron APIs.
- **Electron-browser:** Electron renderer's APIs.
- **Electron-main:** Electron main-process APIs.

The code-base is organized around a service architecture, where the services and clients are connected via constructor injection. The services are defined by an interface and a identifier. An example of such a constructor can be found in Listing 2.6. The usage of constructor injection enforces a certain degree of flexibility to all new classes – and as discussed earlier, more flexible code is by effect more extensible. But, if one would measure extensibility by the amount of code affected by an added function, adding a *new* dependency for a class, you would need to add the code to inject the dependency in all places it has been constructed. VS Code implements a instantiating service that handles the specific injection, so this change should ideally only occur in one place, mitigating this issue somewhat.

Adding a new extension point is somewhat arduous. You define the namespace for your new API inside a large definition file (around 14300 lines), as shown in Listing 2.8. When an extension is being initialized, an internal service sets up and provides the extension with the actual

```
class Client {
  constructor(
    @IModelService modelService: IModelService,
    @optional(IEditorService) editorService: IEditorService
  ) {
    // use services
  }
}
```

Listing 2.6: Example usage of Constructor injection in VS Code

implementation of the VS Code API. This implementation is created by a large factory method (around 1300 lines). As the system currently stands, creating a new API requires that it's implementation is written inside this factory method – unlike Firefox's system where new a API's logic is mostly encapsulated within their own source files.

It should also be mentioned that VS Code extensions can define dependencies to other extensions in their manifest. This means that those extensions will be installed alongside the given extension. An extension can export it's own API when it activates, which in turn dependent extensions can access by lookup using VS Codes Extensions API.

2.3.4 Example 1: Eclipse Plugins

Eclipse is an open-sourced IDE written in Java, presented as "an integrated IDE for anything and nothing in particular". The application was not made to be a tool itself, but rather a platform where people could build any tools they wanted, resulting in an highly modular and scale-able architectural design (Brown & Wilson 2011).

To some extent, the Eclipse IDE is similar to VS Code in that it provides a lighter base platform of some core functionality, and has much of it's core functionality developed as normal extensions. The Eclipse architecture is shown in Figure 2.7, and as with VS Code, we see it simply as a core platform, with some connected plugins. However, where VS Code's platform is a more straight-forward layered architecture, Eclipse is more strongly built around the concept of plugins. The entire platform is driven by the Equinox platform runtime – an implementation based on the OSGi Plugin Framework (OSGi n.d.) – and each subsystem is built as a set of plugins implementing a key function, providing functionality or supplying libraries (Eclipse Foundation n.d.*a*). The plugin system handles them dynamically, and they can be installed, started, stopped, and uninstalled during run-time.

```
// ... other implementations
// ...
const extensions: typeof vscode.extensions = {
    getExtension(extensionId: string): vscode.Extension<any> | undefined {
        const desc = extensionRegistry.getExtensionDescription(extensionId);
        if (desc) {
            return new Extension(extensionService, extension.identifier, desc,
            ↪  extensionKind);
        }
        return undefined;
    },
    get all(): vscode.Extension<any>[] {
        return extensionRegistry.getAllExtensionDescriptions().map((desc) => new
        ↪  Extension(extensionService, extension.identifier, desc,
        ↪  extensionKind));
    },
    get onDidChange() {
        return extensionRegistry.onDidChange;
    }
};
// ... other implementations
// ...
```

Listing 2.7: Snippet: The implementation of the VS Code Extensions API

```
// ... other namespaces and defintions
// ...
export namespace extensions {
    /**
     * Get an extension by its full identifier in the form of: `publisher.name`.
     *
     * @param extensionId An extension identifier.
     * @return An extension or `undefined`.
     */
    export function getExtension(extensionId: string): Extension<any> |
    ↪  undefined;

    /**
     * Get an extension by its full identifier in the form of: `publisher.name`.
     *
     * @param extensionId An extension identifier.
     * @return An extension or `undefined`.
     */
    export function getExtension<T>(extensionId: string): Extension<T> |
    ↪  undefined;

    /**
     * All extensions currently known to the system.
     */
    export const all: readonly Extension<any>[];

    /**
     * An event which fires when `extensions.all` changes.
     * This can happen when extensions are installed, uninstalled, enabled or
↪   disabled.
     */
    export const onDidChange: Event<void>;
}
// ... other namespaces and defintions
// ...
```

Listing 2.8: Snippet: Namespace definition for an VS Code extension API

One can categorize the plugins that make up Eclipse into three groups: The Workbench IDE, which contains plugins related to IDE features like code-editing and so forth; the base Rich Client Platform (RCP), which contains the bare minimum set of plugins required to make a running program; and lastly a set of optional helper plugins for the RCP group, such as input and text elements. Using the RCP, one can create a vast amount of varied software, showcasing how the Eclipse platform provides extensibility at a more core level than that of the previous two examples. Regardless of plugins, VS Code will always – at it's core – be a code editor.

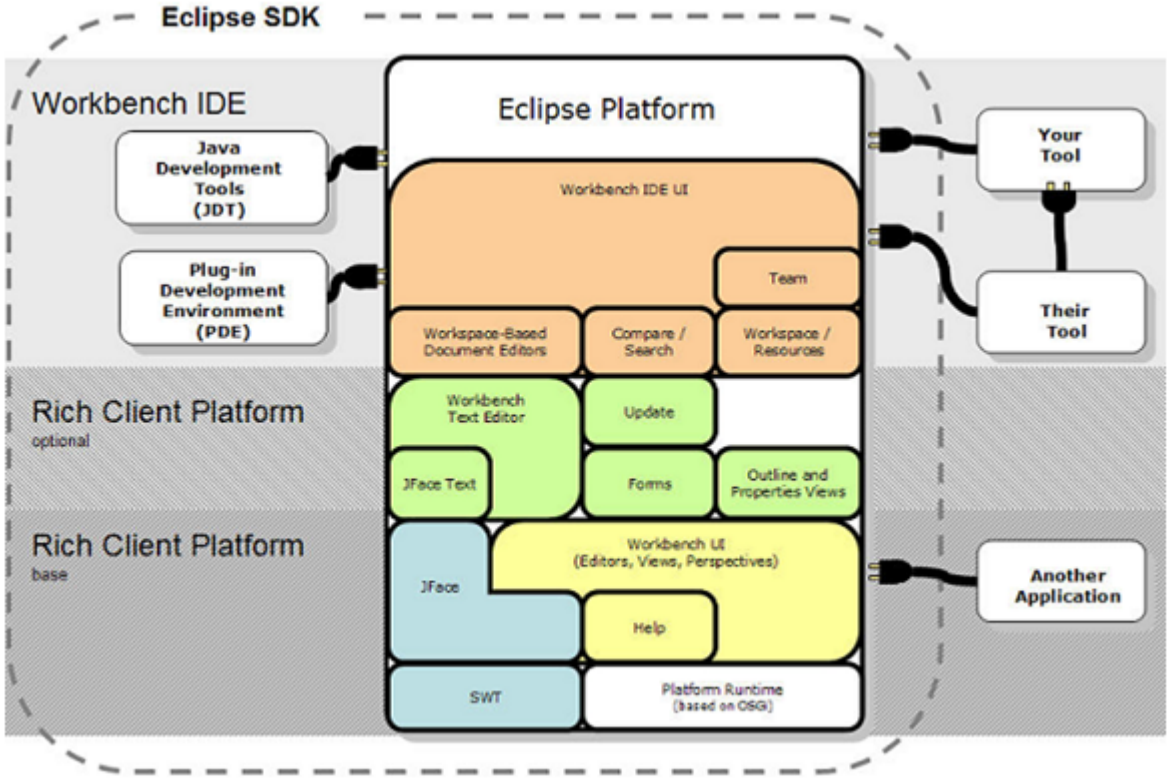


Figure 2.7: The Eclipse architecture (Tutorials Eye n.d.)

An Eclipse plugin is a collections of files packaged within a `.jar` file. As we've seen with Firefox and VS Code, the plugin describes itself using a manifest file, but unlike the others, Eclipse makes use of two separate manifest files: The bundle manifest file, `MANIFEST.mf`, and the plugin manifest file, `plugin.xml` (Eclipse Foundation n.d.*c*). The bundle manifest defines how the plugin gets registered in the Equinox plugin registry, providing metadata such as name, id and a version number. It also informs of what Java code the plugin supplies, and which other plugins it depends on. The plugin manifest defines exactly how it extends the available extension points and defines the plugin's own extension-points. The latter here is a notable difference to the Firefox and VS Code extensions: User created Eclipse plugins are as extensible as any internal plugins, as they can expose their own extension points in the same manner as one would an

“internal” extension point. VS Code somewhat achieves this with allowing extension lookup, but the workflow and structure here is arguably not equivalent to the platform’s own internal extension points.

To provide an example of an Eclipse plugin, I will use code examples from Blewitt (2013), which is a simple plugin that opens a “Hello World" dialogue box when a shortcut is pressed. Some code will be omitted for brevity. Firstly, Listing 2.9 shows the `MANIFEST.MF` file. Take note of the `Bundle-Activator` line, pointing to the Java class that will be run when the plugin activates, and the `Require-Bundle` line, establishing a dependency on the Eclipse UI and runtime. Listing 2.10 shows a snippet of the `plugin.xml`, where the manifest defines two extensions: a new UI event handler, which points to the `SampleHandler` class; and a new UI binding, which invokes the aforementioned handler when the `M1+6` sequence is input.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Hello
Bundle-SymbolicName: com.packtpub.e4.hello.ui;singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: com.packtpub.e4.hello.ui.Activator
Bundle-Vendor: PacktPub
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Bundle-ActivationPolicy: lazy
```

Listing 2.9: Eclipse HelloWorld bundle manifest

As for the Java classes, the activator in this example is less interesting, it’s simply extends the abstract class `AbstractUIPlugin`, which is provided by the core Eclipse UI plugin, and adds some required specifics. I will not provide a listing for it as it’s mostly boilerplate. The actual functionality is defined in the `SampleHandler` class, which is shown in Listing 2.11. This class extends an `AbstractHandler`, provided by the Eclipse core plugin, and is invoked when the events defined in the plugin manifest is detected in the system.

One thing not shown in Blewitt’s example, is the process of adding and exposing a new extension point. Listing 2.12 shows how a new extension point is defined in a plugin manifest, found in the Eclipse Nebula Repository (Eclipse Foundation n.d.*b*). The tag itself does not define the extension point, but points to a larger schema file that does. This schema file is fairly large, and usually you’d generate this using the plugin development tools available in Eclipse, instead of writing it by hand. This file will list the values, classes and methods you’d want your extension point to expose (along with relevant parameters and default values).

```
<extension point="org.eclipse.ui.handlers">
    <handler
        commandId="com.packtpub.e4.hello.ui.commands.sampleCommand"
        class="com.packtpub.e4.hello.ui.handlers.SampleHandler">
    </handler>
</extension>
<extension point="org.eclipse.ui.bindings">
    <key
        commandId="com.packtpub.e4.hello.ui.commands.sampleCommand"
        contextId="org.eclipse.ui.contexts.window"
        sequence="M1+6"
        schemeId="org.eclipse.ui.defaultAcceleratorConfiguration">
    </key>
</extension>
```

Listing 2.10: Snippet: Eclipse HelloWorld plugin manifest

```
// ... imports
public class SampleHandler extends AbstractHandler {
    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        IWorkbenchWindow window =
        ↪  HandlerUtil.getActiveWorkbenchWindowChecked(event);
        MessageDialog.openInformation(
            window.getShell(),
            "Hello",
            "Hello again, Eclipse world");
        return null;
    }
}
```

Listing 2.11: Snippet: Eclipse HelloWorld Sample Handler

```
<?xml version="1.0" encoding="utf-8"?>
<?eclipse version="3.0"?>
<plugin>
    <extension-point id="examples"
        name="\%extension-point.name"
        schema="schema/examples.exsd"/>

    <!-- ... extensions -->
</plugin>
```

Listing 2.12: Snippet: Eclipse Nebula extension point example

Eclipse also supports dependency injection as of version 4. VS Code also uses this mechanism internally, but their plugin manager does not provide it for extensions. This is not the case in Eclipse, since external plugins are handled the same as internal ones, which arguably does make writing extensions require less effort. The injections are defined using Java 5 annotations, as seen in Listing 2.13. Note that this annotation syntax more or less provides the same functionality as the constructor injections seen in VS Code, with the primary difference being that the annotation is scanned by the Equinox system at run-time. In a pure logical sense, the inversion of control is equivalent.

```
@Inject
IStatusLineManager statusLine;
// ...
statusLine.setMessage(msg);
```

Listing 2.13: Example of Eclipse/OSGi dependency injection

Because Eclipse's core platform is built up as a set of plugins, the development process for an external developer is generally the same as that of the system developer: you'd write a plugin as above, but in this case you would have it be shipped as part of the core system. Therefore, I will not repeat myself, and jump straight to the end-user perspective.

For the end-user, the installation process is the same as Firefox and VS Code: plugins are found on the Eclipse marketplace, downloaded and installed. An interesting thing to note however, is that Eclipse is more relaxed in terms of security/validation: The other two programs requires the use of a packaged and signed installation file, whilst Eclipse permits installations by simply adding a plugin `.jar` to the programs plugin folder.

3 | Supporting Extensibility in “Odin”

3.1 Context

The “Odin” web application was developed as part of a Master’s Thesis during the spring of 2021. The system provides an intermediary web-service that helps NTNU course staff manage student groups and repositories across the Blackboard LMS and Gitlab SCM services (Rein & Tefre 2021). As a thesis project, the existing system is currently a MVP, focusing on those core features that were possible to develop within the timeframe of a single semester.

For it’s development, the existing system has focused on two tasks commonly performed by course staff: creating and modifying student groups, where both need to be synced across these two services; and gaining insight into the work-performance of groups and their members by gathering and analyzing repository data such as commit counts, pull requests, and so on.

As it stands, the MVP offers the following core functionalities:

- **Group Management & Synchronization:** The system provides course staff with an overview of student groups in the course. It also allows staff to create, edit, and delete groups in a single location; synchronizing these changes across the two external systems.
- **Group Metrics:** The system provides visualizations of the groups activity in their respective repositories, which the staff can use when following up on the group’s work.

Figure 3.1 and 3.2 shows screenshots of *some* of the group management features: respectively the page for creating a new group, and the page for editing groups and synchronizing the members across Blackboard and Gitlab. A screenshot of the metric feature is show in Figure 3.3. During Rein & Tefre’s development, the group management feature were presented as their primary focus, as this solved the most pressing issue voiced by the course staff. Naturally, this means that of the two features, group management is the most fleshed out.

In contrast, the metric functionality more-so exists as a proof of concept, and the main issue with The Odin system’s metric functionality is that there was, and still is, uncertainty as to which exact metrics are needed when following up on students. This means that over the system’s lifetime, the exact requirements for the statistics are likely to evolve, and it must therefore be

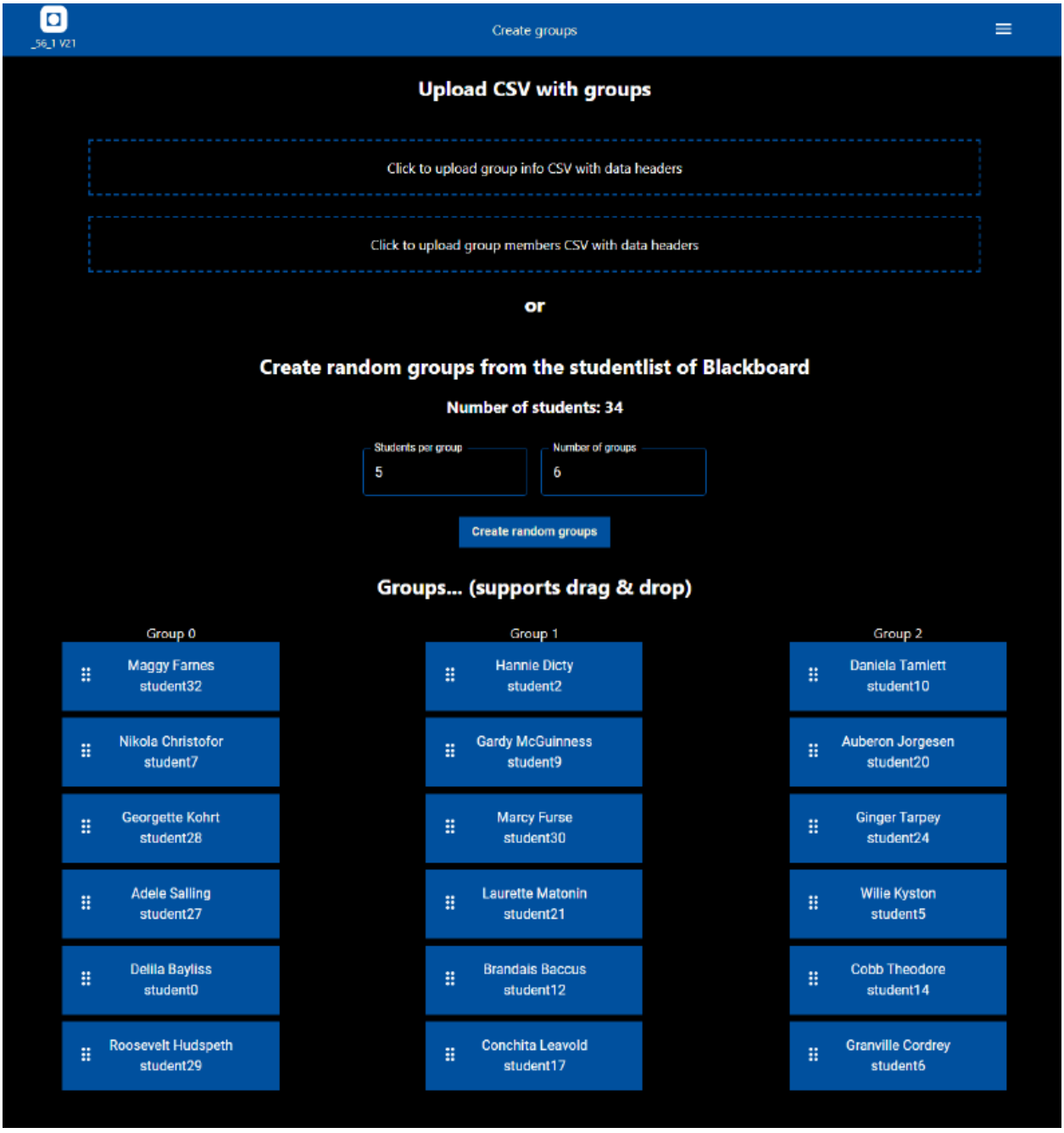


Figure 3.1: Screenshot the Odin system's Create Group page

Figure 3.2: Screenshot of the Odin system’s Edit Group page

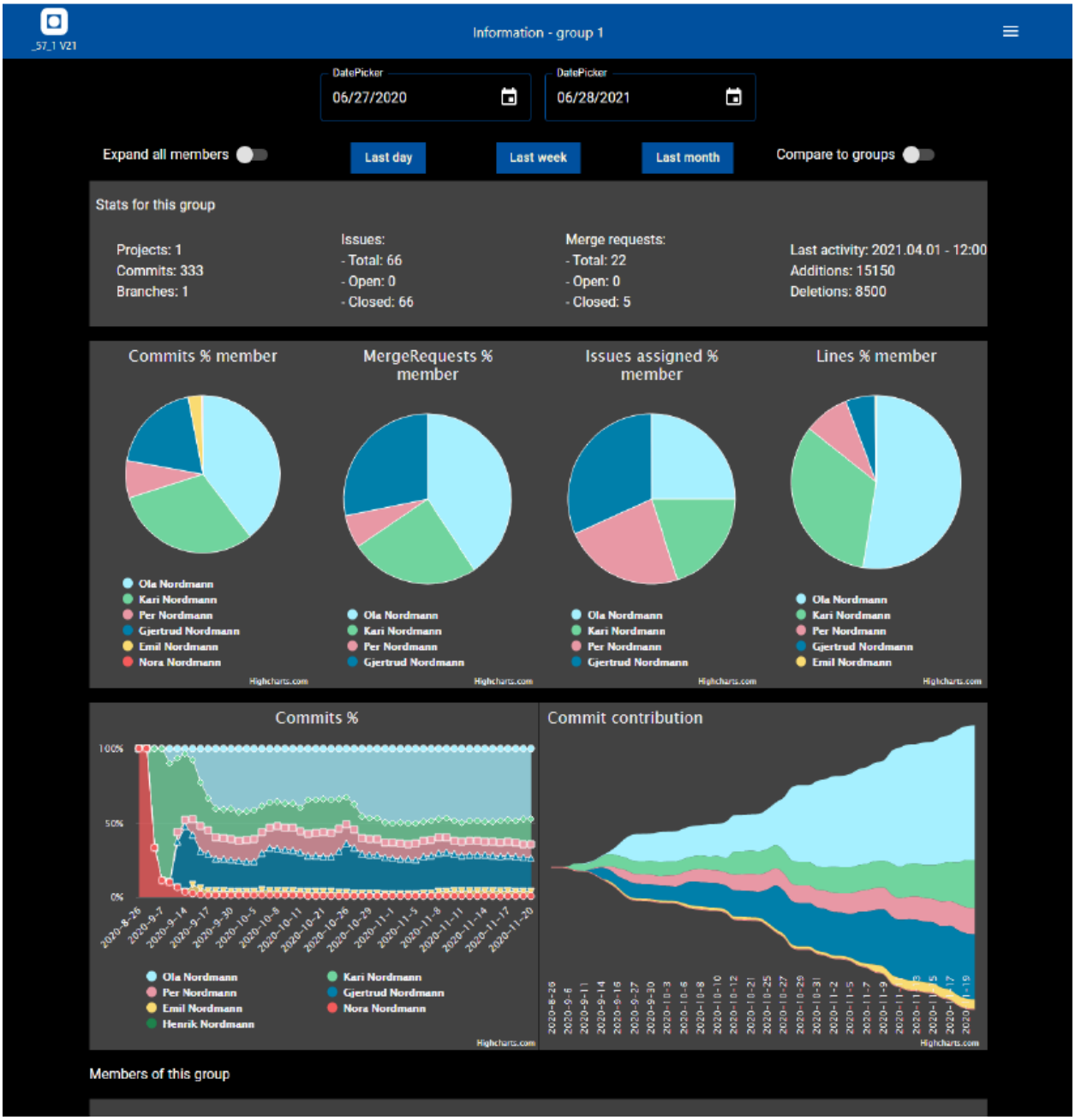


Figure 3.3: Screenshot of the Odin system's Group page metrics

easy to add and remove new aggregations and visualizations of the data available on Gitlab. Simply put, this is an aspect of the system that needs be extensible.

For my thesis I will evaluate the extensibility of the current system, especially as it pertains to the student group metric dashboard. Firstly, in section 3.2, I will present a more in-depth look at the current system, and it's development. For showcase the driving forces during it's development – more specifically driving forces in terms of architectural decisions – I will show the stakeholders defined in Rein & Tefre, along with the non-functional requirements they presented for the system. Afterwards, in Section 3.3, I will present the stakeholders relevant for *my* project, and describe the requirements that will direct the later design and implementation of an extensibility mechanism. Section 3.4 will discuss how the architectural decisions during Odin's development has affected the extensibility in the system – particularly in regards to implementing new metrics. Lastly, in section 3.5, I will present a design of a plugin system that will be implemented to improve the system's extensibility, making it easier to develop new metrics for the dashboard. The implementation of this plugin system is detailed in Chapter 4.

3.2 The current system

3.2.1 Stakeholders & Quality Concerns

In their thesis, Rein & Tefre states that the system stakeholders can be defined by three groups: The *end-users*, the *system developers*, and the *system administrators*. Here, the end-users consists of the course staff (Course coordinators, teaching assistants, and so on) using the system for group management. The administrators are the ones responsible for deploying and keeping the system running, and the developers are the ones implementing changes to the system. Note that these groups may overlap, where a course coordinator with development knowledge could be both a developer *and* an end-user. Furthermore, Rein & Tefre attributed specific system qualities to each of these groups, as shown in Table 3.1.

For their project, Rein & Tefre approached the development of Odin following a user-centered design process, selecting Usability as the main driving force for Odin's development. Furthermore, in the interest of ensuring that the system saw future use and development after their initial thesis project, maintainability was chosen as a secondary focus. This also supported making the system easy to tailor for different course and environments, which was a desired feature stated by their stakeholders.

From their initial stakeholder interviews and the quality attributes mappings from Table 3.1, Rein & Tefre defined a set of requirements for the system. As *functional* requirements are not relevant when making architectural decisions, I will not go into details about them, rather

Stakeholder Group	Quality attribute	Explanation
End-users	Usability	The application should be easy to learn and use for all users.
	Security	The information in the application should be safely stored and processed.
	Availability	The application should be available when the user needs it.
Developers	Modifiability	The implementation should be possible to change when needed.
System administrators	Interoperability	The system should exchange and store information in a meaningful way.

Table 3.1: Quality attribute concerns for stakeholders during Odin's development (Rein & Tefre 2021)

focusing on the *non-functional* ones. The non-functional requirements defined by Rein & Tefre is shown Table 3.2.

NF1, 3, 4, and 6 are requirements focused on visual design and access, likely stemming from usability being the main focus, whilst NF5, 7, 8 and 9 seem to stem from the secondary focus on maintainability. Lastly, NF2 seems to satisfy the system administrators interoperability concern. Notably, NF7 directly states the system should be maintainable (which they previously denotes as a part of modifiability). NF8 and 9 could potentially be considered some form of variability. As I will discuss later, the system's support of NF7 is the requirement most relevant regarding extensibility, but it useful to show all of them to provide the context of current system architecture.

3.2.2 System Architecture & Technology

The existing system is a web application developed with Next.js, a React framework, and the general architecture presented in Rein & Tefre's thesis is shown in Figure 3.4. As is usual for modern web-apps, the application separates into a frontend and a backend. Here, the frontend provides the user interface, and the backend is managing data and using functionality from external services. The services are Dataporten for authentication, the Blackboard LMS, and the Gitlab git-hosting site. An example of a process communicating across these external services is shown in Figure 3.5, which showcases the logic flow for when an end-user views and edits a student-group.

ID	Non-functional Requirement
NF1	Possible to access from everywhere. (On the bus, at home, at school)
NF2	Easy to host in a production environment.
NF3	User-friendly, easy to use, require little training to use.
NF4	Accessible for everybody, color blindness, etc.
NF5	Easy to set up and run for developers.
NF6	Can be used on the phone and/or the computer.
NF7	Maintainable for future development.
NF8	Support other LMS than Blackboard
NF9	Support other Git hosting platforms than Gitlab.

Table 3.2: Non-functional requirements defined during Odin's development (Rein & Tefre 2021)

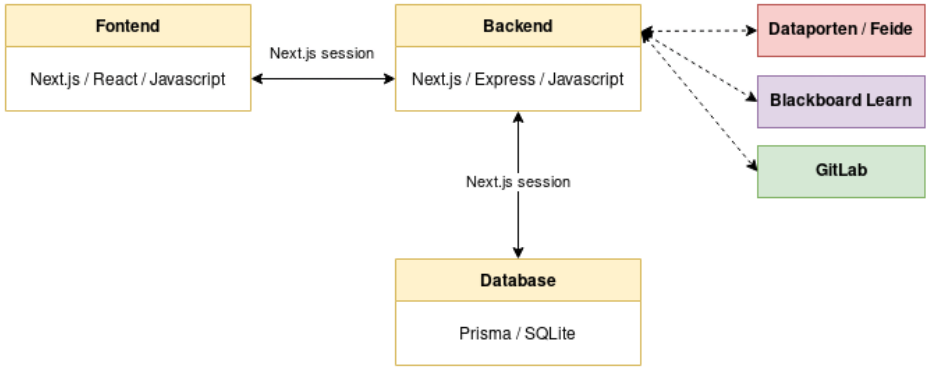


Figure 3.4: Architecture of the Odin system (Rein & Tefre 2021)

Rein & Tefre presents NF1 and 6 as the driving requirements for designing a web-application with a frontend/backend separation. As for the technology, React is a component-based JavaScript library, which Rein & Tefre reasons helps satisfy NF7 – as components offers strong separation, cohesion, and code reuse.

Supporting requirement NF8 and 9, the connections to Gitlab and Blackboard are wrapped and abstracted to make them loosely coupled from their respective service, and the data fetched and used within Odin is transformed into a more system-agnostic format. Although most data handled by the system is stored on the external services – the backend serving as a broker between them – a small amount is stored on the Odin server itself: The end-users' Personal Access Tokens (PATs) for Git Lab[1] and mappings between the Blackboard courses and their

[1]The PAT authenticate a Gitlab user, and is required to access private repositories.

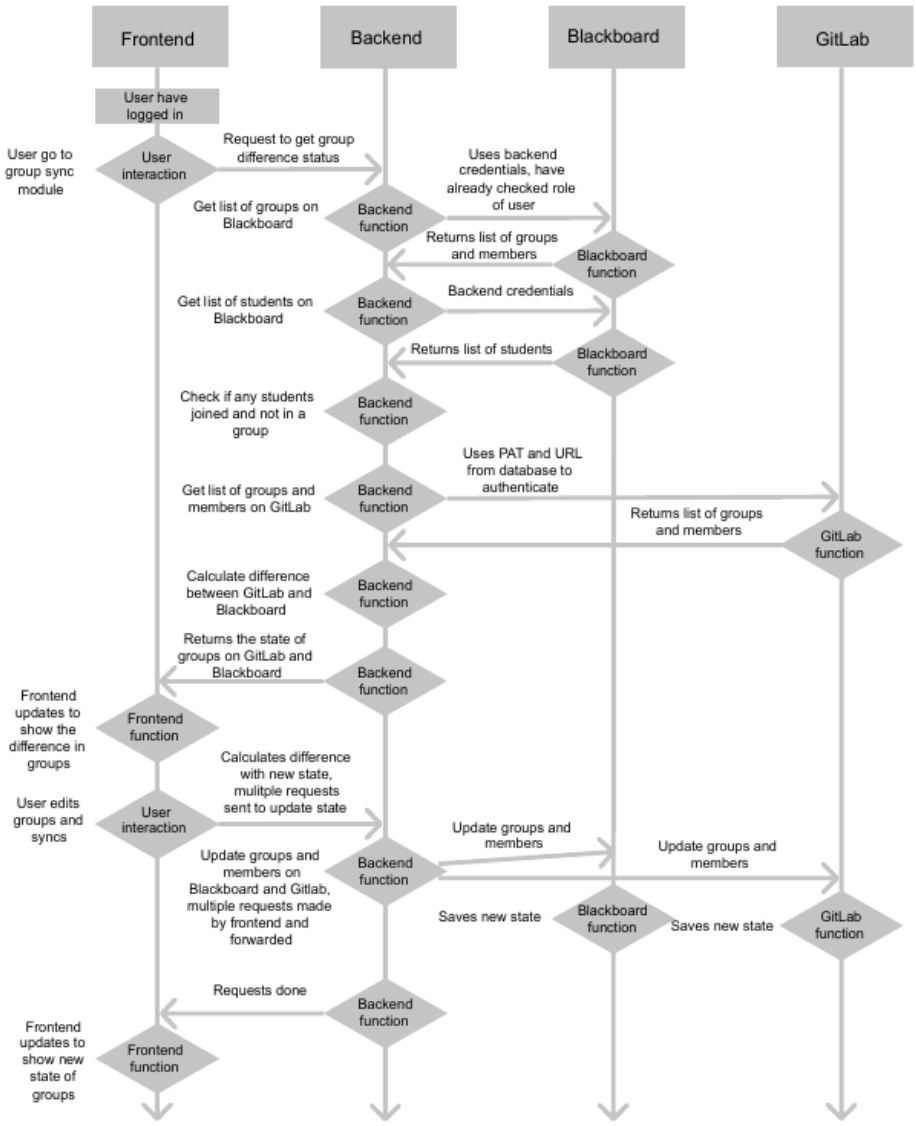


Figure 3.5: Flow for checking and updating a group’s state (Rein & Tefre 2021)

respective Gitlab groups and repositories.

Of these technologies, the Next.js framework is potentially the most important one, as it drives much of the architectural design and provides the underlying run-time for serving web-pages to connecting users. In addition to the more general server functionality one would expect from a web-app framework, Next.js also promotes itself by an extensive set of additional features. Some of the main features presented as selling points are: Server-side rendering, optimized code-splitting and bundling, file system routing, to mention a few (Vercel n.d.).

3.3 Stakeholders and requirements of an extensible system

As mentioned in the section above, there exists three different groups of stakeholders. However, when analyzing and modifying the system for the purpose of supporting extensibility, the stakeholder group most important is arguably that of the system developers, as they are the ones most directly affected by modifications to the the application architecture. The system administrators are also potentially affected, since a change to the current development flow could affect the deployment process.

End-users are less in focus, since ideally my changes should not affect the usability of the released product, as my changes are concerned with the internals of the system. The end-user might still provide insight into what parts should/must be extensible however, as they have insight into what new functionality might be needed down the line.

My supervisor is part of both the developer and end-user groups, as they are both a developer looking to improve the metric functionality, and a course coordinator using the system. Effectively, they are acting as a customer, who has hired me to produce a more extensible software solution. For this chapter, I will hereby refer to my supervisor as “The Customer" when discussing them in relation to the system.

With this assignment being a more exploratory look into a possible extensibility solution, no formal requirement process has been held to solicit requirements of a modified system. Reasoning behind what could be considered a “good” solution, that satisfies The Customer’s needs, is based on causal one-on-one conversations and supervision-meetings regarding my thesis, as well as my own opinion and developmental knowledge. Of course, this does invalidate my decisions to some degree, as I might go against the wishes of other system stakeholders.

3.3.1 Requirements of an extensible system

For the system to satisfy The Customer’s needs, extensibility *must* be supported specifically for the part of the system that deals with group metrics – which as of now consists of the dash-

board in Figure 3.3. This means that ensuring extensibility in other parts of the system is not a requirement (although a “best" solution should preferably be general enough to allow support of future needs for extensibility).

In addition to generally supporting extensibility, two non-functional requirements were voiced by The Customer during our conversations:

- **Strong separation between core system and extensions:** The Customer expressed that adding new functionality should not require any editing to the existing system code. This was expressed as something a solution should support to be considered a “good" solution.
- **Late life-cycle binding:** The Customer expressed that the solution preferably should allow functionality to be added later in the life-cycle so that there is no need to build and deploy a new version of the core system when introducing extensions. Restarting the system is not an issue, meaning that bindings should be done at initialization-time or run-time.

Lastly, The Customer requires that the existing core technologies remain the same. This means that a solution *must* be built within the Next.js framework, disallowing a rewrite that makes use of some alternative framework.

3.4 Extensibility in the current architecture

As extensibility has a connection to modifiability, the manner in which NF7 has been supported is the one most relevant to my thesis, both since some code-level extensibility is potentially already present, and the implementation of a more effective extensibility mechanism might be easier.

Unfortunately, reading Rein & Tefre’s thesis, the degree to which the NF7 affected architectural decisions seems limited, although this somewhat unsurprising considering usability was stated as the main focus. As far as I gleam from the text, the one architectural decision maintainability affected, was the choice to use React and the Next.js framework. Apart from this, the text has no mentions of any mechanisms explicitly implemented for the purpose of satisfying NF7. Because of this, I expect the support for extensibility to be limited to that which is indirectly caused by using React and Next.js. For instance, Next.js inherently provides some code/build-time extensibility by it’s file-system routing: Adding a new page to the application only requires adding a new JavaScript file within the `pages` directory (potentially within some amount of sub-directories), and the Next.js build process generates a new page with the logic from that file, with the page’s URL matching the file-path. Here, the we see an extension point for the internal

routing (and build) system, with the interface being simply “any file in the *path* folder.”

3.4.1 Extending the system as-is

To contextualize the extensibility of the current system, I will design some new metric that uses Gitlab data. My goal here is not to do an actual implementation, but to get a picture of how difficult such an implementation *would be*, particularly looking at how much of the source code must be changed to accommodate my proposed feature, and how the currently implemented mechanisms reduces (or increases) my implementation effort.

For brevity, I will keep the new functionality relatively simple: I want the group page (Figure 3.3) to display a simple counter to show how many commits the group wrote on a Sunday. Albeit not that useful of a metric, I’d argue that this reflects the base logic most metric functionality consists of, as it does the following:

- Data is requested from the Gitlab API, here as a list of every commit the group-members have pushed to group projects.
- The data is aggregated, here as the commits gets filtered by date and reduced into a single integer number.
- The aggregated data is displayed for the frontend user, here as a component with the text “N commits where done on a Sunday!"

Implementation

Currently, the metric data used by the groups dashboard is carried to the frontend within a single large object (Listing 3.1). This object is comprised of different data from various Gitlab API endpoints, which the Odin backend collects and combines (Listing 3.2). Since commit data is already available to the example extension no new API call is needed, only some simple filtering must be performed. Here, there are two options: I can calculate the needed aggregation in the `groupStats.js` file, or write it as part of the UI component’s internal logic. To follow the code organization promoted in the system, my aggregation would be written in the `groupStats` file, adding the field `sundayCommits` to the return object. Likewise, had it been the case that the example extension required some new API call, that change should also be placed here, with the data added to the return object

With the required data available, the next step is creating the UI element on the frontend. Currently, the various graphs visible are written inside a larger component, named `GroupStatsGraphs.jsx`. To handle the actual drawing of the graphs, the project makes use of an external Node package, named Highcharts (Highcharts n.d.). For example, the implementation of the

```
const mergeBBGitKeyStats = async (/* <params> */) => {
  const groupKeyStats = await fetcher(
    /* <gitlab api endpoint url> , */
    {},
    "GET"
  )
  return { ...courseGroupsBB, groupKeyStats }
}
```

Listing 3.1: Snippet: Odin frontend fetching the groups gitlab stats

```
// utils/gitlab/groupStats.js
const getGroupKeyStats = async (path, pat, fullPathGit, since, until, fileBlame)
↪  => {
  // ...
  // ... Init values
  // ...
  const groupStats = await /* <Gitlab GraphQL API call> */

  const commits = await /* <Gitlab REST API calls> */
  const branches = await /* <Gitlab REST API calls> */
  const wikiPages = await /* <Gitlab REST API calls> */
  const projectFiles = await /* <Gitlab REST API calls> */

  // ...
  // ... Various aggregations
  // ...

  return {
    ...groupStats,
    commits: commits,
    commitsCount: commits.length,
    branches: branches,
    wikiPages: wikiPages,
    contributorStats: contributorStats,
    commitStats: commitStats,
    projectStats: projectStats,
  }
}
```

Listing 3.2: Snippet: Odin backend fetching and combining data from different Gitlab APIs

pie chart showing commits per member is shown in Listing 3.3. Here the `Grid` component is used for placement purposes. The `HighChartsReact` component is used for every type of graph in the dashboard, with the calculated options defining chart-type, values, and so on.

```
{/* Other elements ... */}
<Grid
 container item
 direction="column"
 xs={12} md={3}
>
  <HighchartsReact
    highcharts={Highcharts}
    containerProps={{ style: { width: "100%", height: "100%" } }}
    options={optionsCommits(group.groupKeyStats.contributorStats)}
  />
</Grid>
{/* Other elements ... */}
```

Listing 3.3: Snippet: Pie Chart implementation in Odin.

For my example extension, it is not actually relevant to use HighCharts, as the counter simply shows a text string. So for that example, all that will need to be done is add a new Grid element with a text element inside, getting the value via the `group.groupKeyStats.sundayCommits` object. In the case where I'd need a graph for my extension, it's worth noting that the function calculating the graph's options in Listing 3.3 cannot be reused, as options such as `title` have been hard-coded to "Commits % Member" .

Overall, implementing the example extension into the code-base should not require too much effort. The component based structure keeps the modified files to a minimum, only requiring the editing of two files. Adding the UI element is also fairly easy, only requiring adding a few lines of code to implement the Highcharts component. However, as mentioned above, graph options are not reusable, meaning adding new graph elements comes with some extra overhead, since new options must be defined. Changes are still encapsulated within the same file, so the overhead is relatively small.

After having looked at the source code, it should be noted that there is some extra complications involved when the new metrics are to be tied to a specific user. Data for an individual student's contributions is stored as the `contributorStats` value. This contains merged data from different APIs, and object's structure is hard-coded in various steps of the data aggregation. This especially visible in the `mergeContributorStats` function which is shown in Listing 3.4, which merges two dictionaries of stats (one containing commit numbers, and one containing issues

and merge-requests).

It appears that implementing extension to the system's metric dashboard is generally easy. The separation of concerns imposed by React and Next.js provides *some* extensibility, albeit a limited amount: frontend extension code is encapsulated within component files, although no explicit extension points are present. Additionally, the general code structure of the project means only a few lines would have to be modified in existing code. However, support for *proper* extensibility is not ideally supported as no mechanism concretely separates an extension from a normal code change, meaning the system is more-so *modifiable*, than extensible. This is especially apparent in the case of adding a new contribution metric, as described above.

In addition, when regarding the requirement of strong separation voiced by The Customer, one would not consider this system "extensible". Even with such a simple extension, source code is directly edited in two system files. Furthermore, the system does not whatsoever support late binding of extensions.

3.5 Designing an extensibility mechanism

To make the implementation of new metrics easier, and to solve the additional requirements put forth by The Customer, I will be implementing a plugin manager in the Odin System. As seen in the previous chapter, a well realized plugin architecture should nicely fit all requirements voiced by The Customer – plugins can be developed independently from the source system, and they can be loaded into the system at startup (or later). Furthermore, The Customer has personal experience developing plugins within the Eclipse ecosystem, and voiced that a solution similar to it would be preferable.

For my implementation to be considered successful, it must *at the least* be possible for plugins to hook into the group metric dashboard, providing new visualisations and perform the underlying data tasks. To validate the correctness of my solution, it should be possible to develop the example metric from the previous section. However, as this example is fairly trivial, implementing it will not fully prove that the plugin system supports actual metrics. Therefore, to further prove that this system could support more advanced metrics, the metrics currently present on the group dashboard will be extracted, and reintroduced into the system as "default" plugins. If the system supports building plugins of their scale, then it should hopefully also support other metric requirements needed in the future.

The following subsections will go over design considerations that had to be addressed when implementing a plugin architecture for the Odin system.

```
const mergeContributorDicts = (dictEmail, dictUserName) => {
  const arrayEmail = Object.values(dictEmail)
  const arrayUserName = Object.values(dictUserName)
  const contributorStats = {}
  arrayEmail.forEach((user) => {
    const userNameIndex = arrayUserName.findIndex(
      (userUserName) => userUserName.name === user.name
    )
    if (userNameIndex >= 0) {
      const userNameStats = arrayUserName[userNameIndex]
      contributorStats[userNameStats.userName] = {
        userName: userNameStats.userName,
        name: user.name,
        commits: user.commits,
        lines: user.lines,
        additions: user.additions,
        deletions: user.deletions,
        mergeRequests: userNameStats.mergeRequests,
        issues: userNameStats.issues,
      }
      arrayUserName.splice(userNameIndex, 1)
    } else {
      contributorStats[user.name] = {
        userName: undefined,
        name: user.name,
        commits: user.commits,
        lines: user.lines,
        additions: user.additions,
        deletions: user.deletions,
        mergeRequests: [],
        issues: [],
      }
    }
  })
  arrayUserName.forEach((user) => {
    contributorStats[user.userName] = {
      userName: user.userName,
      name: user.name,
      commits: 0,
      lines: 0,
      additions: 0,
      deletions: 0,
      mergeRequests: user.mergeRequests,
      issues: user.issues,
    }
  })
  return contributorStats
}
```

Listing 3.4: Snippet: Odin backend combining contribution stats

3.5.1 Plugin Discovery

To support plugins, there needs to be some logic for finding and loading plugins into the system. The “easiest” way this can be done is to explicitly import each plugin in some core script, and register them in the plugin manager. Of course, this solution is not very dynamic, and does not satisfy the customers requirement of late binding. Therefore, logic must be implemented to automatically locate available plugins, without needing to change internal files.

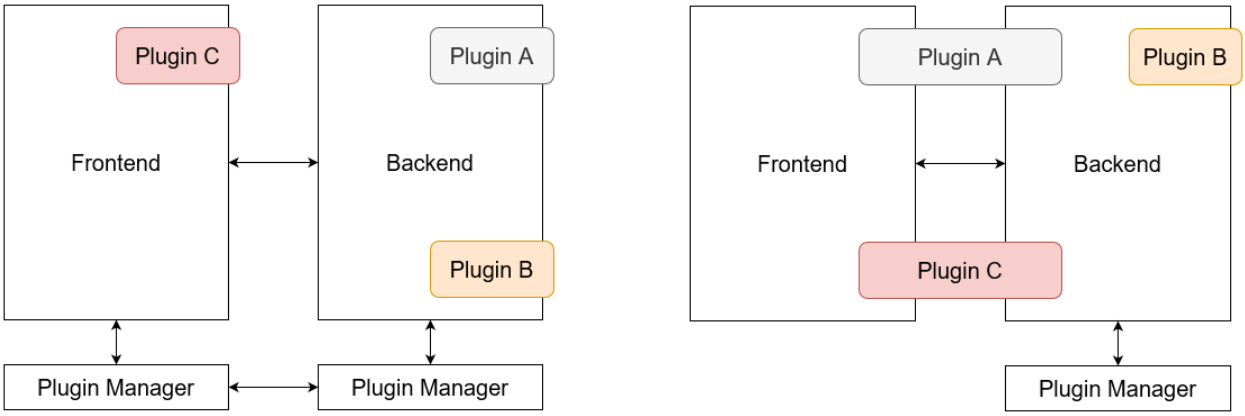
One way to do this, could be to have some *external* JavaScript or configuration file which imports each plugin and exports a collection containing all of them. In this case, this file would need to be updated for each added plugin, but code changes are still happening outside of the core system. Another option is to have the system automatically search through some predefined plugin folder, looking for some expected file – this is how it is done in the three systems discussed in Section 2.3: The system searches through the systems extension folder, and registers plugins as it locates their manifest files.

For my solution, I’ll use an automatic lookup option, as this creates the least cumbersome installation flow for plugin developers: Add plugin files to a specific folder, and the system handles the rest.

3.5.2 Frontend and Backend seperation

The current system is comprised of a frontend-backend server architecture. Unlike the systems discussed in section 2.3 – where the software is running as a single instance on the end-user’s machine – this architecture means there are at the least two different contexts one must take into consideration: The backend running on the server machine; and the frontend running in the end-users web browser. Plugins for data fetching and aggregation needs to plug into the backend, so there must at least be a plugin manager for the backend parts of the system. But, since metrics need to be visualized, there needs to be a way for plugins to extend the front-end of the application. Realistically, we have two possible options for how such a manager can be implemented, as shown in Figure 3.6

Figure 3.6a shows the first of these options. Here, there exists one main plugin manager on the backend running on the server, whilst a secondary plugin manager is instantiated for the frontend whenever an end-user connects to the system. The two plugin managers should need functionality to communicate events and potential dependencies to each-other, but they only manage plugins meant for their respective end – The frontend manager only cares about Plugin C. The benefits of this solution is that it allows some clear distinction between the different plugins and their purpose, and allows plugin developers to only focus on a single context. This also allows some of the work for loading and running the plugins to be offloaded to the frontend de-



(a) Manager on both frontend and backend

(b) Manager only on backend

Figure 3.6: Two different options for a web-server plugin system

vices. A negative of this approach is that plugins that needs to use both the front- and backend are forced to be split into two plugins (or the main manager must have logic to split a plugin automatically). Additionally, offloading work to the end-user's devices could potentially reduce the application's perceived speed, reducing usability. Lastly, implementing logic for communication between the two (or more) managers could potentially require extra implementation effort, or risk being error-prone.

The second option is shown in Figure 3.6b. Here, there is only a single plugin manager, located on the backend. In this solution, plugins that run on the backend are allowed to inject HTML elements into the frontend. This will require that application supports rendering its pages server-side, which luckily can be done easily in Next.js. Keeping the server responsible for performing loading and activation work should also keep sites load-times low, thus less negatively affect user experience. The main benefit of this solution is that you avoid having to develop communication between the two managers, and only needing to worry about the context of a single manager, which should make the logic easier and reduce implementation work.

Since most of the logic involved in writing metrics plugins will focus on backend work (fetching and aggregating external data) and a minimal amount of frontend logic (data visualization is likely to be a single injected HTML element), the *latter option was chosen* as the design for this system. This is also personally beneficial, as it should require less developmental work for me.

3.5.3 Plugin Activation

Plugins can either activate immediately when they are found during startup – such as with Firefox extensions' background scripts – or they can stay idle until a certain part of the system

is used – such as with VS Code extensions' Activation Events. The first approach should make system implementation easier, as you simply assume all plugins are up and running from the get-go. With idle plugins, a system for dynamic activation needs to be implemented. However, if the amount of plugins becomes large, the server's startup time can increase significantly and the applications memory usage can increase, as all plugins are kept loaded.

Since Odin' plugin manager runs only on the backend, a delay to the servers startup-time should not affect end-users. When users are connecting and loading the frontend, the server has already finished loading the plugins. Rather, dynamically activating plugins could potentially *slow down* the server response time during user requests, lessening the user experience. Activating plugins during startup would only be noticeable whenever the server needs to be restarted, which happens less frequently: only when plugins are added, or the core system updates.

Having all plugins active at the same time does increase the systems memory usage somewhat, but as the backend system is running on a dedicated server, I'd argue that this memory usage is negligible. Additionally, by having plugins simply activate whenever they are used, each plugin is likely to have been loaded after a couple of hours regardless, as multiple end-users connect to different parts of the system. The server itself keeps running, meaning that plugins stay active after their first activation. Unless a subroutine for dynamically *deactivating* plugins is implemented, one could consider the memory usage to be equivalent in both cases.

For my implementation, I have decided not to implement dynamic plugin activation, as I do not consider it's benefits for a backend manager to outweigh the additional work that must be done to support it.

3.5.4 Plugin Manifest

As seen with the examples in Section 2.3, using some manifest file is a common way for plugins to declare themselves to the system. When the system performs plugin discovery, the manifest file can be used as the target file to locate. The manifest file is usually considerably smaller than the rest of the plugin's code, allowing some rudimentary verification steps to be done without loading the actual plugin: making sure required fields are present and dependencies to other plugins can be resolved. This does not verify that the plugin works correctly internally, but it at the least ensures some general structure has been followed. Alternatively, the plugin-manager can look for an expected "main" script file, and simply load it in immediately.

Making use of a manifest file in the plugin manager would require some extra development overhead, as one must decide necessary fields, and implement logic to parse the manifest file. This overhead might be worth it regardless, since it should make it significantly easier to implement dynamic activation at some later time, in addition to allowing pre-load verification.

Being an internal NTNU system, I'd argue that the plugin ecosystem will be considerably less complex than that of the public examples looked at. Developed plugins are not published on some external marketplace, but simply added to the server after their development. Apart from some potential utility plugins, or a visualization plugin depending on a data plugin, I expect there to be few plugin-to-plugin dependencies. But, letting plugins to declare their dependants allows the system to give early warnings if a required plugin is missing.

When it comes to plugin versioning, the fact that the Odin system's use-case is fairly specific, and since plugin development is happening in a controlled environment (with all developments likely known to NTNU staff), I see it as unlikely that plugin dependencies will require a strict versioning system, meaning that hosting multiple plugin versions is unnecessary. The uploaded plugin can be considered the *only and correct* version of that plugin.

Lastly, a decision must be made for what file format the manifest file should be written in. Of course, almost any file format could be considered viable, but in the three examples shown in Section 2.3, we've seen the use of three different file-types: Firefox and VS Code both write their manifests as JSON files; while Eclipse plugins uses an XML and a MF file. JavaScript has built in support for reading JSON files, allowing them to be used in code equivalently to a normal JavaScript object. Furthermore, JSON is written in human-readable format, which should make it easier to get an overview of a plugin's configuration when reading the manifests.

For my plugin system, the manifest will be a JSON file, with the following fields:

- **name:** The name can both act as an identifier to avoid multiple instances of the same plugin, and can be used while logging and debugging errors.
- **main:** Name of the main JavaScript file, where events will be called.
- **contributes:** A map of all extension points the plugin wants to attribute to.
- **dependencies:** A list of other plugins this depends on.

One field not included, but could be considered, is allowing plugins to declare *their own* extension points in their manifest. After having looked at Eclipse's handling of plugins, I think that this feature could be powerful and worth supporting. However, developing a system to validate an unknown amount of extension points could prove difficult. Therefore, I do not consider this a feature a *must-have* to deem my solution successful.

3.5.5 Plugin Security

Since plugins can potentially be developed by some external party, one should consider the aspect of security to prevent plugins from running malicious code. In all the examples looked

at in Section 2.3, this is done by restricting the plugins' access to internal data and logic via the use of APIs, and rules for what a given extension point allows a plugin to do. You especially see a focus on security in Firefox's system, where certain APIs are completely inaccessible without explicit access being requested in the manifest.

This security most likely stem from the fact that these systems are introducing plugins developed by completely unknown third parties, but I would argue that the same amount of security is not needed for the Odin system. Although plugins for Odin *could* be developed by someone external to NTNU, they would likely be explicitly hired by NTNU, and NTNU staff would still be the ones uploading the plugin to the Odin server. Therefore, the likelihood that active plugin code is purposefully malicious is unlikely. However, exposing internals via an API would still be beneficial to the plugin developers, as it keeps the plugins more loosely coupled to the system, while still allowing access to internal logic.

Although a proper extension API provides benefits to both security and plugin development, I have deemed it to not be a *must have* for this plugin system. The benefits to security are not particularly important, and as only metric plugins is within the scope for this assignment, the amount of internals needing to be exposed is limited.

3.5.6 Extending the metric dashboard

As mentioned, the plugins will have the option to inject frontend elements during run-time, specifically for adding new data visualizations to the metric dashboard.

Whilst this is necessarily only tangentially related to the core design of the plugin system itself, visual clutter could become an issue when new a seemingly unknown amount of new frontend elements is added. As an example, the metrics needed when course staff wants a general overview of group's progress is different than those needed when analyzing if the students' workload is skewed. This implies that the plugins need some way to define what context they are shown in, and logic should be implemented – likely in the core system – to allow switching what is shown on the fly.

Implementing a fully fledged flexible system for activation, layout, and configuration of these dashboard plugins is outside of the scope of this thesis, but I'd argue that the use of a manifest file allows later iterations of the system to add new attribute keys for these extension points. Some easy example of such configurations could be implemented as a proof of concept, but is not considered a must-have for this delivery.

4 | Plugin System Implementation

In this chapter I will go over the specific details of the plugin system that was implemented. To showcase how an actual plugin is implemented and loaded into this new system, I also implemented the Sunday Commits plugin designed in the previous section. The plugin system has been developed as a fork of the original Odin project, and the source code for the project can be found at the NTNU Gitlab (Hunderi 2021). An important thing to note for this fork is that due to issues with the setup of the development environment, some source code outside the scope of this thesis had to be changed, namely: adding/editing some missing/outdated fields in the mock Blackboard API; adding a mock service worker to fake authentication calls; and adding a button to the start-page to skip some internal authentication and setup-steps. These changes should be reverted before the fork is merged back into the original project.

4.1 Building on Plug-And-Play

NodeJS allows developers to introduce external node-modules into their projects, and there currently exists a large ecosystem of available code to be reused. Before implementing a new solution from scratch, I looked for available packages that could provide some – or all – of the functionality needed in my system.

Whilst I could not find a module that could cover *all* my needs, I decided to integrate the Plug-And-Play (Adaltas n.d.) module as the core logic for my plugin manager. The module itself is simple, but it provides some useful logic for registering plugins and defining extension points. An additional feature it provides – which had not been considered during design – is giving plugins a degree of control over their respective execution order. This point will be explained in more detail below. For Plug-And-play, a “plugin" is simply defined as an object with a name, and a map of various “hook" objects. The hook simply tells the manager when to invoke the plugin’s defined handler function. An example of a simple plugin is shown in Listing 4.1. Here, `MyPlugin`’s hook handler will be invoked by Plug-And-Play when the system later calls on the `foo:bar` hook. Additionally, the `after:"myOtherPlugin"` option ensures that `MyPlugin`’s handler is resolved sometime after `myOtherPlugin`’s handler for that same hook. The module is still fairly simple, and does not provide functionality for automatic plugin discovery, frontend injection, dependencies, or manifest registration, which had to be implemented manually.

```
pluginManager.register({
  name: "myPlugin"
  hooks: {
    "foo:bar": {
      after: "myOtherPlugin"
      handler: (args) => {
        console.log("foo:bar happened!")
      }
    }
  }
})
```

Listing 4.1: An example of a Plug-And-Play plugin.

4.2 Manager initialization & Plugin Discovery

The plugin manager initializes during the Odin server's start-up routine, so there is only ever one instance of the Plugin Manager during the system's runtime. The initialization logic of the manager itself is purely internal Plug-And-Play logic, and therefore not relevant to show here. After initialization, the server runs a script which automatically searches the server's `plugin/` folder and locates sub-folders containing a `plugin.json` manifest file. Listing 4.2 shows the logic for locating and registering plugins. After a plugin is discovered, the `buildPluginObject(...)` method analyzes the manifest and generates a Plug-And-Play compliant plugin-object based on the keys present in the manifest. After all discovered plugins have been registered, the manager instance is made globally available (within the backend environment, see Listing 4.3), so the backend system can use the instance to call upon the plugins during run-time.

The implementation checking dependencies is fairly simple. After registration, a reference to each plugin is stored in the `registeredPlugins` object, with the name mapping to the object. When a plugin declares a dependency, the system checks if an entry for the named plugin is already present (in which case the dependency is already satisfied), if not, the dependency name is added as a key, but maps to a `false` boolean value. This boolean will be overwritten if the named plugin is then later registered. After all plugins are registered, the system checks if any `false` booleans are present in the map, and issues a warning that some dependency is missing.

Unique names are also enforced fairly simply: Before registering, the system checks if the `registeredPlugins` object has a "truthy"[1] entry for the given name, issuing a warning and

[1] In JavaScript, a value is "truthy" if JavaScript's built-in type coercion converts it to a `true` boolean, with values like `0`, `false`, `undefined` being some typical *non-true* values

pluginManager/index.js

```
// ... constants and imports

const manager = plugAndPlay()

const registeredPlugins = {}

glob.sync(extensionsDir + "*/plugin.json").forEach((manifestPath) => {
  // ... Values extracted from manifest

  if (registeredPlugins[pluginManifest.name]) {
    console.warn(
      `!! [PluginManager] Plugin "\${pluginManifest.name}" already registered.
      ↪  Skipping`
    )
    return
  }

  const plugin = buildPluginObject(extDirLocal, extDirAbsolute, pluginManifest)
  manager.register(plugin)

  registeredPlugins[pluginManifest.name] = plugin
  // Add a "false" plugin for unmet depedencies
  pluginManifest.dependencies?.forEach((depName) => {
    registeredPlugins[depName] = registeredPlugins[depName] || false
  })
})

Object.entries(registeredPlugins).forEach(([name, plugin]) => {
  if (!plugin) {
    console.warn(
      `!! [PluginManager] Dependency plugin "\${name}" was not installed`
    )
  }
})
```

Listing 4.2: Snippet: Plugin system discovering and registering plugins

next.config.js

```
const pluginManager = require("./pluginManager")

module.exports = {
  // ... other Nextjs settings
  serverRuntimeConfig: {
    pluginManager,
  },
}
```

Listing 4.3: Snippet: Plugin Manager being made globally available

skipping the registration if a truthy entry is found, as it implies that the given plugin has already been registered.

4.3 Declaring extension points in the application

Declaring some arbitrary extension point in the application is done by using the Plug-And-Play manager's built-in `call(...)` method, providing the `hookName`, `args`, and `defaultHandler` parameters. This method will run the respective hook handler in all registered plugins, passing along the `args` object. The `defaultHandler` parameter can be used to define a default handler that runs after all plugins have resolved, though this option has not been used in any of my own declarations. To support the Sunday Commits plugin, two extension points were created: one for injecting frontend elements into the group dashboard, and one to append new aggregated data fields to the Gitlab data payload on the backend. The simplest of these two extension points is the one for data aggregation, shown in Listing 4.4. Here, the `keyStats` object is the original aggregated data payload, which is then passed on to the contributing plugins as an argument. Since JavaScript objects are mutable, the plugin's handler simply generates the new aggregation from the provided data, and then attaches it to the provided `keyStats` object with some unique key.

The other extension point enables plugins to direct the system to a frontend component in wishes to inject, as shown in Listing 4.5. Notably, instead of simply passing a set of data, the system sets up an internal loader function which is passed to the plugins. This way, the plugin does not need to know the exact logic behind how the system loads the plugin component, and just needs to worry about providing the given loader loader with the required values required fields. Here, the loader's functionality is simple, adding the components *local* path (local in terms of the extension folder) and view-group to an internal array. The specifics surrounding component loading is discussed later in section 4.4.

utils/gitlab/groupStats.js

```
const getGroupKeyStats = async (/* parameters */) => {
  // ... Data fetching and existing aggregations

  const keyStats = {
    // ...
  }

  // Get aggregations from plugins
  await getPluginManager().call({
    name: "aggregation:group:keystats",
    args: { data: keyStats },
  })

  return keyStats
}
```

Listing 4.4: Snippet: Extension Point for adding new data aggregations

pages/courses/[term]/[courseId]/groups/[groupId]/index.js

```
export const getServerSideProps = async (context) => {
  // ... other serverside props
  const extDashboardComponents = []

  await getPluginManager().call({
    name: "component:group:graphs",
    args: {
      loader: ({ path, viewGroup }) => {
        extDashboardComponents.push({ path, viewGroup })
      },
    },
  })

  return {
    props: {
      // ... other props
      extDashboardComponents,
    },
  }
}
```

Listing 4.5: Snippet: Extension Point for loading frontend elements

Although not used for the Sunday Commits plugin, an extension point for *fetching* data has also been introduced, since this extension point is likely to be relevant for future use-cases. It also showcases a more involved case of exposed internal logic. Listing 4.6 shows this extension point. As with the data aggregation extension point, plugins are given a data holder object where plugins are free to add new fields to as necessary. The important internal logic to note here is the fetcher method that is also passed to the plugins. This allows us to hide the exact fetching details from plugins, imposing some standardization on fetching logic and lessening the amount of arguments needed to be sent to the plugins (For example, here we are making each request use the `cachedFetch` helper method, enforcing the internally defined Gitlab base-URL, and adding the PAT header). While not a major focus, it also provides some added security since the Gitlab PAT can be fully hidden from the plugins. The internal logic for plugins using this hook would effectively consist of calculating which endpoint to use – perhaps based on some previously fetched data – and provide the loader with the endpoint URL, parse the received payload as necessary, and add it to the provided data object.

Whilst it is not enforced by the plugin manager, I decided to keep a specific structure when naming the extension point hooks: `<category>:<subcategory/context>:<identifier>`. The category signifies what type of extension is expected by connecting plugins – the `component` category implies a frontend element will be injected – while the latter two provide a descriptor for *where* in the application the extension happens – `group:graphs` shows its for components injected to the group-dashboard graphs. While being descriptive, it also allows us to impose a structure the arguments provided to plugins: We can, as a rule, say that plugins using `component` category hooks are *always* provided with a loader method.

4.4 Supporting frontend components

In the current solution, a plugin injects it's frontend elements by providing a path to some `.jsx` file which defines the React Component that should be injected (Listing 4.5). Internally, the array of these component paths is given to the system component for the metric dashboard. For each listed path, a component named `DynamicComponent` is loaded, as seen in Listing 4.7. Within `DynamicComponent` – shown in listing 4.8 – the component for the given path is loaded dynamically – with element properties passed down to the loaded component. Note that the implementation passes a local instance of the `Highcharts` (the library used to show charts), allowing plugin developers to implement the same Highcharts settings in their plugins, if needed.

While this solution has the benefit of allowing plugin developers to create new components using JSX syntax – the arguably "correct" approach to React development – it was unfortunately later discovered that the solution was not optimal in terms of late plugin binding. Due to how Next.js optimizes frontend code during the build phase, the code for these frontend elements *must* be

utils/gitlab/groupStats.js

```
const getGroupKeyStats = async (/* parameters */) => {
  //... Init values and other data calls

  const fetcher = async ({ endpointUrl }) => {
    const fetchUrl = `${path}/${endpointUrl}`
    console.log("[PluginManager] Plugin doing call to:", fetchUrl)
    const response = await cachedFetch(fetchUrl, {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
        "PRIVATE-TOKEN": pat,
      },
    })
    return response.json
  }

  await getPluginManager().call({
    name: "dataSource:group:restGitData",
    args: { data: groupStats, fetcher },
  })

  // ... Data Aggregations
}
```

Listing 4.6: Snippet: Extension point for Gitlab data fetching

components/Stats/GroupStats/GroupStatsGraphs.jsx

```
<>
  {/* ... Built-in metric elements */}
  {extDashboardComponents
    .filter(({ viewGroup }) => viewGroups.includes(viewGroup))
    .map(({ path }) => (
      <DynamicComponent
        key={path}
        path={path}
        extData={group.groupKeyStats}
        highcharts={Highcharts}
      />
    ))}
</>
```

Listing 4.7: Snippet: Plugin frontend elements injected on group dashboard

components/Plugins/DynamicComponent.jsx

```
import dynamic from "next/dynamic"

const getPluginComponent = (c) =>
  dynamic(
    () =>
      import(
        /* webpackInclude: /.*\.jsx$/ */
        /* webpackChunkName: "plugin-component" */
        `../../plugins/${c}`
      ),
    { ssr: true }
  )

export default function DynamicComponent(args) {
  const PluginComponent = getPluginComponent(args.path)

  return <PluginComponent {...args} />
}
```

Listing 4.8: Dynamic loader for component

present at build time. This issue will be covered more in-depth in section 4.9. Having limited experience with Webpack, I was unable to solve the issue for this implementation. Additionally, it seems that Next.js enforces the use of Webpack, and since one requirement was that the solution keeps the same framework as the existing system, looking into alternative build tools was also not a viable option.

4.5 Plugin Manifest Structure

The plugin manifest is declared using a file named `plugin.json`, and the root structure of the manifest is the same as the one designed earlier in Section 3.5.4, the “dependencies" key being optional. Table 4.1 shows each key and explains which data should be provided to them.

Of the root keys, the `contributes` key is naturally the most important and complex one, as it defines which extension points the plugin contributes to. Inspired by the VS Code manifest structure, plugins declare their contributions with a map where category keys points to arrays of category specific declaration objects. When parsing the manifest, the system can then expect certain fields to be present depending on the category that is declared and do category-specific

Manifest Key	Datatype	Details
`name`	`string`	Identifier for the plugin. System issues a warning if name is not unique.
`main`	`string`	*Local* path to the plugin's main script file (e.g. `./main` for a file located in the root).
`contributes`	`Object`	Object with category-to-object mappings, declaring which extension points the plugin contributes to. See Table 4.2 for keys used in each category.
`dependencies`	`[string]`	**(Optional)** Array of plugin names that the plugin depends on. System issues a warning after startup if any dependency is missing.

Table 4.1: Root keys for the plugin manifest file (`plugin.json`)

setup logic. The expected keys for each category is listed and detailed in Table 4.2[2].

4.6 PluginBuilder

The PluginBuilder script is called using `buildPluginObject(...)` during discovery/registration (Listing 4.2), and parses the `contributes` declarations in the plugin's manifest, generating a single Plug-And-Play compliant plugin-object (Listing 4.1) which is then registered to the plugin manager.

As each category of extension points can expect a certain behaviour from their respective plugins – the `aggregation` points expects a new data field to be added by all contributing plugins – the PluginBuilder script provides a set of category-specific builders for each category, where it either fully defines the plugin's hook for the given extension point, or adds some enforced logic before or after invoking the plugin's own handler. This makes it so plugin developers only need to worry about the explicit task their plugin should perform: point to a component, or write an data-method that returns a value. Listing 4.9 shows the implementation of the `buildPluginObject(...)` method.

For each contribution-category, a category-specific handler is built using its respective hook-builder. The builder for the `component` category is seen in Listing 4.10. Notably, this builder creates the entire hook handler, as all that needs to be done is provide the loader method with the component path from the manifest. The `aggregation` builder is shown in Listing 4.11.

[2] An even more in-depth explanation can be found in the project repository

Key	Datatype	Details
General (applies for all contributions)		
`before`	`[string]`	(Optional) List of plugin-names. This plugin will resolve it's contribution *before* the listed plugins
`after`	`[string]`	(Optional) List of plugin-names. This plugin will resolve it's contribution *after* the listed plugins
`component`		
`hook`	`string`	Target extension point hook. Currently supports: – `group:graphs`
`path`	`string`	*Local* path a `.jsx` component file to be injected. Component receives the following props: – `extData` The metric data payload – `highcharts` The system's Highcharts instance
`viewGroup`	`string`	(Optional) An arbitrary group name. Ties the element to that group, allowing the user to filter what is shown.
`aggregation`		
`hook`	`string`	Target extension point hook. Currently supports: – `group:keystats`
`handler`	`string`	Name of the aggregation function *in the main file*. Function should return a set of data. Arguments: – `data` The existing data payload
`dataKey`	`string`	The data will be stored within the data payload using this key
`dataSource`		
`hook`	`string`	Target extension point hook. Currently supports: – `group:restGitData`
`handler`	`string`	Name of the fetching method *in the main file*. Function should return a set of data Arguments: – `data`: The existing data payload – `fetcher`: Interal helper method for cached web-calls
`dataKey`	`string`	The data will be stored within the data payload using this key

Table 4.2: Available keys for each extension point

Here, the builder ensures that the value returned from the plugin's own handler is added to the provided data object. Also, as an example of some basic security, the builder logs a warning if the plugin overrides an already existing data key.

pluginManager/pluginBuilder.js

```
const hookBuilderMap = {
  component: buildComponentHooks,
  aggregation: buildAggregationHooks,
  dataSource: buildDataSourceHooks,
}
function buildPluginObject(extDirLocal, extDirAbsolute, manifest) {
  let hooksObject = {}
  Object.entries(manifest.contributes).forEach(([key, contributions]) => {
    const mainFile = require(path.join(extDirAbsolute, manifest.main))

    if (!hookBuilderMap[key]) {
      console.warn(/* Warning about unknown extension point */)
    } else {
      hookBuilderMap[key]({ mainFile, hooksObject, manifest, contributions,
      ↪  extDirLocal, extDirAbsolute })
    }
  })
  return { name: manifest.name, hooks: hooksObject }
}
```

Listing 4.9: Snippet: The `buildPluginObject()` method

4.7 The Sunday Commits plugin

With the plugin system in place, the Sunday Commits plugin was implemented. The plugin contributes to two extension points: `aggregation`, where the amount of Sunday-commits will be counted; and `component`, where a simple text message component will be injected. Additionally, to showcase the "viewGroup" option, this frontend element is set to a view-group named "fun". The plugin itself consists of three files: the `SundayCommits.jsx` component, the `main.js` script, and the `plugin.json` manifest file. The manifest file is shown in Listing 4.12. Note that even if this example puts the SundayCommits component within a `components/` folder, the component could have been located in the plugin's root folder instead (or any other sub-folder).

The two source files are shown in Listing 4.13 and 4.14. Using the commits array available in the core data object, the aggregator handler simply filters the commits based on commit date, and returns the length of the filtered object.

pluginManager/pluginBuilder.js

```
function buildComponentHooks({ hooksObject, extDirLocal, contributions }) {
  contributions.forEach((contribution) => {
    const { hook, component, viewGroup } = contribution

    if (!component) {
      console.warn(/* warning about missing component */)
      return
    }

    const relativeComponentUrl = path.join(extDirLocal, component)

    const pluginHandler = ({ loader }) => {
      loader({
        path: relativeComponentUrl,
        viewGroup: viewGroup || "Default",
      })
    }

    hooksObject[`component:${hook}`] = genHook(pluginHandler, contribution)
  })
}
```

Listing 4.10: Snippet: Builder for the `component` category

pluginManager/pluginBuilder.js

```
function buildAggregationHooks({ hooksObject, mainFile, contributions }) {
  contributions.forEach((contribution) => {
    const { hook, handler, dataKey } = contribution

    const pluginHandler = ({ data }) => {
      if (data[dataKey]) {
        console.warn(/* warning that data is overwritten */)
      }

      // Call handler to calculate new value
      data[dataKey] = mainFile[handler]({ data })
    }

    hooksObject[`aggregation:${hook}`] = genHook(pluginHandler, contribution)
  })
}
```

Listing 4.11: Snippet: Builder for the `aggregation` category

plugins/sundayCommits/plugin.json

```
{
  "name": "Sunday Commits",
  "main": "./main",
  "contributes": {
    "component": [
      {
        "hook": "group:graphs",
        "viewGroup": "fun",
        "component": "./components/SundayCommits"
      }
    ],
    "aggregation": [
      {
        "hook": "group:keystats",
        "handler": "sundayCommitsHandler",
        "dataKey": "sundayCommits"
      }
    ]
  }
}
```

Listing 4.12: Manifest for the Sunday Commits plugin

plugins/sundayCommits/main.js

```
module.exports = {
  sundayCommitsHandler({ data }) {
    console.log("Adding new data aggregation...")

    // getDay returns "sunday" as 0
    return data.commits.filter((commit) => {
      const commitDate = new Date(commit.created_at)

      return commitDate.getDay() === 0
    }).length
  },
}
```

Listing 4.13: Aggregator for the Sunday Commits plugin

plugins/sundayCommits/components/SundayCommits.jsx

```
const SundayCommits = ({ extensionData }) => {
  return <h3> {extensionData.sundayCommits} commits where done on a Sunday!
  ↪  </h3>
}

export default SundayCommits
```

Listing 4.14: Component file for the Sunday Commits plugin

There are two things worth noting in these files: The plugin developer here knows/expects that the commits array is part of the provided data object, which requires knowledge of the internal system; and the aggregator function does *not* add the data result to the data object, but returns the value and lets the wrapping handler from the Plugin Builder (Listing 4.11) append it.

In the frontend component, the aggregated value is found attached to the provided data object stored as the “sundayCommits" value. The end result is shown in Figure 4.1. Also notice the selection of the “fun" view-group in the "View Groups" input field.

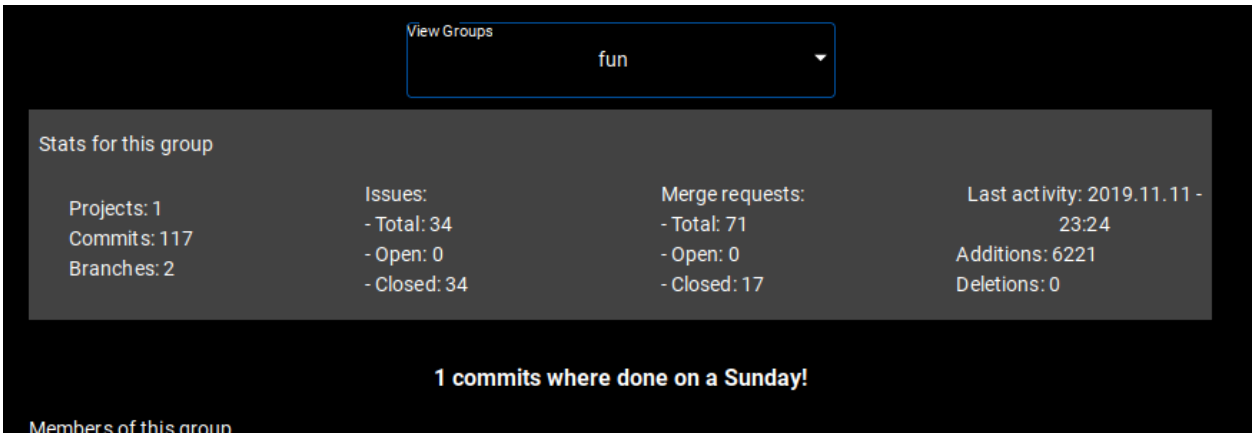


Figure 4.1: The Sunday Commits value visualized on the frontend

4.8 Transforming existing metrics to plugins

The Sunday Commits plugin works as anticipated, but is very basic and does not make use of the data fetching extension point. This example alone arguably cannot, in it’s own right, prove that the solution supports the more advanced plugins future use-cases might require.

The existing metrics used in the system, particularly the various contribution-graph elements, showcases metric functionality that’s more advanced than the Sunday Commits plugin (larger aggregated data-sets and visualizations using Highcharts). For the implemented plugin system to considered a valid solution, the plugin system should at least be able to support plugins as advanced as these existing metrics. To showcase that the implemented system supports such plugins, select parts of the existing metric system was extracted and re-implemented as standalone plugins. The parts extracted were:

- The Gitlab API call that fetches the groups full collection of commits.
- The aggregation method that calculates each members’ contributions.
- Each frontend Graph element, which visualize the aggregated member contributions.

In the end, the plugins could be implemented more or less without issue. Some minor changes had to be done to the data source and aggregation extension points, as I needed to provide a few additional system values to the plugins (a fileblame boolean, since/until date limits for fetching data sources, and a holder-object for some default data needed to calculate contributor stats, but that was not to be part of the final payload).

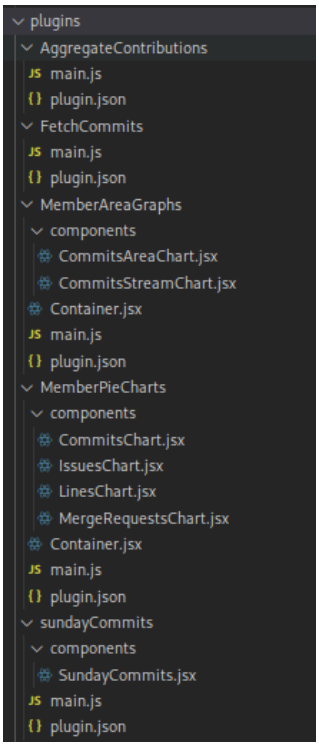


Figure 4.2: Content of the plugins folder

From this, the following four plugins where created (The two rows of charts where made as separate plugins as they only seemed tangentially related):

- **Commits Fetcher**: Gets the commits data from the Gitlab API. Manifest is shown in Listing 4.15. Note that it also provides an aggregation, since this was closely related to the commits data.
- **ContributorStats Aggregator**: Performs the larger aggregation of user contributions. Manifest shown in Listing 4.16. Note that it has a dependency to Commits Fetcher.
- **Member Area Graphs**: Displays a row of two area graphs. Manifest is shown in Listing 4.17. Note the use of the "after" key to place it below the row of pie charts.
- **Member Pie Chart**: Displays a row of four pie charts. Manifest is shown in Listing 4.18.

For the fetching and aggregation logic, most script code was simply pulled straight from the original code and wrapped inside the plugin's handler function (A few sections had mixed in aggregations for other statistics which had to be rewritten in the source code). I will therefore refrain from showing most of these scripts, as they are just direct copies of existing code.

The two visualization plugins contributes a whole row to the `component` extension point. The plugin system only allows the contribution to be a single component, and contributing components are each loaded row-wise below each-other. Because of this, the graph elements that shared a row could not be separated into distinct plugins, but since each row of graphs seemed closely related, it made sense to have each plugin contribute a row element displaying a set graphs, as shown in Listing 4.19. For each of these plugins, charts are created in the same manner, using the HighchartsReact node package to generate it's chart, shown in Listing 4.20.

The most important thing to note here, is that the plugins are importing node packages that are present in the source project. As discussed later in Section 4.9, the plugins shouldn't themselves add node modules, and are thus dependent on the HighchartsReact package being available in the source project. Additionally, as future metrics are likely to use Highcharts for their visualizations, the `utils/commonjs/charts.js` helper file was created. The file contains a set of helper functions that plugins can use to generate Highcharts-compliant configuration objects for their graph elements.

The resulting frontend after implementing these plugins can be seen in Figure 4.3. As one can see, the this frontend is exactly as the one we had back in Figure 3.3[3]. The final contents of the plugins folder is shown in Figure 4.2.

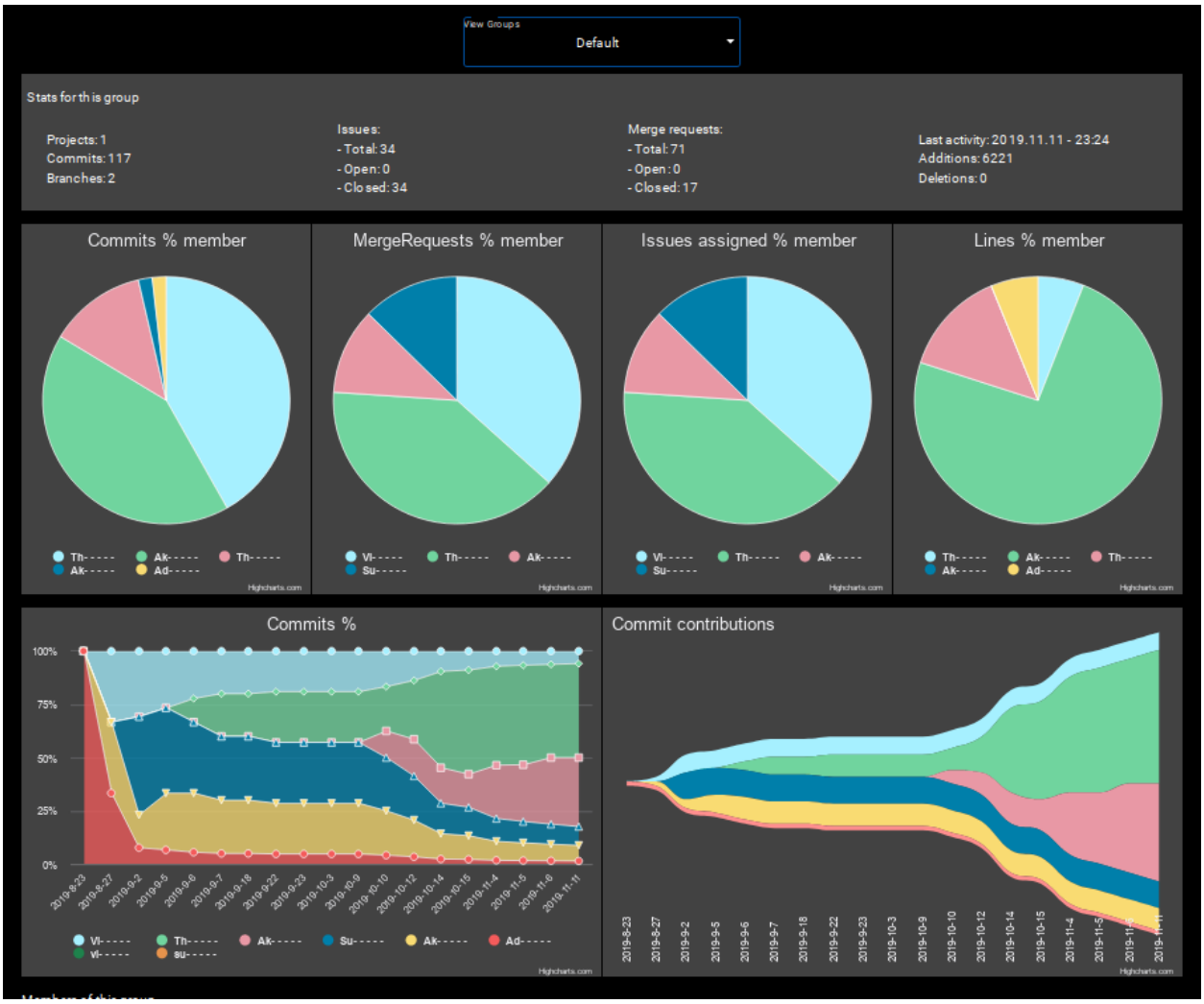


Figure 4.3: Previously Internal Charts extracted as plugins

[3]The screenshot uses a different target group for it's data, but the underlying methods are the same.

plugins/FetchCommits/plugin.json

```
{
  "name": "Commits Fetcher",
  "main": "./main",
  "contributes": {
    "aggregation": [
      {
        "hook": "group:keystats",
        "handler": "commitStatsHandler",
        "dataKey": "commitStats"
      }
    ],
    "dataSource": [
      {
        "hook": "group:restGitData",
        "handler": "commitsFetchHandler",
        "dataKey": "commits"
      }
    ]
  }
}
```

Listing 4.15: Manifest for the Commits Fetcher plugin

plugins/AggregateContributions/plugin.json

```
{
  "name": "ContributorStats Aggregator",
  "main": "./main",
  "contributes": {
    "aggregation": [
      {
        "hook": "group:keystats",
        "handler": "contributorStatsHandler",
        "dataKey": "contributorStats"
      }
    ]
  },
  "dependencies": ["Commits Fetcher"]
}
```

Listing 4.16: Manifest for the ContributorStats Aggregator

plugins/MemberAreaGraphs/plugin.json

```
{
  "name": "Member Area Graphs",
  "main": "./main",
  "contributes": {
    "component": [
      {
        "hook": "group:graphs",
        "after": "Member Pie Charts",
        "component": "./Container"
      }
    ]
  },
  "dependencies": ["ContributorStats Aggregator"]
}
```

Listing 4.17: Manifest for the Member Area Graphs plugin

plugins/MemberPieCharts/plugin.json

```
{
  "name": "Member Pie Charts",
  "main": "./main",
  "contributes": {
    "component": [
      {
        "hook": "group:graphs",
        "component": "./Container"
      }
    ]
  },
  "dependencies": ["ContributorStats Aggregator"]
}
```

Listing 4.18: Manifest for the Member Pie Charts plugin

plugins/MemberPieCharts/Container.jsx

```
import { Grid } from "@material-ui/core"
import CommitsChart from "./components/CommitsChart"
import IssuesChart from "./components/IssuesChart"
import LinesChart from "./components/LinesChart"
import MergeRequestsChart from "./components/MergeRequestsChart"

const Container = (props) => {
  const charts = [CommitsChart, MergeRequestsChart, IssuesChart, LinesChart]
  return (
    <Grid
      {/* ... general props */}
    >
      {charts.map((Chart, i) => (
        <Grid
          {/* ... general props */}
        >
          <Chart key={i} {...props} />
        </Grid>
      ))}
    </Grid>
  )
}

export default Container
```

Listing 4.19: Snippet: The root row element for the Member Pie Charts plugin

plugins/MemberPieCharts/components/CommitsChart.jsx

```jsx
import HighchartsReact from "highcharts-react-official"

import { makePieChartOptions } from "utils/commonjs/charts"

const CommitsChart = ({ extData, highcharts, containerProps }) => {
  const data = Object.entries(extData.contributorStats)
    .filter(([key, value]) => value.commits > 0)
    .map(([key, value]) => ({
      name: value.name,
      y: value.commits,
    }))

  return (
    <HighchartsReact
      options={makePieChartOptions(data, "Commits % member", "Commits")}
      highcharts={highcharts}
      containerProps={containerProps}
    />
  )
}

export default CommitsChart
```

Listing 4.20: Implementation of the "Commits" pie chart used in the Member Pie Charts plugin

4.9 Limitations in the current solution

The current solution has allowed the development of the Sunday Commits plugin, which could be developed without making changes to the systems source code. However, the solution, as is today, does come with some notable limitations to how plugins can be developed, which will be covered in this section.

4.9.1 Plugin Main-file and the NodeJS runtime

Plugins are discovered and loaded during the server's startup phase. At that point in time, the JavaScript is runs purely in the NodeJS runtime. This means, that the JavaScript written for the plugin's main file (and subsequently imported helper files) must be written in a version of JavaScript supported by the server environment's current NodeJS version (the frontend component file is processed by Webpack and therefore not limited by this). As of now, this means that the plugin's main file must be written as CommonJS, as opposed to the more frequently used ES6 syntax. While mostly being an issue of preference, this means `import` and `export` statements must be replaced with `require()` and `module.exports = {...}`. Furthermore, non-CommonJS scripts cannot be imported by the main file. There are also some deeper technological differences, but the aforementioned consequence is the most relevant for plugin developers.

4.9.2 Frontend elements must be built

Implementing support for frontend injections proved more difficult than originally anticipated. This difficulty primarily had to do with the manner in which web development frameworks like Next.js transforms its source code during it's build phase – before production. To optimize server load times, the source code is transformed into a series of bundles using Webpack, transpiling "non-traditional" web files – such as React's `.jsx` files – into traditional files – into CommonJS JavaScript, HTML, and CSS files.

The dynamic import statement used in the `DynamicComponent` component (Listing 4.8) in reality means that the system preemptively builds each file that could possibly be imported with the variable string during the Next.js *build-phase*, which in this case is each file within the `plugins/` directory[4]. However, if a component file for some plugin is not present within the plugins folder during the build-phase, the expected component will not be available during runtime. Plugins

[4] Consequently this also means that the plugin folder location *must be statically stated* in DynamicComponent. If the folder had been set with an environment variable (as shown below), the entire string becomes dynamic, causing a pre-packaging of *every single source file.*

```
import(processs.env.PLUGINS_LOCATION + <component path>)
```

providing frontend elements *must therefore be present during build-time*, rather than startup-time. This means that the current solution has not managed to fully support plugins bound during start-up, which was one of the requirements state by The Customer. However, the *most* important requirement – as stated by The Customer – was that plugins could be developed fully separate from the source code, something which is still possible with the current solution.

To ensure only the minimum amount of required files are pre-packaged by Webpack, `DynamicComponent` makes use of a Webpack option called "Magic Comments" to only allow `.jsx` files to be pre-packaged. This however add a second limitation for component development, as developers *must* write components as `.jsx` files.

4.9.3 Plugins should not be NodeJS sub-projects

Because of the aforementioned issue with dynamic import statements, the current solution struggles if one wants to develop their plugin as a secondary NodeJS project (i.e. adding a `package.json` and `node_modules` folder within the plugin's folder). Since Webpack's prepackages every possible import, Webpack will end up traversing down the plugin's `node_modules` folder, which can potentially slow down the building phase significantly, and cause a large amount of unwanted `.jsx` files to be packaged. Similarly, it is possible that the presence of a local `.git` folder within the `plugins/` folder could cause a similar slowdown as Webpack traverses through it. The most important consequence of this, is that if a plugin need to make use of some new external node package, these packages should for preferably be installed as a dependency in the main project. For example, this is the case for the existing data visualizations elements, which depend on the Highcharts package.

To my understanding, this issue can be somewhat easily resolved: one should be able to exclude these folders from the traversal process using various Webpack configuration options and Magic Comments. Unfortunately I was not able to implement a working solution within the timeframe of this project.

5 | Result and Discussion

5.1 Research Question 1

Research question 1 was: “What characterizes ‘Extensible’ software? What makes some software be regarded as more extensible than others?”

In researching literature surrounding the expression “extensibility”, it was found that a software’s architecture can be considered extensible if the architecture and its mechanisms define clear extension points where new capabilities can be introduced. extensibility can be regarded as a subset of “modifiability”, but one must be mindful that not all mechanisms that support modifiability also supports extensibility: if extension points are not well defined, one would not claim that the system is extensible, as there is not discernible difference between an extension and a modification.

Generally, for a piece of software to be categorized as extensible, these separations between modification and extension must be strongly defined. In the software eco-system, it appears that the software being regarded as more “extensible” than others, are the ones where this separation between extension and modification is the most noticeable. Commonly among these, the application implements a plugin system, as was seen with Firefox, VS Code, and Eclipse. While other approaches might also manage to separate extensions and modifications to a high degree, I’d claim that the plugin systems we’ve seen tend to be more recognized as extensible since they generate their own ecosystems of plugin developers, who all clearly develop extensions to the program, without modifying existing code. Additionally, the end-users of these systems directly interact with the plugin system as they themselves discover and install plugins, which further solidifies these programs as extensible in the zeitgeist of their ecosystems.

5.2 Research Question 2

Research question 2 was: “To what extent does the current system support the extensibility quality goal?”

Looking at the development documentation it was found that while the development had a secondary focus on maintainability and modifiability, this did not result in a strong enough sep-

aration between extension and modification to characterize the system as extensible. The use of React component classes offer a small degree of polymorphism, so it can be claimed that the existing system supports white-box extensibility to a very limited extent, and some extensibility is inherited from the Next.js framework itself. Regardless, the system as is today does not strongly define any discernible extension points in the application, and can therefore not claim to support the extensibility quality goal to any notable extent.

5.3 Research Question 3

Research question 3 was: "How can the current system's architecture be modified to better support extensibility?"

In an effort to improve the extensibility of the existing Odin system, a plugin system was implemented over the course of this project. Implementing the example plugin, Sunday Commits, shows that the implemented solution has managed to support extensibility in the group metrics part of the system. Additionally, as it was possible to extract and re-develop existing metrics as independent plugins, we can conclude that the implemented plugin system is, at least, powerful enough to support metrics of the same complexity as the currently identified requirements.

Generally, implementing the plugin system did not require larger changes to the existing architecture. The plugin manager, which contains most of the functional logic to support plugins is separated from the core system itself, only hooking it's own logic into the backend server's build and startup logic. Introducing extension points within the core system does not require any major architectural changes: the plugin manger instance is freely available to any backend script, and new points are defined by invoking the plugin manager's `call()` function, where any arbitrary new hook can be defined.

5.3.1 Fulfilment of Requirements

In answering Research Question 3, arguing if the implemented system "better supports extensibility" could be considered a question of whether the implemented plugin system can be called a "good" solution. If the system is to be considered good, then it should at least manage to satisfy the requirements put forth by The Customer.

For this project, The Customer stated one primary requirement of the implemented solution: extensibility must be supported for the parts of the system that deals with group metrics. Two additional non-functional requirements were also stated by the customer: plugin code should be strongly separated from the core system code, and plugins should be bound late in the project life cycle.

Requirement 1: Extensibility supported

Following the definition of extensibility proposed earlier (Section 2.1), one can consider a system extensible if new capability can be implemented with little to no changes to existing code, and without impacting existing capabilities.

The implemented solution realizes a plugin system that exposes the Odin system's internal group metric functionality, and provides extensions points for the three main steps of metric logic flow: data fetching, aggregation, and visualization. Furthermore, introducing new extension points into the system is easily done: Ideally, one only needs to invoke the PluginManager's `call()` method within any backend process, naming some arbitrary hook name. If a new contribution category is introduced, one needs to also modify the PluginBuilder accordingly. But if the new extension point makes use of an existing category, no more work needs to be done.

The development of the Sunday Commits plugin proves the plugin system supports the addition of new group metrics, without affecting the existing metrics. Furthermore, as it was possible to extract the existing metrics into standalone plugins, the system can be said to – at the least – support the metric requirements of the original system. Therefore, the project has achieved the main requirement stated by The Customer: *Extensibility is supported for the part of the system that deals with group metrics*. Additionally, since adding new extension points can be done in an arguably easy manner, the solution delivered can also be considered generic enough to support future needs of extensibility.

Requirement 2: Strong separation

The plugin mechanism strongly separates developed plugins from the systems source code, as plugins are automatically discovered and loaded. The internal system implements extension points without any knowledge or requirement of available plugins, and plugins can freely be removed and added as needed, without internal capabilities or other plugins breaking (as long as no stated plugin dependencies are broken). This shows that plugins are very loosely coupled from the core system. Some coupling still exists, as discussed in Section 4.9: As plugins cannot be developed as completely self-contained node-projects, plugins must depend on that their needed node packages are available in the core system – as was the case for the Highcharts frontend elements in the two graph Plugins. Aside from this issue, plugin code itself is fully separate from the core system's code, and it can therefore be claimed that *the implemented solution satisfies the requirement of strong separation*.

Requirement 3: Late binding

Plugins are dynamically discovered and loaded into the system during the server's startup process, aka during initialization time. However, as discussed earlier in Section 4.9, the issues surrounding Webpack and dynamic imports means plugins contributing frontend components must be present at build time. Due to this issue, the following is true: If a plugin only contributes a data source or data aggregation, no build is required, and bindings are happening at *initialization time*. However, if a frontend element is contributed, bindings happen at *build time*. Therefore it can be concluded that *the developed solution only manages to* partially *satisfy the requirement of late binding*.

Overall, most stated requirements have been satisfied. Whilst Requirement 3 is only partially satisfied, The Customer did not considered it as important as the 1 and 2. From this, we can claim that *the solution has improved the Odin system's extensibility.*

5.4 Validity of results

The results produced for this thesis project have succeeded to provide answers for all three research questions that were put forth in chapter 1. Additionally, a plugin manager system was integrated into the system, and has managed to satisfy the most important of The Customer's requirements. While these results have seemingly managed to cover all issues this study aimed to solve, the validity of these results should be discussed. The following subsections will discuss some aspects of this thesis' progression, and how they potentially could affect the validity of the presented results.

5.4.1 Brief literature review

This project was conducted in full during the autumn semester. Unlike many other thesis projects at NTNU, it does not build upon a literary review from the previous semester. Because of this smaller time-frame, the theory presented in Chapter 2 has limited depth and might not fully showcase the deeper theory and characteristics surrounding extensibility.

Furthermore, because of this time-constraint, the project did not find the time to fully research the state-of-the-art for extensible web-applications. When looking for relevant examples, the three applications looked at all showcase a plugin architecture, which could be a consequence of this shallow insight. More thorough research could potentially have found valuable examples that show *other* forms of extensible architectures, which also could have satisfied The Customer's requirements – potentially even better than that of a plugin architecture.

5.4.2 Limited pool of stakeholders

While developing the plugin system, all discussions surrounding the requirements and design of an extensible system were done with a single stakeholder – my supervisor. Being both a course leader and one of the system's developers, my supervisor *does* represent the two stakeholder groups most affected by this implementation, so their opinions are nevertheless valid for this project. However, other teachers and developers are also equal stakeholders, but have not been part of the design and development discussions, meaning the developed solution might not reflect the wants and needs of all stakeholders.

5.4.3 No real life test of solution

The solution has satisfied the main requirements of The Customer, and example plugins have been developed as a proof of concept. However, the solution has yet to be tested by other developers, so it is difficult to definitively conclude that the solution works well in a real-life scenario.

Developing the example plugins has proven that the plugin system allows Odin to be extended with new metrics, and one could therefore consider that from a purely technical standpoint, the resulting implementation is a valid solution. However, without real-life tests there is potential that the system fails to support plugins with other requirements than the currently extracted plugins.

Additionally, it is difficult to conclude if the solution is satisfactory in regards to the developer experience with the system. While I myself managed to use the plugin manager to create plugins with ease, the same might not be said for other developers. Having developed the management system, I am naturally more familiar with the inner workings of the system, and therefore more aware of how the plugin's code and manifest is handled. Without properly testing the solution on unfamiliar developers, it cannot be concluded that the implemented solution is as understandable for other developers.

6 | Conclusion & Further Work

6.1 Conclusion

Conducted over the autumn semester of 2021, this study has provided a brief literature study of the theory and existing work surrounding the “Extensibility” quality goal in software architectures. Additionally, the study has seen the integration of a plugin manager system in the “Odin” web application, improving the extensibility of the application’s metric dashboard.

In the literary study, it was concluded that extensibility can be viewed as a sub-category of the more broader term “modifiability”. As such, there is a strong connection between a system’s ability to support these two. However, a system can only claim to be extensible if one can make clear and concise distinctions between an additive change that introduces a new capability (aka, an extensions), and the more general modifications done during a system’s lifetime. When looking at applications that are generally considered as “extensible software”; systems where new capabilities are easily added without any source code modifications tended to be the most prevalent. Furthermore, the more known of these systems implements a plugin system – as seen in Firefox, Visual Studio Code, and Eclipse. In these systems end-users directly interact with the extensibility mechanism, which could be a reason why these programs are the ones generally considered to be extensible.

To support extensibility in the Odin web-application, a plugin manager system was implemented. Following requirements put forth by The Customer, extensibility had to be supported in the metric dashboard for student groups, as this is the part of the system most likely to see a change in specification. To make the group dashboard extensible, three extension points where introduced into the core Odin system, giving plugins access to the general steps of a metric measurement: data fetching, data aggregations, and data visualizations.

As the system has yet to be tested in a real life scenario, it is difficult to confidently say that the implemented system can support all future functionality the system requires, but as it was possible to both implement a new metric – with the Sunday Commits plugin – and extract the existing metrics into “default” plugins, I will claim that the system at least manages to support extensions of the metric dashboard within the scope of the currently established metric requirements.

6.2 Further work

While the solution provided has managed to support extensibility as required by The Customer, there are some aspects of the plugin system that could be improved. This section will shortly put forth some areas of improvement that could help elevate the power of the plugin system past it's current state.

6.2.1 Properly support late binding

Plugins supplying new frontend elements must be present in the system's plugins folder at build time, due to how source code is bundled with WebPack. While it can be considered only a minor inconvenience, doing a full system rebuild every time a new frontend plugin is installed can cause larger down-times than preferable. Hence, it could be valuable to properly fix Webpacks configuration, and potentially extend the plugin system with some some subroutine that can build plugins during startup, separate plugin builds from the core system build phase, and reducing the potential down-times required. Even better, such a routine could potentially be able to perform during run-time, removing the need for a server restart all-together.

Solving the Webpack issue should also allow plugins to be developed as completely separate NodeJS sub-project, meaning plugins no longer need to depend on their required node modules being present in the core system.

6.2.2 Plugin management features

In the current workflow, a plugin must be directly uploaded to the hosting application server's internal files. However, as one experiments with new plugins, this could be quite cumbersome, as you'd likely swap and reconfigure plugins several times before a final setup is discovered. It could be valuable to implement some user-facing plugin administration page within available on the application frontend, where system developers and select course staff are able to upload, configure and disable plugins as needed.

6.2.3 Test plugin system in a real-life scenario

As discussed earlier in Section 5.4, the implemented plugin system has yet to be tested in a real-life scenario, meaning there is uncertainty in how well the solution will be received by the developers who will *actually* develop plugins. It is especially hard for me alone to judge the development experience for other plugin-developers, as I am inherently more familiar with the system than actual plugin-developers. It would therefore be beneficial to perform one or more real-life tests of the provided solution. Proper testing would also expose technical shortcomings of the solution and allow iterative improvements to resolve these issues.

Bibliography

Adaltas (n.d.), 'Plug-and-play repository'. Available at: https://github.com/adaltas/node-plug-and-play (Accessed 28st September 2021).

Adams, K. M. (2015), Adaptability, flexibility, modifiability and scalability, and robustness, *in* 'Nonfunctional Requirements in Systems Analysis and Design', Springer, pp. 169–182.

Aly, M., Charfi, A. & Mezini, M. (2012), On the extensibility requirements of business applications, *in* 'Proceedings of the 2012 workshop on Next Generation Modularity Approaches for Requirements and Architecture', pp. 1–6.

Apple (2013), 'Plug-in architectures'. Available at: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/LoadingCode/Concepts/Plugins.html (Accessed 27 October 2021).

Bachmann, F., Bass, L. & Nord, R. (2007), Modifiability tactics, Technical report, Carnegie-Mellon Univ. Pittsburgh PA, Software Engineering Inst.

Bass, L., Clements, P. & Kazman, R. (2012), *Software Architecture in Practice*, 3rd edition edn, Pearson Education Limited (US titles);Addison Wesley Professional.

Blewitt, A. (2013), *Eclipse 4 Plug-in Development by Example Beginner's Guide*, Packt Publishing Ltd. Code referenced available at: https://github.com/alblue/com.packtpub.e4/tree/edition2/chapter1 (Accessed 21st September 2021).

Bode, S. & Riebisch, M. (2010), Impact evaluation for quality-oriented architectural decisions regarding evolvability, *in* I. G. Muhammad Ali Babar, ed., 'Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings', Lecture Notes in Computer Science 6285 : Programming and Software Engineering, Springer-Verlag Berlin Heidelberg.

Breivold, H. P., Crnkovic, I. & Eriksson, P. (2007), 'Evaluating software evolvability', *Software Engineering Research and Practice in Sweden* **96**.

Brown, A. & Wilson, G. (2011), *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, Vol. 1, Lulu.

Eclipse Foundation (n.d.*a*), 'Eclipse documentation'. Available at: https://help.eclipse.org/2021-09/index.jsp (Accessed 21st September 2021).

Eclipse Foundation (n.d.*b*), 'Nebula repository'. Available at: https://github.com/eclipse/nebula (Accessed 21st September 2021).

Eclipse Foundation (n.d.*c*), 'What is a plug-in?'. Available at: https://wiki.eclipse.org/FAQ_What_is_a_plug-in%3F (Accessed 21st September 2021).

Elgabry, O. (2019), 'Plug-in architecture and the story of the data pipeline framework'. Available at: https://medium.com/omarelgabrys-blog/plug-in-architecture-dec207291800 (Accessed 27 October 2021).

Grosskurth, A. & Godfrey, M. W. (2006), 'Architecture and evolution of the modern web browser', *Preprint submitted to Elsevier Science* **12**(26), 235–246.

Highcharts (n.d.), 'Highcharts – interactive javascript charts library'. Available at: https://www.highcharts.com/ (Accessed 13th November 2021).

Hillard, D. (2020), *Practices of the Python Pro*, 1 edn, Manning Publications.

Hunderi, A. R. (2021), 'Plugin manager fork of the odin repository'. Available at: https://gitlab.stud.idi.ntnu.no/anderrh/extensible-odin (Accessed 4th January 2022).

ISO (2011), Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, Standard, International Organization for Standardization / International Electrotechnical Commission.

Klatt, B. & Krogmann, K. (2008), 'Software extension mechanisms', *Fakultt fr Informatik, Karlsruhe, Germany, Interner Bericht* **8**, 2008.

Microsoft (2020), 'Source code organization'. Available at: https://github.com/microsoft/vscode/wiki/Source-Code-Organization (Accessed 7 September 2021).

Microsoft (2021*a*), 'Extension api'. Available at: https://code.visualstudio.com/api (Accessed 6 September 2021).

Microsoft (2021*b*), 'Why did we build visual studio code?'. Available at: https://code.visualstudio.com/docs/editor/whyvscode (Accessed 27 August 2021).

Mozilla (n.d.*a*), 'Javascript apis'. Available at: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API (Accessed 2 November 2021).

Mozilla (n.d.*b*), 'Webextensions api development'. Available at: https://firefox-source-docs.mozilla.org/toolkit/components/extensions/webextensions/index.html (Accessed 2 November 2021).

OpenJS Foundation (n.d.), 'Electron – build cross-platform desktop apps with javascript, html, and css'. Available at: https://www.electronjs.org/ (Accessed 21 October 2021).

OSGi (n.d.), 'Osgi working group'. Available at: https://www.osgi.org/ (Accessed 04 October 2021).

Rein, P. G. & Tefre, T. S. (2021), Creating a web application supporting git in software development courses in higher education, Technical report, NTNU.

RhodeCode (2016), 'Version control systems popularity in 2016'. Available at: https://rhodecode.com/insights/version-control-systems-2016 (Accessed 6th October 2021).

Rice, D. & Foemmel, M. (2012), Plugin, *in* 'Patterns of Enterprise Application Architecture: Patterns of Enterprise Application Architecture', Addison-Wesley, pp. 499–503.

Rytter, M. & Jørgensen, B. N. (2010), Independently extensibile contexts, *in* I. G. Muhammad Ali Babar, ed., 'Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings', Lecture Notes in Computer Science 6285 : Programming and Software Engineering, Springer-Verlag Berlin Heidelberg.

Sayfan, G. (2017), 'Building your own plugin framework'. Available at: https://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/204702751 (Accessed 27 October 2021).

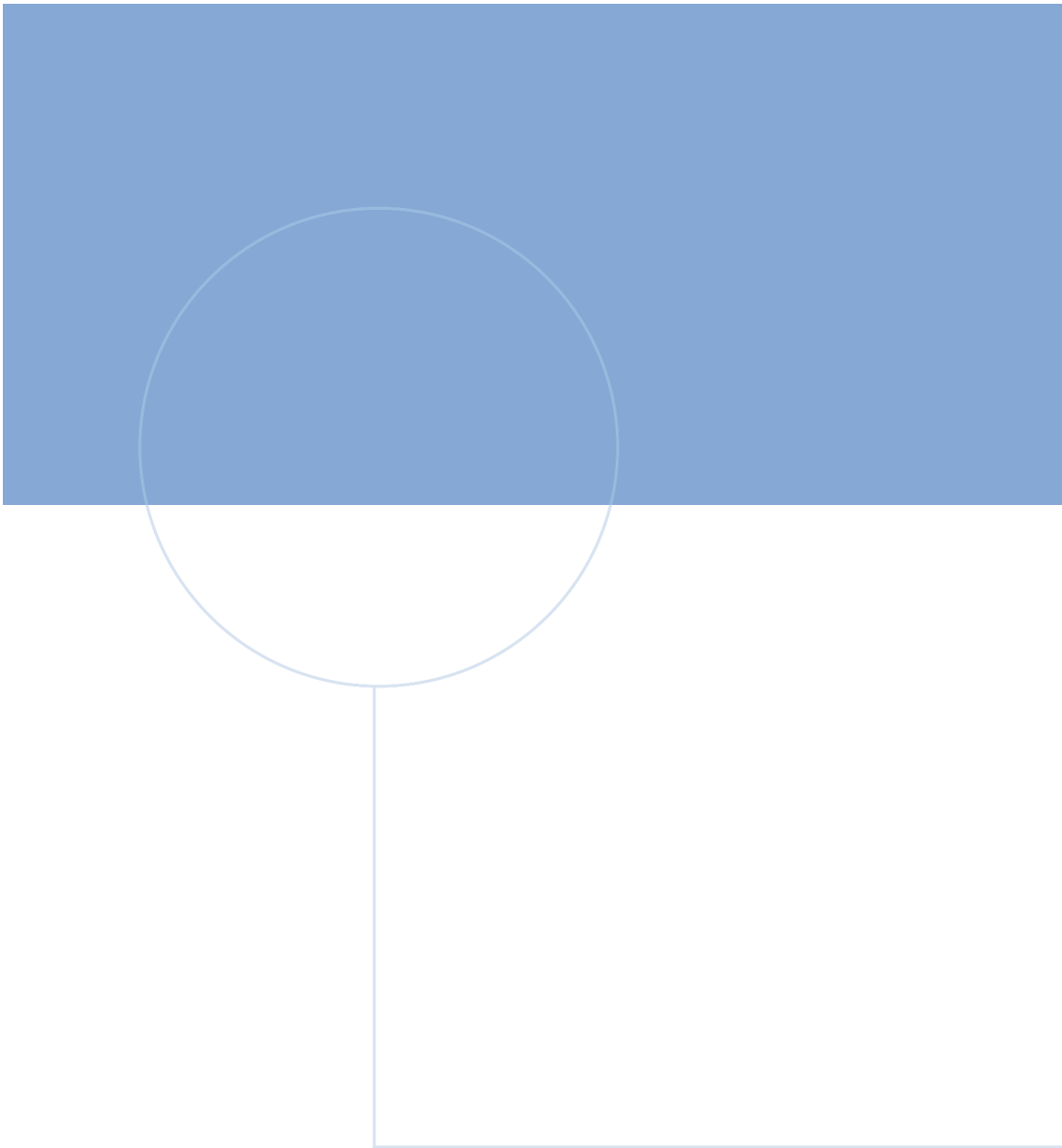
Schipper, D., Faber, J., Proost, R. & Spaargaren, W. (2017), Visual studio code, *in* 'Delft Students on Software Architecture: DESOSA 2017'. Available at: https://delftswa.gitbooks.io/desosa-2017/content/vscode/chapter.html (Accessed 2 November 2021).

Szyperski, C. (1996), 'Independently extensible systems-software engineering potential and challenges', *Australian Computer Science Communications* **18**, 203–212.

Tutorials Eye (n.d.), 'Applications on eclipse'. Available at: https://tutorialseye.com/applications-on-eclipse.html (Accessed 19th August 2021).

Vercel (n.d.), 'Next.js – the react framework'. Available at: https://github.com/eclipse/nebula (Accessed 1st January 2022).

Zenger, M. (2004), Programming language abstractions for extensible software components, Technical report, EPFL.



NTNU

Norwegian University of
Science and Technology