Facilitating Software Reuse

Using Design Science Research To Develop Design Principles For Implementing A Component Repository

Anastasia Bengtsson

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Facilitating Software Reuse

Using Design Science Research To Develop Design Principles For Implementing A Component Repository

Anastasia Bengtsson

Facilitating Software Reuse

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

There is an increase in the development of generic software systems built to serve multiple organizations and used for different purposes. DHIS2, a generic web-based health management information system platform, is an example of such systems and the focus of my thesis. The extensible core of DHIS2 allows the development of complimentary web applications by outside parties as a way of contributing to the DHIS2 platform. The challenge here is that developing these web applications from scratch can be time-consuming and resource-inefficient when similar applications are developed. One way of addressing this challenge is by using the component-based software engineering (CBSE) approach. The main idea behind CBSE is the development of applications by reusing configurable software components. However, previous research identified several challenges that pertain to component reuse, including cataloging and distribution of reusable software components, their trustworthiness, and discoverability.

My Master's project's practical aim was to design, develop, and evaluate a component repository, DHIS2 Shared Component Platform, that facilitates component reuse within the DHIS2 platform ecosystem. This project involved close collaboration with developers in HISP East Africa and the DHIS2 core team at the University of Oslo. The component repository consists of a website (built using React) that aggregates reusable components and two other modules that support the process of component certification: a command-line interface (built using TypeScript) to provide functionality for pre-certification, and a GitHub repository with an automated certification workflow using GitHub Actions workflow.

This study used the Design Science Research (DSR) methodology within a pragmatic research paradigm to contribute to the body of knowledge by developing a set of theoretically and empirically grounded design principles. These principles contribute to the body of knowledge on how component repositories can be designed and developed in a context of a software platform ecosystem. The established set of design principles addresses software reuse challenges identified through empirical data analysis and challenges discussed in the literature on CBSE. These principles address component trustworthiness, component discoverability, the role of a certifier, component repository adoption, its complexity, and maintainability.

Keywords: software reuse, component-based software engineering, software platform ecosystem, design science research, design principles, component repository, component certification, DHIS2

Where an exploratory researcher says, “Gee, that’s funny,” (...) and a theoretical researcher shouts, "Eureka!" the successful AS/E researcher exclaims. "It works!" (...).

(Briggs & Schwabe, 2011, p. 10)

Contents

List of Figures

List of Tables

List of listings

Glossary

The definitions presented in this glossary are properly introduced and cited in the body of this thesis.

application boundary resource boundary resource that enables interaction between the platform core and third-party applications. 15, 16, 102, 117

boundary resources interfaces between the platform core and third-party developers, that allow shifting design capabilities from platform owners towards third-party developers. 3, 13–16, 62, 117

certification formal demonstration that a system or component complies with its specified requirements and is acceptable for operational use. i, viii, 18, 23, 24, 38, 41, 55, 60, 67, 68, 70–77, 79–84, 86–88, 90–92, 94–97, 99–104, 106, 107, 111, 115–117

certifier a person or an organization performing the process of component certification. i, 23, 24, 41, 52, 60, 68–76, 80, 81, 86, 96, 99–102, 106, 107, 111

cohesion the degree to which the elements inside a module belong together, a qualitative measure of consistency of purpose within a module. 30–32, 54, 88, 89, 95, 106

component provider a developer that creates components for further reuse. 4, 17, 21–24, 41, 49–52, 60–62, 65–69, 71–77, 79, 81, 83, 84, 87, 92, 93, 95, 96, 100–102, 104, 106–110, 115, 116

component user a developer that uses reusable components. 4, 21–24, 41, 49–52, 60–62, 66, 68, 70, 75–77, 87, 93, 95, 96, 98, 100, 102–104, 107–109, 115–117

coupling the measure of the strength of association established by a connection from one module to another. 20, 30–32, 54, 88, 89, 95

development boundary resource boundary resource that support the third-party development process. 15, 16, 26, 102, 105, 116, 117

maintainability degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. 17, 28, 29, 31, 32, 76, 88, 89, 107

modularity degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. 19, 25, 28, 29, 40, 88

modularization a software design approach that implies dividing a software system into multiple independent modules with well-defined interfaces that describe interactions between the modules. 17, 28, 29, 31, 32, 88, 106, 112

social boundary resources boundary resources that play a supportive role in third-party development and are usually knowledge-based. 14–16

technical boundary resources boundary resources that provide technology necessary for third-party development including APIs, software development kits (SDKs), open data, and secure protocols. 14–16

unit testing the process of testing program components, such as methods or object classes. 6, 49, 55, 90

Acronyms

API Application Programming Interface. 3, 9, 15, 78, 80, 81, 83–85, 94, 95

B2B business-to-business. 12, 62, 108

B2C business-to-customer. 12

C2C customer-to-customer. 12

CBSE component-based software engineering. i, 1, 2, 4, 5, 11, 16–26, 40, 49, 51, 52, 63–68, 76, 88, 89, 96, 98, 103, 105, 107–109, 115–117

CLI command-line interface. viii, ix, 49, 55, 58–60, 67, 71, 76, 77, 79–85, 90–92, 95, 106, 107

CSS Cascading Style Sheets. 9, 59, 78

DHIS District Health Information Software. 8

DHIS2 District Health Information Software 2. i, xv, 1–3, 5, 6, 8–11, 13, 14, 25, 26, 38, 41–43, 50–52, 60–63, 65–72, 74–77, 79, 80, 85–87, 96, 98–102, 105, 107–112, 116, 117

DSR Design Science Research. i, 6, 25, 33, 35–38, 40, 43, 46, 47, 49, 51, 54, 56, 110, 111, 115

HISP Health Information Systems Programme. i, xv, 1–3, 5, 6, 8–10, 26, 34, 38, 41, 50, 58, 60, 62–65, 67, 68, 74, 85, 87, 94, 98, 99, 101, 104, 105, 107–111, 115–117

II information infrastructure. 25–28

NPM Node Package Manager. 55, 60, 64–71, 75–81, 83–85, 87, 91–95, 99, 100, 104, 105, 107, 109, 110, 115

SCP DHIS2 Shared Component Platform. i, viii–x, 2, 4–6, 11, 41, 45, 48–51, 53–56, 58–63, 66–85, 90–97, 99, 100, 102, 105–107, 110, 112, 115, 116, 127, 128

SDK software development kit. 15

UI user interface. 38, 66, 69, 73, 78, 102, 105

UiO Universitetet i Oslo. 1–3, 9, 26, 60

Acknowledgments

First and foremost, I extend my heartfelt appreciation to my supervisor, Petter Nielsen, for his academic guidance and personal support throughout the project work and writing of this thesis. Your constructive comments, insightful discussions, and warm encouragement were instrumental in helping me finish this thesis.

I would also like to express my gratitude to my co-supervisor, Magnus Li, for the tremendous amount of support and invaluable insights he gave the whole DHIS2 Design Lab and me during the past two years. I want to thank HISP UiO, the DHIS2 core team, the developers from HISP Mozambique and HISP Tanzania, and the DHIS2 Design Lab for all the help with this project. I would like to acknowledge my colleagues for all the determination and effort, all the hard work done, and every line of code written to bring our project through from conception to completion. My special thanks go to Håkon André Heskja. Thank you for our fruitful and engaging discussions and all the support you have given me.

Special thanks to Hanna Kongshem and Anders Brustad for doing all the work as DHIS2 Design Lab coordinators, for their enthusiasm and optimism.

I am deeply indebted to my four anonymous evaluation participants, all of whom expressed a genuine interest in my research and helped me conduct an evaluation at the most inopportune time - during the Christmas season.

Lastly, I would like to thank my family and my sweet cat Wega for their constant support and unconditional love.

Chapter 1

Introduction

1.1 Research context and motivation

There is an increase in the development of generic software systems built to serve multiple organizations and used for different purposes. Some generic software examples are MATLAB, WordPress, Adobe Photoshop, and DHIS2 - a generic web-based health management information system platform which is the foundation of this research.

The purpose of Health Management Information Systems is to support routine management and generation of health information data that serves as a basis for management decisions to foster improvements in health service provision. Currently, DHIS2 is the world's largest health management information system, and it is being used by 73 low- and middle-income countries (DHIS2, 2021a, para. 1). HISP is a global network that supports and develops the DHIS2 platform, "the benefits of which will be felt by all the countries that are part of the network" (PAHO eHealth, 2021, para. 3). The core DHIS2 software development is coordinated by HISP at the University of Oslo (HISP UiO) — "a professionalized software development organization" (Adu-Gyamfi, Nielsen, & Sæbø, 2019, p. 73). HISP is comprised of independent HISP groups, e.g., HISP South Africa and HISP Tanzania; Universities, e.g., University of Dar es Salaam in Tanzania and Universidade Eduardo Mondlane in Mozambique; Ministries of Health, NGOs, global policy-makers, researchers, students and more (Adu-Gyamfi et al., 2019, p. 75). One way of contributing to DHIS2 is by developing additional applications on top of the DHIS2 generic software, extending its user interface and functionality. One example of these applications is the WaterpointUserManagement application built by HISP Tanzania (Mpande, 2018).

Building these web applications from scratch can be time-consuming. It is also not resource-efficient when different HISP groups develop similar web applications. One way of overcoming this challenge is by building those web applications from existing components, which are the building blocks those applications are constructed with, using a component-based software engineering approach (CBSE). The process of software component reuse lies at the core of CBSE, promising "high quality, low budget software with shorter time to market" (Kaur & Singh, 2009, p. 1). My research focuses on facilitating component reuse within the DHIS2 platform ecosystem by developing a component repository, DHIS2 Shared Component Platform (SCP). This component repository addresses identified challenges that the HISP community faces with component reuse.

Modern Web development frameworks, such as React and Angular, are built around the concepts of components. A common way to use these frameworks is by building up web applications using components one created or created by others. The use of these frameworks thus implies a certain level of component-based software engineering. However, getting the benefits of CBSE would require not just creating and using components within one project or web application but reusing them across multiple web applications.

Several HISP groups use these component-based web frameworks and reuse components created internally and, in some cases, components created by other HISP groups. However, there are several known barriers to the process of component reuse discussed in the literature, e.g., poor component cataloging and distribution, and lack of certification (Jalender, Govardhan, & Premchand, 2010). The challenges pertaining to component reuse in the HISP community identified during this project are limited component reuse between HISP groups, poor component cataloging and discoverability, and an absence of component quality assurance.

DHIS2 Design lab, a part of HISP UiO, responded to this challenge of component reuse in the HISP community by proposing the design and development of a component repository to facilitate the process of component reuse in the DHIS2 platform ecosystem (Li, 2020, para. 2). This task was carried out by me and two other Master's students, Håkon André Heskja and Mr. Bingley[1], as our Master's project. My team and I digitally collaborated with developers in HISP East Africa and the DHIS2 core team through interviews and focus groups to learn about their application development process and component reuse practices. We then applied this knowledge in practice when developing the practical contribution of this thesis - DHIS2 Shared Component Platform. DHIS2 Shared Component Platform is built to tie the component reuse effort together for the HISP community by providing functionality that allows developers to share and find reusable components. Additionally, it provides capabilities for component quality assurance. The theoretical contribution of my thesis is a set of design principles for implementing a component repository in a software platform ecosystem. This knowledge provides value to the HISP community and can also help others who are building systems that try to solve similar problems.

[1]This is a pseudonym that I have given to one of my colleagues to ensure his anonymity and protect his privacy.

1.1.1 DHIS2 Web application development

DHIS2 is a generic open-source software platform designed around the standard software platform architecture with two major elements - a stable core and complementary applications. According to Tiwana (2014), "a *software platform* is a software-based product or service that serves as a foundation on which outside parties can build complementary products or services" (p. 5).

A platform owner responsible for the governance of the platform core facilitates third-party development through the provision of boundary resources, such as APIs, open data, and secure protocols. The DHIS2 platform core is owned and governed by HISP UiO (Adu-Gyamfi et al., 2019). According to Ghazawneh and Henfridsson (2013), boundary resources aim at promoting third-party application development that is an integral part of a thriving platform ecosystem. Additionally, their purpose is to give the ability to platform owners to gain control in situations where third-party development is harmful to the platform (Ghazawneh & Henfridsson, 2013).

DHIS2 web applications are created to address issues specific to their context of implementation. Given the open nature of the DHIS2, web application development is open to many different actors, including students, the DHIS2 core team developers, implementation-level specialist groups, etc. According to Ghazawneh and Henfridsson (2010), third-party app development is "an innovation network distributed across multiple actors and technologies" (p. 4) and is one of the strategies for value creation in the ecosystem (Boland, Lyytinen, & Yoo, 2007; Chesbrough, 2003; Lyytinen, Yoo, & Boland, 2015). This suggests that increasing the ease of third-party application development will increase the value creation potential, essential for a successful platform ecosystem - self-reinforcing and bringing long-term value to its users.

1.1.2 Component-based software engineering

It is quite common for multiple software applications to share similar functionality, even if the software applications themselves are vastly different. An example could be functionality that handles file selection, which can be relevant in many different applications where a user would need to select a local file on his computer. One approach to improving the software development process's efficiency has been to reuse the same implementation of some functionality, i.e., the code, in multiple software applications. The idea being that this reused or shared functionality will only have to be developed once, and the work required to implement the software applications is reduced by reusing this functionality as it will not have to be reimplemented by different application developers. Furthermore, with this approach, maintenance becomes easier because defects and vulnerabilities can be addressed once in the reusable component, and all developers can get the updated components. Similarly, this approach makes it simpler to roll out new features and functionality to developers. However, this approach brings some challenges in that it makes developers dependent on the continued maintenance of the components they use to develop their software.

A specific means of achieving the reuse of software functionality is component-based software engineering (Tiwari & Kumar, 2020). A software component encapsulates some specific functionality and can be reused in various applications that require this functionality. Component-based software engineering covers the creation, maintenance, distribution, and use of these software components (Capretz, 2005).

CBSE can be further subdivided into CBSE for reuse and CBSE with reuse (Kotonya, Sommerville, & Hall, 2003). CBSE for reuse is concerned with the process of creating software components for later reuse, and CBSE with reuse is concerned with using reusable components to construct applications (Sommerville, 2011). The concept of *component provider* refers to those generally engaged in the process of CBSE for reuse (i.e., creating components) and the concept *component user* refers to those generally engaged in CBSE with reuse (i.e., using components). This study uses these concepts to distinguish between the web application developers that create and publish reusable components and the web application developers that acquire and use reusable components to build web applications. It is important to note that a developer can have both roles, i.e., creating reusable components and reusing components to build web applications.

The purpose of DHIS2 Shared Component Platform is to facilitate the component reuse process that lies at the core of CBSE; thus, the CBSE body of knowledge played an important role in the design and development of this platform.

1.1.3 Component repository in a software platform ecosystem

Platform ecosystems are different from traditional businesses when it comes to their structure, market potential, and management style (Tiwana, 2014). While the boundaries of value creation in traditional business are well-defined, platforms gain their value from a vast number of autonomous participants. According to Tiwana (2014), a platform owner has only partial control over a platform, thus requiring management through orchestration and coordination, keeping the balance between platform control and autonomy of application developers. Contrary to the command-and-control strategy in traditional businesses, a platform owner should guide and coordinate application developers, gently leading into desired direction (Tiwana, 2014). This complex setting also has implications on a component repository that should operate in it, as a component repository cannot be adopted through a command-and-control management strategy in this context. Additionally, within a platform ecosystem, reuse has to occur between multiple independent organizations that could take on vastly different roles. Thus the context of a platform ecosystem should be considered during the design and implementation of a component repository that should operate within this context.

1.2 Research question

In order to investigate how a component repository can be designed to facilitate the component reuse process, the following research question was established:

> RQ: What are the essential design principles for implementing a component repository that facilitates component reuse in a software platform ecosystem?

1.3 Research aim

This project aimed at attaining two aims — a practical one and a theoretical one. The practical aim was a response to a call to explore the possibility of designing and developing a component repository that facilitates software reuse in web application development in a software platform ecosystem. Therefore a primary focus of the practical work in this project was the design, implementation, and evaluation of such a repository in collaboration with the DHIS2 core team and HISP groups' developers involved in web application development work.

The theoretical aim of my research was to identify and establish a set of theoretically and empirically grounded design principles for implementing a component repository in a software platform ecosystem. These design principles contribute to the knowledge base on how component repositories in the context of a software platform ecosystem can be designed and developed.

1.4 Research objectives

The scope of this research includes the following research objectives that drove the achievement of the research aims:

- Establish the objectives and requirements of the artifact (i.e., SCP) by exploring and gaining an understanding of web application development practices with the focus on component reuse processes within the HISP community. To attain this objective, I used the following methods: interviews and focus groups with HISP East Africa developers and the DHIS2 core team.
- Review of the literature on software platform theory and CBSE in order to develop a theoretical framework for the research.

- Design and develop a component repository in collaboration with the DHIS2 core team and HISP East Africa developers using the insights gathered from literature, the objectives, and requirements. Development work, formative evaluations during focus group sessions, and prototyping were done to attain this objective. This component repository is the artifact that serves as the practical contribution of my project. The design principles were established during this activity.
- Perform a final formative evaluation of the DHIS2 Shared Component Platform, the artifact, according to several criteria and evaluate the application of the established design principles. Surveys, expert evaluation, and unit testing were chosen as evaluation methods.
- Establish the theoretical contribution of my thesis - design principles for implementing a component repository in a software platform ecosystem, and discuss them by considering the evaluation and the literature review.

1.5 Research methodology

I have chosen Design Science Research (DSR) as overarching methodology to guide the design and development of SCP. It is a problem-solving methodology, meaning that it can be used to construct a new artifact as a solution to an existing problem. While engaging in design and development activities, a researcher is expected to develop knowledge that can be useful for other researchers and practitioners to solve similar problems.

1.6 Thesis structure

Chapter 1 - Introduction

This chapter includes the research context and motivation, my research question, research aim and objectives, and the structure of the remainder of the thesis.

Chapter 2 - Background

This chapter provides additional background pertaining to the conducted research.

Chapter 3 - Literature review

This chapter includes prior work that is relevant for the research and the design of the artifact.

Chapter 4 - Kernel theories

This chapter includes theories that influenced the design and development process of the artifact.

Chapter 5 - Research approach

This chapter includes the description of chosen research paradigm and methods for the research and work distribution during the design and development process.

Chapter 6 - Artifact description

This chapter includes a discussion on the artifact's role in the platform ecosystem, the artifact's design considerations and its architecture, and the set of established design principles.

Chapter 7 - Evaluation

This chapter details the artifact's evaluation according to various criteria and evaluation of the application of the established design principles.

Chapter 8 - Discussion

This chapter presents the answer to the research question and discusses it in light of the findings, evaluation and literature review.

Chapter 9 - Conclusion and future work

This chapter provides a conclusion to this study and offers suggestions for future work.

Chapter 2

Background

In this chapter, I introduce DHIS2, its historical background, and elaborate on its technical aspects. I also present DHIS2 Design Lab, which this project is a part of.

2.1 DHIS2

The government of post-apartheid South Africa in 1994 planned to reconstruct the health sector in the various provinces of the country; these reconstruction plans included a project for developing a district-based health system (Adu-Gyamfi et al., 2019). As part of this initiative, HISP was founded, initially, as a "collaborative project involving the University of Cape Town, the University of Western Cape and a Norwegian PhD candidate from the University of Oslo" (Adu-Gyamfi et al., 2019, p. 74). The goal was to design and implement a solution "for integrated and decentralized information support for district health management" (Braa & Sahay, 2017, p. 2). Braa and Sahay (2017) explain that the challenging part in this was the extreme fragmentation of health data because of segregation of healthcare systems as a consequence of racial and geographical segregation in apartheid South Africa. The first DHIS was developed using Microsoft Access and Visual Basic; it was first tested in the various districts of the greater Cape Town area. By 1998 the first DHIS was scaled to the whole Western Cape province, and in 2001 it was adopted by South Africa's Department of Health Care and was later implemented in all districts of South Africa (Adu-Gyamfi et al., 2019). The expansion of DHIS uncovered problems caused by the technology that the DHIS was built upon - the system was dependent on Microsoft Windows and Microsoft Office, each standalone installation of the system needed maintenance, and thus required the presence of a large maintenance team (Adu-Gyamfi et al., 2019). In order to address these challenges, the development of the DHIS2 was started at the University of Oslo in 2005 and resulted in a system with a client-server architecture supporting centralized maintenance and distributed software development.

Currently, DHIS2 is used in 73 low- and middle-income countries, and if its use by NGO-based programs is included in the count, then it is used in more than 100 countries (DHIS2, 2021a, para. 2). The development of DHIS2 is lead by HISP UiO, where the DHIS2 core team is based.

DHIS2 is an enterprise system, meaning that each country has to implement its own local instance of DHIS2, subsequently taking full ownership of their instance and the data within it (DHIS2, 2021a, para. 11). DHIS2 is not only open-source and thus free of charge but also a software platform with a modular layered architecture that allows the development of applications that extend its core. It is a web application built using Java technology, which implies that it "runs on any Java enabled web server or servlet container, and can be accessed via a web browser over the Internet" (DHIS2, 2021c, para. 2). Furthermore, it can be deployed in various ways - on an online server, in the cloud, or in an offline setting (DHIS2, 2021c, para. 2). DHIS2 provides RESTful Web API, allowing the development of custom applications using web technologies such as JavaScript, HTML5, and CSS (DHIS2, 2021c, para. 3).

2.2 The HISP network

HISP is "a global movement"(DHIS2, 2021b, para. 1) that provides support to DHIS2 development, a local implementation that consists of the processes of configuration, customization, and extension; and various training programs (DHIS2, 2021b). The HISP network consists of HISP groups that are "long term and trusted partners located in developing countries" (Nielsen, 2021, para. 4) working in partnership with HISP UiO. This collaboration includes various activities, such as the development of DHIS2, applications for DHIS2; capacity building activities, including arranging DHIS2 Academies; research and academic activities; and implementation support (Nielsen, 2021, para. 4). The HISP network consists of different regional groups as shown in Table 2.1.

HISP regional group	**HISP groups**
HISP East Africa	HISP Uganda, HISP Mozambique, HISP Tanzania, HISP Rwanda, HISP Malawi, and HISP Ethiopia
HISP Southern Africa	HISP South Africa
HISP West and Central Africa	HISP West & Central Africa, and HISP Nigeria
HISP Asia and the Pacific	HISP Vietnam, HISP Sri Lanka, HISP India, and HISP Bangladesh
HISP Latin America & the Caribbean	HISP Colombia

Table 2.1: The HISP network.

2.3 DHIS2 Design Lab

The DHIS2 design lab is a *generic software design lab* which aims at strengthening "the usability and local relevance of the generic software DHIS2 for end-users" (Li, 2019, p. 11). According to Li (2019), the DHIS2 design lab is "an entity somewhat independent" (p. 6) from the DHIS2 core team developers and HISP groups; however, the participants of the lab are part of the projects that involve close collaboration with the HISP community members including the DHIS2 core team and HISP groups. At the time of writing this thesis, the lab consisted of one faculty representative, a Ph.D. researcher, and more than 30 Master's students studying computer science at the Department of Informatics at the University of Oslo. As a member of the DHIS2 design lab, I attended regular meetings to discuss the lab's ongoing projects, including practical and theoretical aspects, and share our experiences. Participation in the lab made it possible to collaborate with the HISP community members mainly through virtual group and one-to-one interviews, and focus groups. Additionally, the lab has organized various activities, such as thesis co-readings, that have helped me better communicate my research.

Chapter 3

Literature review

This chapter introduces the literature of high relevance to this study, underpinning the research presented in my thesis. Our artifact was built to facilitate component reuse and, consequently, improve web application development in the DHIS2 platform ecosystem. Therefore, in Section 3.1 and Section 3.2, I present the concepts and theory for understanding the research context of our artifact: DHIS2 as a software platform, third-party development as innovation, and SCP's role as a boundary resource and as a nested transaction platform. Section 3.3 on component-based software engineering serves as the design theory, providing a theoretical foundation necessary for understanding the process of component reuse, which was guiding our design and development work.

3.1 Digital platforms

Many of the biggest and successful companies worldwide, such as Apple, Amazon, and Facebook, have adopted a platform business model. The main goal of this model as a means to value creation is to bring distinct users together for interaction and transaction activities. For example, Apple facilitates interaction between several different actors: third-party application developers who supply the platform with applications; end-users who purchase these applications; and different advertising companies attracted by all the valuable data that can be used for marketing purposes. Another known example is Steam, a digital game distribution platform that brings together gamers to engage in multiplayer gaming and social networking, game developers, and game publishers. A platform that enables interaction between different kinds of users that bring value to one another is usually characterized as multi-sided (de Reuver, Sørensen, & Basole, 2018). Many digital platforms share the following common characteristics: they are driven by technology, they enable interaction between distinct user groups, and allow these groups to engage in certain activities (Constantinides, Henfridsson, & Parker, 2018, p. 9; de Reuver et al., 2018, p. 125; Jacobides, Cennamo, & Gawer, 2018, p. 5; Koskinen, Bonina, & Eaton, 2019, p. 320).

Digital platforms lie at the core of a platform ecosystem. Koskinen et al. (2019) argue that platforms are "rarely isolated" (p. 323), and in addition to the end-users, there are always different organizations, actors, and regulations that surround a platform, forming its *ecosystem*. The platform ecosystem model is concerned with the interactions between its actors, focusing on how they engage with each other and generate value (Constantinides et al., 2018, p. 1). Constantinides et al. (2018) further add that digital infrastructures serve as a foundation for platform creation and cultivation. Digital infrastructures are usually represented by resources and organizational structures that are necessary for a platform to function. The Internet, cloud services, and data centers are some examples of digital infrastructures (Constantinides et al., 2018).

Koskinen et al. (2019) and Gawer (2009) distinguish between three types of digital platforms: *transaction platforms*, *innovation platforms*, and *integration platforms*. The first type, transaction platforms, according to Koskinen et al. (2019), is "sometimes referred to as multi-sided markets or exchange platforms" (p. 321). The purpose of such platforms is to facilitate the exchange of products or services among different users, and one of the most important features of transaction platforms (and digital platforms in general) is *network effects* (Koskinen et al., 2019). Tiwana (2014) defines *network effects* as a "property of a technology solution where every additional user makes it more valuable to every other user on the same side (same-side network effects) or the other side (cross-side network effects)" (p. 25). In Steam, same-side network effects occur when every new joining player creates value for the other players enabling multiplayer gaming. The cross-side network effects in Steam occur when game publishers publish newly developed games on the platform, attracting potential players. After conducting a review of research on transaction platforms in a marketing journal, Hänninen (2020) notes that many articles focus on three different perspectives on transaction platforms based on the transaction type. The first type, discussed by Hänninen (2020), is a business-to-customer (B2C) transaction platform that focuses on intermediating transactions between businesses and customers. Amazon is an example of a B2C transaction platform. Second type is a customer-to-customer (C2C) transaction platform that intermediates transactions between customers. While Facebook and Finn.no marketplaces are B2C transaction platforms, they can also be viewed as C2C platforms because their customers can engage in buying and selling activities. The third type, a business-to-business (B2B) transaction platform, is designed to enable transactions between businesses. An example of a B2B transaction platform is the Amazon Business marketplace that provides a venue for sellers who target business buyers.

The second type of digital platform is innovation platforms (also known as software platforms). Tiwana (2014) defines a *software platform* as "the extensible codebase of a software-based system that provides core functionality shared by apps that interoperate with it, and the interfaces through which they interoperate" (p. 7). The main difference between transaction and software platforms is the ability of software platforms to provide a basis for innovation in the form of complementary products developed by third-party developers or complementors. In the case of DHIS2, web application development is inherently third-party development, and web application developers are third-party developers.

Tiwana (2014) defines the main elements of a software platform ecosystem (Figure 3.1) - a platform core (extensible codebase) and complementary apps that interact with the platform core through interfaces, namely boundary resources. Boundary resources and their types will be further discussed in Section 3.2.

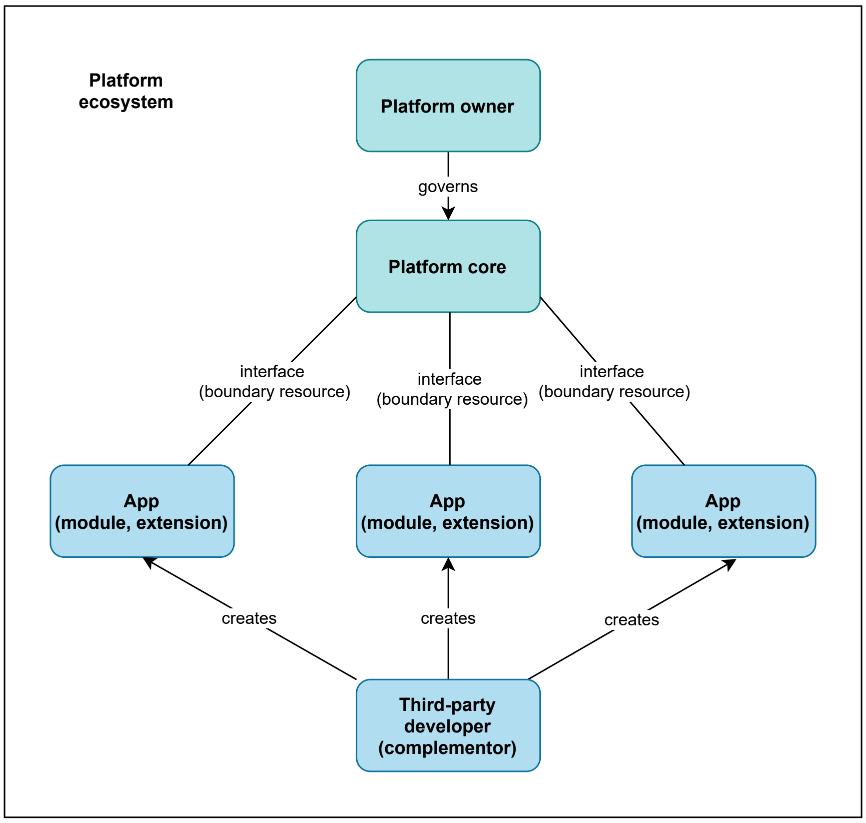


Figure 3.1: Components of a software platform ecosystem.
Note. Adapted from *The rise of platform ecosystems* (p. 6), by A. Tiwana, MorganKaufmann. Copyright 2014 by Elsevier Inc.

Android and Apple are examples of software platforms, which provide a foundation for platform innovation in the form of third-party applications. Koskinen et al. (2019) state that studies within information systems and engineering have a particular focus on this type of digital platforms and their innovation capabilities. Koskinen et al. (2019) also underscore the important role of the boundary resources in a software platform ecosystem and the interplay between platform owners and third-party developers, as these aspects directly impact platform innovation and value creation.

The third type of digital platform is integration platforms (also referred to as integrated platforms). Koskinen et al. (2019) and Evans and Gawer (2016) view integration platforms as a combination of the transaction and software platform types. For example, Apple iOS can be viewed as an integration platform as it has both the software platform surrounding the applications for iOS and it has a nested transaction platform in the form of the Apple App Store, and the Apple App Store would not make much sense without the existence of Apple iOS. There are, of course, some exceptions to this view, such as eBay, which is a transaction platform, but which is not tied to any specific software platform.

My thesis focuses mainly on software platform and its ability to enable platform innovation through boundary resources because our solution aims to support third-party application development. Besides, I view the component repository as a nested transaction platform within the DHIS2 software platform and argue for the importance of network effects (Section 6.1.1).

3.2 Boundary resources

Ghazawneh (2012) emphasizes the fact that recent research recognizes the importance of third-party development as a contribution to value creation in a platform ecosystem. Tiwana (2014) argues that the variety of complementary applications makes platforms more desirable for their target user groups. Therefore a platform aims to facilitate third-party development activities and provide resources to support this activity (Ghazawneh, 2012; Tiwana, 2014). Boundary resources can be viewed as interfaces between the platform core and third-party developers that allow shifting design capabilities from platform owners towards third-party developers (Ghazawneh, 2012). On the one hand, provided boundary resources aim at promoting third-party application development that is an integral part of a thriving platform ecosystem; on the other hand, their purpose is to give the ability to platform owners to gain control in situations where third-party development is harmful (Ghazawneh & Henfridsson, 2013). Therefore, they play an important role in balancing platform innovation and platform control and shape platform dynamics. Platform innovation is a key aspect of generativity; thus, boundary resources play a key role in generativity. *Generativity* is defined as "a technology's overall capacity to produce unprompted change driven by large, varied, and uncoordinated audiences" (Zittrain, 2005, p. 1980). Zittrain (2005) indicates that innovative capacity is driven by technology, while Msiska and Nielsen (2018) point out that innovation is "always a collective and social activity" (p. 399-400). Given this, generativity is socio-technical and can be viewed from technology-oriented and social perspectives (Msiska & Nielsen, 2018). This socio-technical nature is clearly expressed in boundary resources, as, according to Ghazawneh (2012), they can be divided into two types: *technical boundary resources* and *social boundary resources.* Technical boundary resources supply third-party application developers with the necessary technology such as APIs, software development kits (SDKs), open data, and secure protocols, etc (Bianco, Myllarniemi, Komssi, & Raatikainen, 2014; Ghazawneh, 2012). Social boundary resources, such as documentation and guides, play a supportive role in third-party development and are usually knowledge-based (Bianco et al., 2014). Bianco et al. (2014) argue that there are two types of technical boundary resources - *application boundary resources* and *development boundary resources*. The main difference between these two types is that application boundary resources enable interaction between the platform core and third-party applications, while development boundary resources aim to support the third-party development process (Bianco et al., 2014). An API is an example of an application boundary resources. Tooling that supports application development and testing is an example of development boundary resources. Bianco et al. (2014) state that development boundary resources are not required in the ecosystem, as they only play a supportive role, making software ecosystem "more attractive" (p. 16).

Ghazawneh (2012) presents the "Boundary resource model" that provides insights into the balance between the securing and resourcing drivers behind boundary resource design. Ghazawneh (2012) states that his research findings uncover four types of insights pertaining to resourcing and securing - *self-resourcing*, *regulation-based securing*, *diversity resourcing*, and *sovereignty securing*.

I will now provide a summary of these insights. The action of self-resourcing occurs when the third-party developers develop their own boundary resources. It usually happens because of the limitations of existing boundary resources and platform governance leaning towards securing. As an example of self-resourcing, Ghazawneh and Henfridsson (2013) discuss the case when some developers jailbreaked the iPhone to be able to install third-party native applications. Regulation-based securing is the act of limiting third-party development through regulations that affect social boundary resources. Apple's application review process is an example of this, where Apple decides whether or not an application will be published in the Apple App Store (Ghazawneh & Henfridsson, 2013). Diversity resourcing can be viewed as actions that extend platform capabilities to stimulate development in new application areas. Apple's actions to open their platform to third-party developers by introducing an API and SDK is an example of this, which enabled third-party developers to develop their own applications (Ghazawneh & Henfridsson, 2013). The last insight, sovereignty securing, is the platform owners' actions to retain control over their platform's evolution. An example of such an action is Apple's introduction of additional terms in their license agreement which limited the programming languages that can be used to develop applications for the iPhone and thus made Adobe's "Packager for iPhone," which converted Flash applications to iPhone applications, obsolete, as applications that use it would violate the license agreement (Ghazawneh & Henfridsson, 2013, p. 184-185).

To summarize, there are two main types of boundary resources: technical boundary resources and social boundary resources. Further, technical boundary resources can be either application boundary resources or development boundary resources. The role of application boundary resources is to facilitate interactions between complementary applications and platform core, while development boundary resources are to support application development.

3.3 Component-based software engineering

In this section, I provide the review of the literature on component-based software engineering, which serves as a design theory for the design and development of our artifact. First, I will explain what CBSE is and why this approach can be beneficial, elaborate on its challenges, and explain what a software component is. Then I will discuss the CBSE for reuse process, which involves creating reusable components, and CBSE with reuse process, which focuses on using reusable components for building web application. Additionally, I will discuss the process of component management that involves various management activities for components and discuss the role of a component repository. Furthermore, I will discuss the component certification process and elaborate on its benefits.

Component-based software engineering came to light in the late 1990s as a software development based on the idea of software reuse (Crnkovic & Larsson, 2002, p. xxviii; Sommerville, 2011, p. 455). With its philosophy "buy, don't build" (Tiwari & Kumar, 2020, p. 21), CBSE aims at reusing pre-constructed and available components, instead of putting time and effort on developing them from scratch (Tiwari & Kumar, 2020). The idea is to develop a software component and then keep reusing the same component in other applications, instead of developing the same functionality every time (Tiwari & Kumar, 2020, p. 21). Sommerville (2011) discusses that the creation of this approach was driven by software designers' dissatisfaction with object-oriented software development, as it did not fulfill the expectations of the high occurrence of software reuse, as it was meant to. Sommerville (2011) explains that in order to reuse object classes, developers had to have comprehensive knowledge about their inner workings, which had made the distribution or selling of class objects impractical. Software components differ from class objects by having a higher level of granularity, and they are usually larger and composed of multiple object-classes (Crnkovic & Larsson, 2002; Sommerville, 2011). Software components are characterized by *information hiding* (also known as data hiding). ISO/IEC and IEEE (2017) defines information hiding as a "software development technique in which each module's interfaces reveal as little as possible about the module's inner workings and other modules are prevented from using information about the module that is not in the module's interface specification" (p. 220). Component-based software engineering is concerned with the process of developing and integrating loosely coupled independent components into software systems (Sommerville, 2011, p. 453).

Software systems can become large and complex, and adoption of CBSE approach can reduce its time to market, increase productivity, reliability and scalability of a system, as well as improve its quality (Crnkovic & Larsson, 2002, p. xxviii; Jalender, Govardhan, & Premchand, 2010, p. 40; Kotonya et al., 2003, p. 1; Szyperski, 2002, p. xxi; Tiwari & Kumar, 2020, p. 21). CBSE has the following advantages:

> *Reusability.* CBSE approach "relies on reusing [components] rather than re-developing them" Tiwari and Kumar (2020, p. 25). *Software reuse* is defined as a process of "building a software system at least partly from existing pieces to perform a new application" (ISO/IEC & IEEE, 2017, p. 385) and *reusability* as the "degree to which an asset can be used in more than one system, or in building other assets" (ISO/IEC & IEEE, 2017, p. 384).
>
> *Shorter development cycle.* Tiwari and Kumar (2020) argue that software reuse and support of modularization results in an increased development speed and, consequently, a shorter development life cycle.
>
> *Maintainability.* *Software maintenance* is a "totality of activities required to provide cost-effective support to a software system" (ISO/IEC & IEEE, 2017, p. 420). Such activities include adding new functionality of removing and updating old features. Since software components can be added and removed without affecting other parts of the system, it becomes easier to maintain such a system. Moreover, it is easier to maintain a system comprised of independent components than "maintaining monolithic software" (Tiwari & Kumar, 2020, p. 25).
>
> *Improved quality and reliability.* CBSE can also give an improvement in quality and reliability of software, but only if reusable components are pre-tested and qualified for reuse (Tiwari & Kumar, 2020, p. 25-26).

However, CBSE approach has its challenges. The challenges I have identified during the literature review are:

> The challenge of *Component specification.* Crnkovic and Larsson (2002, p. xxxi) assert that there is no universal definition of what a software component is and how it should be specified. A software component has two distinct parts - code and interfaces, which are the only entry points of access to the component. Thus, component provider must provide all the necessary information about the component and its operations.
>
> The challenge of *Component trustworthiness.* According to Sommerville (2011), this challenge is concerned with software components being black-boxes for end-users, making it impossible to detect undocumented failure modes, unexpected non-functional behavior, or even

malicious code. Reusing such components may put a system at risk. In order to address this challenge, one can use an approach called component *certification* (also referred to as qualification).

The challenge of *Component certification.* Component certification is an approach that aims to increase component trustworthiness. But it is also a challenge in CBSE (Crnkovic & Larsson, 2002; Jalender, Govardhan, & Premchand, 2010; Sommerville, 2011). For instance, it is not clear who should take the role of an independent certifier, how the components should be certified, and what metrics should be used.

The challenge of *Composition predictability.* Even if the attributes of a component are completely and correctly specified, the attributes of a system constructed from these well-specified components are undefined without some rules of inference that makes it possible to derive system properties from the attributes of components it is constructed from (Crnkovic & Larsson, 2002, p. xxxii).

The challenge of *Requirements trade-offs.* When you build software using existing components, you might have to make trade-offs between the ideal requirements and available components. Sommerville (2011) suggests that there is a need for "a more structured, systematic trade-off analysis method to help designers select and configure components" (p. 454).

The challenge of *Tools support.* Effective CBSE adoption requires appropriate tooling, e.g., "component selection and evaluation tools, component repositories, (...) component configuration tools, etc" (Crnkovic & Larsson, 2002, p. xxxiii). Kotonya et al. (2003) suggest that there is a need for tooling to support component reuse and management.

Reusability and usability dissension. Crnkovic and Larsson (2002) argue that for a component to be widely reusable, it has to be general and sufficiently scalable, and adaptable, which may become an issue for usability.

Prejudice against software reuse. Jalender, Govardhan, and Premchand (2010) draw our attention to the fact that reuse is not properly taught to computer science students, and students begin to associate code reuse with cheating. "This initial educational bias often manifests itself later in programmers' careers as the suspicion of 'if I reuse it, others will think I'm not smart enough to write it myself'" (Jalender, Govardhan, & Premchand, 2010, p. 39; Nelson, 1996, p. 3).

Administrative impediments to reuse. Jalender, N.Gowtham, et al. (2010) state that it is hard to manage, i.e., catalog, archive and retrieve, reusable components "across multiple business units within large organizations" (p. 6138). The authors add that developers find it hard to find reusable components outside of their "immediate workgroups" (p. 6138).

The challenge of *Component repository maintenance.* Tiwari and Kumar (2020) state that the maintenance of a component repository is a challenge in CBSE. The following aspects require attention:

– Process of identification of useful and outdated components.

– Process of publishing new components in a repository and elimination of outdated components.

– Process of updating component information.

– Proper component versioning, i.e., supporting different versions and version update of the same components. Szyperski (2002) argues that there is a need to address the problem of proper component versioning to ensure component compatibility.

3.3.1 Software component

The concept of a software component is the core element of CBSE. According to Tiwari and Kumar (2020), the Oxford Advanced Learner's Dictionary defines a component as "one of several parts of which something is made" (Hornby, 1995), while Merriam-Webster dictionary defines it as "a constituent part"(Merriam-Webster, 2019). Tiwari and Kumar (2020) view component as an essential building block or unit in CBSE and present some leading definitions of a software component by various researchers (Tiwari & Kumar, 2020, p. 27-28). I have analyzed these definitions and extracted key component aspects that these definitions comprise: independence/separation, composability, deployment, modularity, interfaces, service/function, encapsulation, deliverability, and replaceability. Based on this, I have chosen three definitions that cover all these key aspects (Table 3.1).

#	**Definition**	**Source**
1	Szyperski: "A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed separately and is subject to composition by third parties"	(Szyperski, Gruntz, & Murer, 2011, p. 41)
2	OMG: "A component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces"	OMG as cited in Tiwari and Kumar (2020, p. 27)
3	Brown: "A software component is an independently deliverable piece of functionality providing access to its services through interfaces"	Brown (2000, Chapter 4.4, para.3)

Table 3.1: Leading definitions of software components in research.
Note: Adapted from *Component-Based Software Engineering: Methods and Metrics*, (p.27) by Tiwari and Kumar (2020). Taylor & Francis Ltd. Copyright 2020 by CRC Press.

The essential properties of a component, namely *interfaces*, *services*, and *deployment techniques*, should be considered when designing and developing software components (Kotonya et al., 2003; Lau, 2018; Tiwari & Kumar, 2020). Interfaces define the way software components communicate with the other parts of the system. They are used for passing data, instructions, and control sequence between the components and affect the degree of coupling in the system (Kotonya et al., 2003; Tiwari & Kumar, 2020). According to Kotonya et al. (2003), a component has two kinds of interfaces: required interfaces that are needed for a component to properly function and provided interfaces that the component realizes for interaction with other components. A software component is developed to serve some specific purpose that serves as a selection criterion when the component would be chosen for reuse in a different system or application. Therefore, a component should provide "some desired and defined service (set of services) or functionality (set of functionalities)" (Tiwari & Kumar, 2020, p. 29). Kotonya et al. (2003) define a service as "an abstraction of a set of functions that is designed to achieve some logical purpose" (p. 3). Kotonya et al. (2003) provide an example of a printing service that provides functionality related to printing that can be offered through a set of interfaces, such as printing files, installing a new printer device and etc. Tiwari and Kumar (2020) conclude that services must have a well-defined purpose, be robust and reliable, and be efficient with regard to performance and adaptability. Software components are deployable units in CBSE, therefore deployment techniques must be in place which specifies how a component is deployed into a component framework (Crnkovic, Hnich, Jonsson, & Kiziltan, 2002, p. 38; Szyperski, 2002, p. 41; Tiwari & Kumar, 2020, p. 30).

3.3.2 Component-based software engineering for reuse process

The component-based software engineering for reuse process focuses on two goals: development of reusable components and taking necessary steps to make them available for further reuse (Kotonya et al., 2003; Lau, 2018; Sommerville, 2011; Tiwari & Kumar, 2020).

Reusable components usually emerge during the development of some applications. CBSE for reuse requires effective mechanisms for component "harvesting" (Kotonya et al., 2003, p. 4), a process that involves analysis of software applications to determine whether some parts of the implementation could be separated into independent, standalone modules suitable for further reuse (Kotonya et al., 2003). After the extraction of these modules, it usually necessary to go through a process of generification, removing "application-specific features and interfaces" (Tiwari & Kumar, 2020, p. 464) that might not be necessary for another context, increasing component's reusability.

When all the necessary work in terms of generification has been done, a software component should be prepared to be moved to a component storage.

Several authors use the term "component repository" to refer to such storage unit (Crnkovic & Larsson, 2002, p. 321; Kotonya et al., 2003, p. 4; Lau, 2004, p. 120; Tiwari & Kumar, 2020, p. 30). Some component-based development models define the process of submission of components to a storage unit as publishing or archiving (Crnkovic & Larsson, 2002, p. 233; Tiwari & Kumar, 2020, p. 38, 45). Component repository contains both the components and their metadata, component interfaces, and other necessary information for proper component cataloguing and further reuse (Sommerville, 2011, p. 461; Tiwari & Kumar, 2020, p. 30). Therefore, component providers have to make sure they provide this information before component publishing.

3.3.3 Component-based software engineering with reuse process

Component-based software engineering with reuse involves the use of existing components to build software. This process's main activities are acquiring reusable components through the process of component acquisition and integrating them with other components to construct a software system. The process of CBSE with reuse starts with the development of user requirements (Sommerville, 2011). Sommerville (2011) recommends developing a complete set of requirements for a system while maintaining a high degree of flexibility because it makes it easier to identify as many components for reuse as possible. The next step is the identification of candidate components that meet the developed requirements. In some cases, the initial requirements require modification depending on the components that are available for reuse, but this approach is considered radical (Christiansson, 2002). After that, one can proceed to the architectural design phase. Further, one can repeat the activity of identifying candidate components because some of the previously identified components may become unfit for the system.

Similarly to Sommerville (2011), Crnkovic and Larsson (2002) also divide the development process into several stages, including system requirements specification, identification of initial components among the component candidates, performing architectural analysis, and a repeating step to check whether selected components are a good fit for a system. Moreover, component users should also assess whether candidate components should be adapted, or new components should be developed (Christiansson, 2002).

In this project, we only focus on the process of component acquisition, as our artifact provides the functionality for finding reusable components and does not cover the process of building web applications using these components. Thus, I decided to discuss the process of component acquisition separately in Section 3.3.4.

3.3.4 Component acquisition

The component acquisition process involves the acquisition of reusable components for further reuse in the CBSE with reuse process. The first step, component search, includes the identification of candidate components that could satisfy the established set of requirements. Component users would first look at existing in-house components, then at third-party components that are already being used or third-party components that are not being used (Sommerville, 2011). According to Sommerville (2011), third-party components from trusted sources would usually be preferred as this would make the component validation process easier. Some software companies create and manage their own component repositories and exclusively use these components to avoid components developed by external component providers (Sommerville, 2011, p. 466). The second step, component selection, involves selecting a component from the candidates identified during the component search process (Sommerville, 2011). Tiwari and Kumar (2020, p. 31) note that developers tend to select the components based on the requirements without considering other important factors like the degree of reusability, component testing efforts, the scope of maintenance, and scalability. Lau (2004) states that level of component granularity in a component repository impacts component selection because component users essentially look for components of a similar size to the problem being solved. The third step, component validation, involves validating that a selected component fulfills the original requirements and is suitable in the system being developed (Crnkovic & Larsson, 2002; Sommerville, 2011). Component validation usually involves testing the component in some way. Ideally, the component should be tested integrated with other components that it will interact with to ensure there are no undesirable emergent properties from such integration.

3.3.5 Component management and repository

Overcoming a challenge of storage and retrieval of reusable software components is essential to successfully adopting the CBSE approach. Sommerville (2011) indicates that a component repository is part of the component management process and suggests that components may be stored in a component repository that includes both the components and their specification. Kotonya et al. (2003) argue that there is a need for a "centralized management of reusable, shareable components" (p.4) and "establishment of a repository for maintaining the information about available reusable components" (p. 4). "Component management process is concerned with management (...) of reusable components, ensuring they are properly cataloged, stored, and available for reuse" (Sommerville, 2011, p. 461). Tiwari and Kumar (2020) define component repository as a database that contains the component and its metadata, component interfaces, and other necessary information. Crnkovic and Larsson (2002) state that a component repository stores components and their descriptions, including component versioning, and allows component acquisition. Lau (2018) also states the necessity of a component repository and argues that the components should be developed, cataloged, and stored in such a way that allows further retrieval, e.g., as a binary or as a source code. Tiwari and Kumar (2020) claim that component repositories require maintenance in order to ensure their consistency and efficiency, and currently, repository maintenance is one of the challenges in CBSE.

3.3.6 Component certification

The component certification process is part of component management and focuses on certifying that components have specific attributes or qualities. In particular, trustworthiness, specifically in that it has no malicious code, complies with specification and standards, that it has accurate user and design documentation, specific performance characteristics, and no defects. Heineman and Councill (2001) explain that if components placed in a component repository lack a proper description and do not comply with their specification, it results in a decrease of component trustworthiness and their extensive reuse. Furthermore, the availability of a component's source code would improve the situation, because component user would be able to inspect component's source code before taking it in use. However, if a component provider changes the source code of a component without documenting these changes in a component specification, it might cause side effects. Heineman and Councill (2001) claim that introducing component certification can reduce the risks associated with untrustworthy components. Sommerville (2011), Crnkovic and Larsson (2002), and Jalender, Govardhan, and Premchand (2010) agree on that components need to be trustworthy to make people more comfortable with software reuse.

ISO/IEC and IEEE (2017) defines *certification* as a "formal demonstration that a system or component complies with its specified requirements and is acceptable for operational use" (p. 63), and the person or organization performing this process is referred to as certifier. Heineman and Councill (2001) suggest that such formal demonstration could be "a signature on a test summary form or report, a mark applied to the [component], or a certificate that accompanies the [component]" (p. 701). Tiwari and Kumar (2020) indicate the importance of the component quality assessment, which includes identification and definition of quality attributes, which is "a broad area with a good deal of scope" (p. 94), and definition of quality metrics. Crnkovic and Larsson (2002) note that there is "a belief that certification means absolute trustworthiness, it in fact merely provides the results of tests performed and a description of the environment in which the tests were performed" (p. xxxii).

Heineman and Councill (2001) and Sommerville (2011) point out the importance of independent assessors, meaning that a component provider cannot certify his own components. Previous research shows that component certification is a major challenge in CBSE, and there is a clear lack of standard procedures and methods to certify component's validity and trustworthiness (Crnkovic & Larsson, 2002, p. xxxii; Jalender, Govardhan, & Premchand, 2010, p. 39; Mohammad, 2011, p. 135; Sommerville, 2011, p. 455; Tiwari & Kumar, 2020, p. 95). One of the challenges pertaining to certification is the role of component certifier (Jalender, Govardhan, & Premchand, 2010). Sommerville (2011) wonders who's responsibility it would be, if the component wouldn't operate as certified - would one blame component provider or the component certifier? Mohammad (2011) claims that component certification is a challenging activity and suggests further research that can address the question of certifier's trustworthiness, ways to ensure that the component has not been modifier after certification, and how component modification affects certification.

3.3.7 Component-based software engineering processes

To summarize the Section on CBSE, I present an overview of the CBSE processes and roles. The CBSE processes:

The *CBSE for reuse* process involves the development of components with the aim that others reuse these components. A person performing this process is referred to as a component provider.

The *CBSE with reuse* process involves the use of existing components (i.e., reuse) to build software. A person performing this process is referred to as a component user.

The *Component acquisition* process involves the acquisition of reusable components for further reuse in the component-based software engineering (CBSE) with reuse process.

The *Component management* process involves various management activities for components, such as ensuring proper cataloging and storage of the components. Components may be stored in a component repository.

The *Component certification* process involves certifying that components have some specific attributes or qualities, such as trustworthiness.

The CBSE roles:

Component provider is a developer that creates reusable components for further reuse and is engaged in the CBSE for reuse process.

Component user is a developer that uses reusable components and is engaged in the CBSE with reuse process and the process of component acquisition.

Certifier is an independent third-party (a person or an organization) that performs certification and is engaged in the component certification process.

Chapter 4

Kernel theories

Kernel theories are defined as "well-established theories in the natural and social sciences, which may exert some influence in the design process and should be considered by the researcher" (Dresch, Lacerda, & Antunes Jr, 2015, p. 78). In Design Science Research methodology, kernel theories are used in conjunction with the main design theory, which is CBSE in my case, and influence the design and development of an artifact. In my research, I introduce two kernel theories - installed base cultivation and software modularity. In Section 4.1.2, I describe and motivate the installed base cultivation strategy that we used to increase the likelihood of the artifact's adoption. In Section 4.2.2 on software modularity, I present and motivate a modular approach to software development that was used in our project as part of the installed base cultivation strategy.

4.1 Installed base cultivation

The theory on installed base cultivation proposed by Hanseth and Lyytinen (2010) is a design approach for the development of complex IT systems, namely *Information Infrastructures (II)*. In my research, I view DHIS2 as a software platform; however, it is also possible to look at it from the *information infrastructure* perspective. Hanseth and Bygstad (2018) explain that platforms are discussed based on their "*architecture/governance configuration*" (p. 3), i.e., a stable, extensible core governed by a single owner and peripheral complementing apps created by a large number of third-party developers. Hanseth and Lyytinen (2010) define II as "a shared, open (and unbounded), heterogeneous and evolving socio-technical system (which we call installed base) consisting of a set of IT capabilities and their user, operations and design communities" (p. 4). According to Hanseth and Bygstad (2018), information infrastructures are complex socio-technical systems characterized by a large number of vendors and heterogeneous users. However, II lacks centralized control, which is not the case for software platforms.

I will now explain how DHIS2 fits the definition of II proposed by Hanseth and Lyytinen (2010), which will enable the analysis and interpretation of DHIS2 from the II theoretical lens. First, DHIS2 is *shared* by multiple actors in the community, such as ministries of health, universities, developers, end-users, and various non-governmental organizations. DHIS2 is also *open*, meaning that "new components can be added and integrated with them in unexpected ways and contexts" (Hanseth & Lyytinen, 2010, p. 4) and "there are no clear boundaries between those that can use an II and those that cannot" (Hanseth & Lyytinen, 2010, p. 4). DHIS2 is open-source software, and its code can be found on GitHub, meaning it is fully open for inspection and forking, allowing developers to copy the source code and engage in independent development. There are no limitations on who can use DHIS2 and develop extensions for it. Hanseth and Lyytinen (2010) explain that II are heterogeneous, as they consist of different components "of very different nature" (Hanseth & Lyytinen, 2010, p. 4). DHIS2 is a *heterogeneous* socio-technical system as it includes multiple actors I previously mentioned and various technical components. Further, as a platform that is generative in nature, DHIS2 is evolving by virtue of being extensible, i.e., providing development capabilities to the other actors in the ecosystem.

Regarding control and governance, DHIS2's ecosystem is different from centralized digital platform ecosystems controlled by a single entity. Hein et al. (2020) explain that in software platform ecosystems like Facebook and SAP Cloud Platform, the power is centralized, and the platform owner is a single governance authority. In the case of DHIS2, while the platform core is controlled and governed by HISP UiO, their governance does not extend to the ecosystem as a whole due to in-country ownership, control, and governance of DHIS2 instances.

4.1.1 Motivation for Installed base cultivation strategy

In this section, I motivate the relevance of this strategy to the context of this project. Our practical research aim was to design and develop a component repository to be used as a development boundary resource to support web application development in the HISP community. First, while the DHIS2 core team maintains centralized control of the DHIS2 platform core, there is a lack of centralized control over web application development. Therefore, this complex setting only allows for a change in an evolutionary manner, a bottom-up approach considering the existing infrastructure already used by web application developers. Second, a component repository is an entity that does not exist on its own. It is a fundamental part of the CBSE approach, meaning that the repository would be used by the developers who engage in CBSE and consequently integrated into their component reuse practices and technologies, which constitute the installed base. Given that the practical problem of this study was to build a component repository, I assume the existence of some level of CBSE in the HISP community. Therefore, such a repository should not be built from scratch but instead built on top of what already exists and is already used by developers, or stated differently, built upon the installed base.

4.1.2 Installed base cultivation

Hanseth and Lyytinen (2010) argue that the present IT systems are complex, thus requiring new design approaches that can guide the development of these complex IT systems, referred to as information infrastructures. Nielsen (2006) explains that IIs are complex in a number of ways - they are open and heterogeneous, and this, combined with the fact that they consist of numerous diverse components, make them "inherently uncontrollable" (p. 1); expansion brings new interdependencies to the II. Additionally, they are often developed in a distributed fashion. Gare (2010) and Nielsen (2006) argue that

> IIs do not develop due to planned and controlled actions by some developers, but rather in a process imbued with surprises, blockages, diversions, side effects, and vicious circles, as well as inherent tensions between the need for universal standards and locally situated practices. (Gare, 2010, p. 28; Nielsen, 2006, p. 2)

To address the complexity of II, Hanseth and Lyytinen (2010) propose a design theory with a set of design principles and rules to guide the development of complex IT systems. As opposed to the 'design from scratch' strategy, their design theory focuses on the installed base cultivation, i.e., building on what already exists, for example, end-users and their practices, standards, regulations, and technology (Hanseth & Lyytinen, 2010, p. 15). The theory raises two problems which must be addressed - *bootstrap problem* and *adaptability problem*. With regard to *bootstrap problem*, Hanseth and Lyytinen (2010) state that a large number of users increase II's value; therefore the designers of II should develop solutions that drive the adoption when the user base is small or does not exist. This means addressing the needs of the existing user base before "addressing completeness of (...) design and scalability" (Hanseth & Lyytinen, 2010, p. 2). When the II is growing, and its user base is expanding, designers should be able to continuously improve the existing design based on the users' demands and needs. This requires technical and social flexibility of II and is defined as *adaptability problem* (Hanseth & Lyytinen, 2010, p. 2).

To address the *bootstrap problem*, Hanseth and Lyytinen (2010) encourage designers to follow these design principles (Hanseth & Lyytinen, 2010, p. 9):

1. Design initially for usefulness
2. Build upon the existing installed base
3. Expand the installed base by persuasive tactics to gain momentum

According to Hanseth and Lyytinen (2010), designing directly for usefulness implies focusing on a small group of users and developing a solution that can be beneficial to this user group. To gain the first adopters, designers should favor simple solutions that are easy to design, develop, and learn. Further, building upon the installed base implies utilizing the existing infrastructure that is in use by the targeted user groups, as this approach decreases adoption barriers and learning costs (Hanseth & Lyytinen, 2010, p. 10). With regard to the third design rule of installed base expansion, Hanseth and Lyytinen (2010) claim the importance of user base growth to generate positive network effects, while the development of the new functionality should happen only when it is necessary.

In order to address the *adaptability problem*, Hanseth and Lyytinen (2010) encourage designers to follow these design principles (Hanseth & Lyytinen, 2010, p. 9):

1. Make the organization of IT capabilities simple
2. Modularize the II

According to Hanseth and Lyytinen (2010), the first principle *Make the organization of IT capabilities simple*, encourages designers to keep their architectural designs simple and reduce their technical complexity, e.g., by having fewer technical elements and interconnections between them, adopting encapsulation, and using simple protocols. Further, the authors encourage designers to develop "loosely coupled sub-infastructures" (Hanseth & Lyytinen, 2010, p. 14), thus embracing modular design to accommodate the growth of II. The modular approach allows achieving II design stability, by "localizing the change and permitting fast and deep change in parts of the system" (Hanseth & Lyytinen, 2010, p. 7). To summarize, the proposed theory offers a bottom-up approach to design, focusing on installed base utilization.

4.2 Software modularity

4.2.1 Motivation for Software modularization

Section 4.2.2 explains how one can use an approach called modularization to achieve a high degree of software modularity as a way to reduce the system's complexity and increase its maintainability. Modularity is chosen as a kernel theory because this approach is part of the installed base cultivation strategy to address the adaptability problem. The architecture of a component repository should be built to allow improvements to the existing design based on users' demands and needs. Additionally, this approach was beneficial to us as developers and is expected to be useful for someone who will eventually take over this project and continue developing and maintaining it.

4.2.2 Software modularity

Designing and developing a high-quality software system is a complex and demanding task. ISO/IEC 25010 (2011) standard defines *quality* of a system as "the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value" (p. 358). The product quality model defined by ISO/IEC 25010 (2011) consists of eight quality characteristics, and one of them is *maintainability*. According to ISO/IEC 25010 (2011), maintainability "represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements" (p. 258). This characteristic is important, as software is inherently designed, developed, and maintained by people. Even if the software has a high degree of functional suitability, i.e., meeting the needs and requirements of its users, poorly designed architecture increases software complexity, making it difficult for its creators and maintainers to understand and predict the system's behavior.

One way of reducing software complexity and increase its degree of maintainability is by adopting *modularization*. Modularization is a software design approach that implies dividing a software system into multiple independent modules with well-defined interfaces that describe interactions between the modules (Bourque, Fairley, & IEEE Computer Society, 2014, p. 2-3). This approach allows achieving a high degree of software *modularity*, which is, according to ISO/IEC and IEEE (2017), defined as "a degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components" (p. 279).

The benefit of composing software from simple and independent modules has been known since the 1960s when L. Constantine first presented the idea of software modularization in his paper "Structured Design", written in collaboration with W. Stevens and G. Meyers (Stevens, Myers, & Constantine, 1974). The authors observed that software composed of simple and independent modules was "easiest to implement and change" (p. 116) because "problem solving is faster and easier when the problem can be subdivided into pieces which can be considered separately" (p. 116). The modules used to construct modular systems still need to be connected. Having fewer and simpler connections between modules brings several benefits to the overall system. It reduces the impact that changing one module has on other modules. For example, changing one module would not require you to change other modules. It prevents errors from propagating from one module to other modules; for example, if one module is broken, it would not affect the other modules. Moreover, it makes it simpler to refactor or change the system. For example, if we have one module that many other modules depend on, it would be hard to remove it from the system.

The number of connections between the modules and the degree to which each connection associates two modules impacts the software's complexity.

The degree to which modules are associated with each other is generally referred to as *coupling*. Stevens et al. (1974) define coupling as "the measure of the strength of association established by a connection from one module to another" (p. 117). Eder and Schrefl (1995) define coupling as "a measure of the interdependencies between different modules" (p. 2) and state that a well-designed system has to have loose coupling (also referred to as low coupling). Loose coupling implies a simpler or weaker association between two modules and is considered preferable, while the tight coupling is not (Booch et al., 2001; Hunt & Thomas, 2000; Ingeno, 2018; Tiwana, 2014).

With regard to software design, *cohesion* is defined as "a measure of the strength of association of the elements within a module" (ISO/IEC & IEEE, 2017), or, simply put, "the degree to which the elements inside a module belong together (...), a qualitative measure of consistency of purpose within a module" (Ingeno, 2018, p. 172). Since cohesion is a degree and not a binary value, there are different degrees of cohesion; these different degrees of cohesion also correspond to different types of cohesion (Ingeno, 2018). As opposed to coupling, a higher degree of cohesion is preferable (Eder & Schrefl, 1995; Hunt & Thomas, 2000; Ingeno, 2018; Stevens et al., 1974).

Ingeno (2018) discusses various principles and practices that software architects can use to design high-quality software applications. The author uses the concept of *orthogonal software* to refer to well-designed software comprised of independent modules and states that the modules in orthogonal software have two key attributes - loose coupling and high cohesion (Ingeno, 2018, p. 168). The concept of *orthogonality* originally comes from geometry, where two Euclidean vectors are considered orthogonal if they meet at the right angle without intersection, thus being independent of each other (Hunt & Thomas, 2000; Ingeno, 2018). Hunt and Thomas (2000) state that in computing, the concept of orthogonality came to "signify a kind of independence and decoupling" (p. 57).

There are several ways to achieve loose coupling between software modules. When implementing a change in one of the modules, one can assess how localized this change is, i.e., determining whether this change would affect only this specific module, or would there be an impact on the entire system (Hunt & Thomas, 2000). If a change is not localized and affects the rest of the system, it is an indication of tight coupling, and then the reasons behind it should be analyzed to determine what changes in the system's design should be done to eliminate this issue. Therefore, continuous assessment of the system could help to achieve loose coupling. Law of Demeter, or "principle of least knowledge," is a design method to achieve loosely coupled modules in a system (Ingeno, 2018, p. 171). I. Holland proposed this guideline in 1978, and its goal "is to organize and reduce dependencies between [modules]" and the achievement of high modularity (Lieberherr & Holland, 1989). The main idea behind Law of Demeter is that a module should only know and communicate with as few other modules as possible, and only when such knowledge or communication is essential (Ingeno, 2018; Lieberherr & Holland, 1989). According to Hunt and Thomas

(2000), following the guidelines of Law of Demeter will make your system adaptable and robust, as well as having fewer errors.

As mentioned before, there are various types of cohesion, and each of them representing a different degree, from lowest and less desirable to highest and most desirable. High cohesion allows reusability of the module, increases its maintainability and testability (Ingeno, 2018). According to Ingeno (2018), to achieve high cohesion, one should make sure that each module in the system has its own single purpose and that the elements within that module should all contribute to it. Elements that do not fulfill these criteria should either be moved to a new module or to one of the other existing modules that serve the same purpose (Ingeno, 2018). An example of cohesive software design that is a system composed of high cohesion modules could be a system for predicting future energy consumption with neural networks. Such a system would have several independent modules, one for data pre-processing, one for building a training model, and so forth. A similar system with a low degree of cohesion within modules would combine the functionality of pre-processing and for building a data training model in one module.

In this section, I have described modularization, which aims at reducing software complexity and increasing its maintainability. I have also outlined the concepts of coupling and cohesion and the importance of low coupling and high cohesion in modular systems. A summary of the most important concepts introduced in this chapter can be seen in Table 4.1.

Concept	**Definition**
Quality	"The degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value" (ISO/IEC 25010, 2011, p. 358)
Maintainability	"The degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements" (ISO/IEC 25010, 2011, p. 258)
Modularization	A software design approach that implies dividing a software system into multiple independent modules with well-defined interfaces that describe interactions between the modules (Bourque et al., 2014)
Modularity	"Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components" (ISO/IEC & IEEE, 2017, p. 279)
Coupling	"The measure of the strength of association established by a connection from one module to another" (Stevens et al., 1974, p. 117)
Cohesion	"The degree to which the elements inside a module belong together (...), a qualitative measure of consistency of purpose within a module" (Ingeno, 2018, p. 172)
Orthogonal software	Well-designed software comprised of independent modules (Ingeno, 2018, p. 168)

Table 4.1: Summary of the concepts with regard to software modularity.

Chapter 5

Research approach

This chapter introduces the research paradigm of this study and elaborates on its ontological, epistemological, and methodological assumptions. In Section 5.2, I provide a general description of DSR methodology, and in Section 5.3, I explain how it was applied in our project. I also discuss research methods used for data collection and analysis, while in Section 5.7 I elaborate on the artifact's evaluation activity.

5.1 Philosophical foundation

A researcher needs to understand the nature of the reality being studied and various means to obtain knowledge about it. Moon and Blackman (2014) explain that to be able to understand the nature of knowledge and ways to acquire such knowledge, it is "necessary to understand the principles and assumptions of scientific research, in other words, philosophy" (p. 1168). Two branches of philosophy are important in natural and social sciences: *ontology*, which focuses on the study of being, and *epistemology*, which focuses on the study of knowledge (Moon & Blackman, 2014). Ontology and epistemology are ultimately interconnected and create a holistic view of knowledge, the researcher's relation to it, and the methodological approach to developing it. A research paradigm consists of ontology, epistemology, and methodology and is considered to be a model and an approach to research that gives a means to understand reality and study it (Rehman & Alharthi, 2016; Schwandt, 2001).

I consider the nature of my research problem and my research aim to be key factors to establishing my ontological and epistemological positions. My research aim was to attain two aims - a practical one and a theoretical one. The practical aim involved designing and developing a component repository that facilitates component reuse in a software platform ecosystem. The theoretical aim was to create abstract knowledge valuable for other researchers and developers - a set of design principles on how a component repository should be developed in a software platform ecosystem. To achieve these aims, I have established the following research question: *What are the* *essential design principles for implementing a component repository that facilitates component reuse in a software platform ecosystem?* In order to attain these aims and answer my research question, my choice was to gain an understanding of the software reuse practices in the HISP community and challenges pertaining to them and then use the obtained knowledge to design and develop an artifact that could improve the current software reuse process and practices. I view software reuse as a socio-technical activity, as it clearly has some social aspects in addition to technical aspects and is heavily impacted by people and their practices. For example, the metadata specification for a component is highly technical, as it must be exactly specified and machine-readable to ensure proper component cataloging in a component repository. There are, however, also social aspects, for example, the developers' attitude towards software reuse, which could be influenced by social factors such as trust and understanding. Even if a component repository is in place, developers might be reluctant to use others' components if there are no reliable methods to ensure component trustworthiness. Using qualitative methods for learning about software reuse practices in the HISP community would allow me to deepen my knowledge on the topic and explore problems, feelings, thoughts, and meanings of people engaging in software reuse activities. This would be in line with the interpretivist approach, which, according to Crotty (1998), "looks for culturally derived and historically situated interpretations of the social life-world" (p. 79). However, I could also see a clear application for quantitative methods to measure the utility of our artifact, for example, when measuring its accuracy, i.e., the degree of agreement between the artifact's expected and actual output. For example, if a user runs a command-line interface command to validate whether his `package.json` file is structured correctly, he would expect validation to report the errors found. This would be in line with the positivist approach that would, among other things, offer "assurance of unambiguous and accurate knowledge of the world" (Crotty, 1998, p. 27), which in this case would be testing that the artifact behaves as expected and is compliant with its technical requirements. Given that both approaches would be of value, I conclude that I, as a researcher, am open to any approach that shows its utility for solving my particular research problem. Therefore I reject the dichotomy of positivism and interpretivism and choose to adopt a pragmatic research paradigm.

The main idea of pragmatism, "a quintessentially American philosophy" (Crotty, 1998, p. 85), is to embrace the most beneficial approach in the circumstances. In some cases, researchers adopt practices that are positivist in nature, and in other cases, they adopt practices in line with interpretivism. Some purists advocate incompatibility of different paradigms and even treat them as religions, engaging in so-called 'paradigm wars' (Darlington & Scott, 2020; Rehman & Alharthi, 2016). Contrary to them, the pragmatists believe that the methods should be chosen "on the basis of their suitability to address specific research needs" (Darlington & Scott, 2020, p. 120) and should be linked directly to the research problem.

Choosing a pragmatic research paradigm has its implications on my research. Pragmatism is inherently a problem-solving paradigm and does not seek to address broader philosophical questions. The goal of pragmatism is to determine what can produce the most utility for solving a concrete problem, thus embracing any method that could help achieve this goal.

5.2 Research methodology: Design Science Research

The methodological framework chosen for this study is Design Science Research (DSR). In contrast to other methodologies that have a goal of understanding reality, DSR's focus is to "construct new and innovative ways to solve a class or classes of problems, thus creating new reality" (Iivari & Venable, 2009, p. 8). Dresch et al. (2015) view DSR as a problem-solving research method that can be used to construct new artifacts in order to change the situations to a better or more desirable state. Baskerville, Kaul, and Storey (2015) and Iivari and Venable (2009) emphasize the dualism of DSR and explain this dualism as DSR's two goals: to develop a solution applicable to generalization and to produce new knowledge beneficial to other practitioners and researchers. DSR's dualism presents in the research aim of this study, which was to develop an artifact, a component repository, and produce prescriptive knowledge which takes a form of design principles in this study. Thus, I argue that Design Science Research methodology can fully support the aim of this study and is appropriate for the targeted research question.

The DSR process model proposed by Peffers, Tuunanen, Rothenberger, and Chatterjee (2008) (Figure 5.1) is highly referenced compared to other existing process models (Brocke, Hevner, & Maedche, 2020). This process model is cyclical and includes the following activities: "problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication" (Brocke et al., 2020, p. 5). Additionally, it provides four entry points: "problem-centered initiation, objective-centered solution, design- and development-centered initiation, and client/context initiation" (Brocke et al., 2020, p. 6).

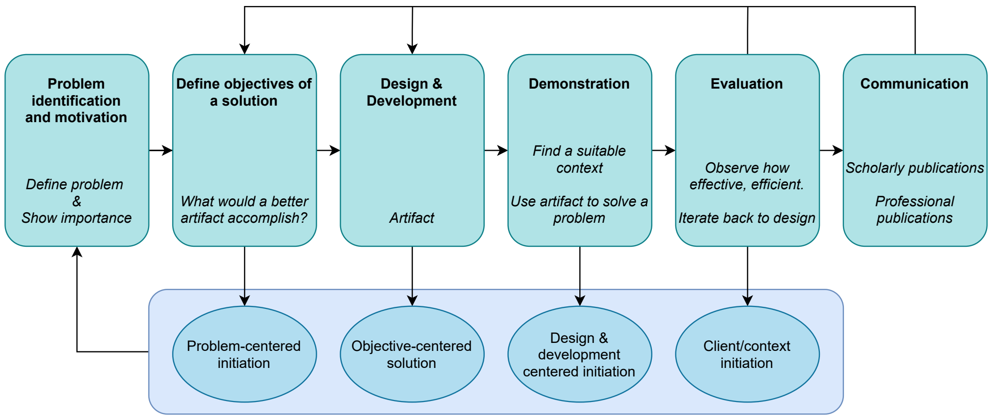

Problem identification and motivation
Define problem & Show importance
Define objectives of a solution
What would a better artifact accomplish?
Design & Development
Artifact
Demonstration
Find a suitable context
Use artifact to solve a problem
Evaluation
Observe how effective, efficient.
Iterate back to design
Communication
Scholarly publications
Professional publications
Problem-centered initiation
Objective-centered solution
Design & development centered initiation
Client/context initiation

Figure 5.1: DSR cycle diagram.
Note. Adapted from "A Design Science Research Methodology for Information Systems Research", by Peffers et al. (2008), *Journal of Management Information Systems*, 24:3, 45-77. Copyright 2008 by M.E. Sharpe, Inc.

During the *problem identification and motivation* activity, the research problem is identified and defined. According to Brocke et al. (2020), this activity requires such resources as "knowledge of the state of the problem and the importance of its solution" (p. 6). Regarding the pragmatic research paradigm, this activity is concerned with the knowledge of what-is, i.e., it aims to explore what already exists and is available for investigation (Goldkuhl, 2012). The goal of the second activity, *definition of objectives of a solution*, is to use the knowledge of what-is and transform it into the knowledge of to-be, or "*prospective knowledge* - knowledge about the possible" (Goldkuhl, 2012, p. 87). The third activity, *design and development* of the artifact, includes the development of functional and non-functional requirements of the artifact, its architectural design, and the actual implementation process. The goal of the fourth activity, *demonstration*, is to show the artifact's ability to solve the problem it was built to solve. To perform this activity, a researcher can take in use different methods considering the artifact's nature. The fifth activity, *evaluation*, measures the artifact's utility to solve the problem. A. R. Hevner, March, Park, and Ram (n.d.) and Venable, Pries-Heje, and Baskerville (2016) argue that this activity is an integral part of DSR and the researchers are expected to "demonstrate the utility, quality, and efficacy of a design artifact using well-executed evaluation methods" (p. 77). During the sixth activity, *communication*, the researchers communicate all the aspects of their work, i.e., the research problem and its importance, the artifact, its utility, etc. Since the DSR process model is cyclical, the researchers can iterate back to previous activities, as shown in Figure 5.1, and improve the artifact.

vom Brocke and Maedche (2019) explain that DSR has two types of knowledge - the input knowledge, i.e., prior knowledge that can guide the DSR project, and the output knowledge, i.e., a contribution in the form of prescriptive knowledge. vom Brocke and Maedche (2019) distinguish between three types of input knowledge: "*kernel theories*, *design theories*, and *design entities*" (vom Brocke & Maedche, 2019, p. 381). A *design theory* is defined as "a set of principles and knowledge that describes and guides the development of a design artifact to attain a specific goal in the material world" (vom Brocke & Maedche, 2019, p. 380). vom Brocke and Maedche (2019) define *design entities* as "design artifacts like constructs, models, methods and instantiations, design processes, and artifact evolution processes" (p. 380). Further, *kernel theories* are "well-established theories in the natural and social sciences, which may exert some influence in the design process and should be considered by the researcher" (Dresch et al., 2015, p. 78).

DSR's contribution to knowledge, as shown in Table 5.1, can be assessed using three maturity levels (Gregor & Hevner, 2013). Situated implementation of artifact is on the first maturity level as context-specific limited and less mature knowledge. Goldkuhl (2012) classifies it as *local functional pragmatism* and states, that the artifact is the core local contribution.

Level	**Contribution to knowledge**	**Example contribution**
3	Well-developed design theory about embedded phenomena	Design theories
2	Nascent design theory - knowledge as operational principles/architecture	Constructs, methods, models, design principles, technological rules
1	Situated implementation of artifact	Instantiations (software products or implemented processes)

Table 5.1: DSR contribution to knowledge.
Note. Adapted from "Positioning and Presenting Design Science Research for Maximum Impact", by Gregor and Hevner (2013),(p. 342) *MIS Quarterly*, June 2013, Vol. 37, No. 2 (June 2013), pp. 337-355. Copyright 2013 by Management Information Systems Research Center, University of Minnesota.

The second level of maturity includes more generalized and abstract knowledge in the form of constructs, methods, models, design principles, and technological rules (Gregor & Hevner, 2013). Goldkuhl (2012) argues that the first level of maturity is not enough for a scientific contribution, and there is a need for abstract, prescriptive knowledge, valuable beyond local practice. "In the perspective of general functional pragmatism, such knowledge should be useful for practitioners belonging to different practices" (Goldkuhl, 2012, p. 91). The third level of maturity includes "well-developed design theories about the phenomena under study" (Gregor & Hevner, 2013, p. 341). Goldkuhl (2012) states that Design Research, in general, is conformant with pragmatism and its philosophical foundation, as they both emphasize the importance of the utility of an artifact.

In this section, I have introduced the DSR methodology, explained the key activities it consists of, and their relation to pragmatism. Additionally, I have explained types of knowledge in DSR and the maturity levels of the output knowledge.

5.3 Research process

This section describes how the DSR research methodology was applied in our project.

The *problem identification and motivation* activity started at the end of February 2020. My team and I conducted a focus group discussion with the DHIS2 core team developers and one developer from HISP East Africa. Later, as part of the same activity, we conducted a focus group and two interviews with developers in HISP East Africa. The qualitative data gathered through these activities was transcribed, analyzed, and used to define the artifact's objectives and technical requirements. As seen in Figure 5.2, the second activity, *definition of objectives of a solution*, was happening simultaneously with the first activity, as the objectives and requirements were continuously revised as we obtained more data. In August, we started to work on the implementation of the artifact. My team and I had at least one meeting per week to discuss what needed to be implemented and how and planned various data gathering activities. Otherwise, we worked independently of each other, sometimes engaging in discussions on Slack. During the early stage of the *design and development* activity, I worked on the implementation of the website; however, later, my main responsibility was the development of the command life interface and the implementation of certification functionality. Although, I would still work on the website from time to time, mostly focusing on the UI.

During the *demonstration* activity, we conducted several focus groups with the DHIS2 core team to review what has been done and what still had to be implemented and improved. These demonstrations could also be viewed as an evaluation because the data gathered through these activities was analyzed and allowed us to iterate back to the *definition of objectives of a solution* activity. The DHIS2 core team mentioned several times that they would be interested in participating in the development activities, specifically taking necessary steps to implement certification functionality. However, none of this was done, presumably because of resource and time constraints on their part, and it was hard to establish contact with them. Therefore, after discussing this with Heskja, we decided to proceed with the implementation of the certification functionality on our own, and this task was appointed to me.

In early December 2020, all the development was rapidly stopped as it was time to move to the next DSR activity - *evaluation* of the artifact.

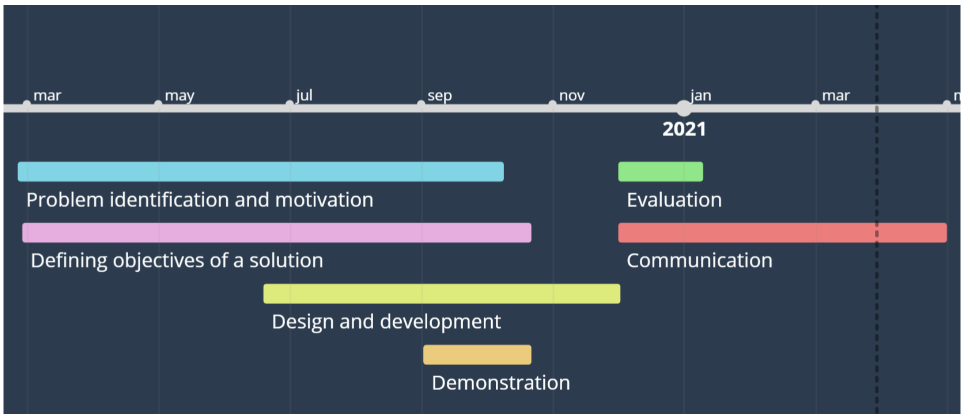


Figure 5.2: Research activities timeline.

I decided to proceed with this activity on my own because my colleagues and I had different evaluation aims and foci, and consequently, evaluation criteria and methods. Doing the artifact evaluation on my own has allowed me to be more flexible with regard to time management and decision-making, as I would not need to compromise with my colleagues and could tailor the evaluation activity to my research question. I will discuss the *evaluation* activity, the chosen criteria, methods, and participants in Section 5.7. Table 5.2 shows the list of conducted interviews and focus groups preceding the artifact *evaluation*.

Date	**Data gathering method**	**Participants**	**Number of partici-pants**
February 26, 2020	Focus group	The DHIS2 core team, HISP East Africa developer	5
July 8, 2020	Interview	HISP East Africa developer	1
August 26, 2020	Focus group	HISP East Africa developers	2
October 2, 2020	Focus group	The DHIS2 core team	2
October 9, 2020	Interview	HISP East Africa developers	4
October 28, 2020	Focus group	DHIS2 core team developer	1

Table 5.2: Data collection methods.

With regard to the maturity levels discussed in Section 5.2, my contribution to knowledge is on the second level of maturity. The artifact belongs to the first level of maturity, while the established design principles belong to the nascent design theory, and consequently, the second level of maturity.

5.4 Development of the design principles

As discussed in Section 5.2, there are three types of input knowledge in DSR: kernel theories, design theories, and design entities. My study is guided by CBSE, which is my design theory, and two kernel theories: modularity and installed base cultivation. The data gathered during the interviews and focus groups and our design and development work on the artifact served as the basis for establishing the design principles, which emerged during the first four activities of the DSR cycle (Figure 5.1). The design theory and kernel theories serve as a theoretical grounding for the design principles.

I introduce my design principles in Section 6.5, and provide empirical evidence to support each of them. Section 7.5 discusses the evaluation results of the application of the design principles it was possible to evaluate. In Section 8.1, I discuss the established set of design principles as a contribution of this research by considering the results from evaluation, literature review, and kernel theories.

5.5 Data collection

This section will discuss the goals and methods for data collection during the following DSR activities (Figure 5.1):

- Problem identification and motivation
- Definition of the objectives for a solution
- Design and development of the artifact
- Artifact's demonstration

and explain how the participants were selected. Data collection methods and goals for the research evaluation activity will be discussed separately in Section 5.7.

5.5.1 Goals

The goals of data collection in this study are ultimately linked with the study's aim. The first goal of data collection was to learn about software reuse practices and challenges, impediments to software reuse, and developers' attitudes towards reuse. This data helped us to identify the problem and define the objectives of our solution. The second goal was to gain feedback from our participants throughout the development and design activity that could further be used to improve the artifact.

5.5.2 Participants

Lopez and Whitehead (2013) explain that an effective sampling selection in qualitative research is important because it has an impact on the research findings and outcome. To establish the sampling method for participant selection of this study, we considered its aim. SCP's aim was to facilitate component reuse in web application development in the HISP community. Therefore, the participants of this study needed to be representative of SCP's user group, i.e., they would have to be engaged in web application development for DHIS2 and be willing to share their experiences with us. Given this, a non-probability *purposive sampling* strategy was adopted for participant selection. Lopez and Whitehead (2013) state that:

> purposive sampling is designed to provide information-rich cases for in-depth study. This is because participants are those who have the required status or experience or are known to possess special knowledge to provide the information researchers seek. (Lopez & Whitehead, 2013, p. 125)

Web application development for DHIS2 is primarily done by developers in HISP groups and the DHIS2 core team developers; therefore, they were our target groups for participant recruitment. We tried to establish contact with several HISP groups and inquire whether they would be interested in participating in this study. In total, five HISP groups were contacted, but only two agreed to participate. Some of our emails were left with no reply, some HISP groups replied but showed no further interest in participation. Developers from two HISP groups (three developers in total) in HISP East Africa agreed to participate. Additionally, two DHIS2 core team developers agreed to participate and collaborate. This allowed us to explore potential diversity in the interests of HISP groups' developers as platform complementors, and DHIS2 core team, as platform owners. The DHIS2 core team and the HISP East Africa developers are engaged in web application development; therefore, in our project, they could act as component providers and component users. Additionally, since it was the DHIS2 core team who suggested the implementation of component certification, they were expected to take the role of component certifiers. The sample size of five participants may be considered relatively small, but we consider it adequate given it was hard to recruit our participants, and our collaboration was solely digital due to our inability to travel and engage in fieldwork as a consequence of the COVID-19 pandemic. Additionally, even though the DHIS2 core team developers are based in Oslo, our collaboration was mainly digital due to COVID-19 regulations in Norway.

5.5.3 Interviews and focus groups

Semi-structured interviews and focus groups were used as the main methods for *direct data* gathering, which is defined as data that "include recordable spoken or written words and also observable body-language, actions and interactions" (Lopez & Whitehead, 2013, p. 127).

We have developed an interview guide (Appendix F) containing a set of questions to ensure that our learning goals are covered. Some examples of the questions are:

1. Is everything written from scratch, or do you reuse components?
2. What kind of components are you interested in reusing?
3. Where do you store and share components?

Semi-structured interviews have given us flexibility, as we were free to ask additional or follow-up questions where we felt it was necessary for the course of the interview and helped us ensure we adhere to the main topic. Lopez and Whitehead (2013) argue that interviews "provide the researcher with a valuable opportunity to enter the world of the participant and reflect on a particular event" (p. 130). We have gained valuable information we used to define the objectives of the solution. However, further analysis of the gathered data has shown that a large amount of the data was not relevant and thus not used in the study. In this case, I would attribute this to our inability to establish the relevance of our learning goals for the aim of our research early in the process. When discussing this issue with Heskja, he noted that "it is clear that we didn't know exactly what we were doing" (Heskja, personal communication, March 29, 2021). Although, this explorative approach was still useful, as it has given us what we needed in order to develop our artifact and our theoretical contributions.

For our first interview, we established that only one of us would act as an interviewer, while the two others would only listen and ask follow-up questions. We aimed to make our participants comfortable and not feeling as if they are being interrogated by the three of us. However, this strategy happened to be quite exhausting for the interviewer. Therefore for the following interview, we decided that all of us would act as interviewers, taking turns and covering different learning goals. I found our second interview to be a so-called panic-interview, as it was "intense and mentally exhausting" (Crang & Cook, 2007, p. 71). First, the participants were explaining their application development practices, and while doing so, they often touched upon software reuse practices and the topics that I was planning to inquire about when my turn to ask questions would come. To avoid repetitions, I had to fully focus on what was being said, constantly looking through the checklist with the interview questions and revising them when necessary. I conclude that it would perhaps be better to conduct a set of interviews with a more narrow focus rather than cover a relatively large set of learning goals in one interview.

Our team considered focus groups the most appropriate method for our interactions with the DHIS2 core team, as we mainly used their expertise to guide the artifact design and construction. Crang and Cook (2007) view focus groups as "groups of people [that] meet to discuss their experiences and thoughts about specific topics with the researcher and with each other" (Crang & Cook, 2007, p. 90). A. Hevner and Chatterjee (2010) argue that focus groups can be adapted to DSR methodology. A. Hevner and Chatterjee (2010) propose the use of focus groups in two situations: (1) incremental artifact improvement using *exploratory focus groups* and (2) demonstration of the artifact's utility in a field setting using *confirmatory focus groups*. In our case, we used the *exploratory focus groups*, as we either demonstrated or explained the current state of the artifact and then discussed various aspects that needed re-design and improvements. Additionally, we discussed the aspects that we were still to implement.

Participants of our focus groups were the DHIS2 core team developers who worked together on a daily basis; therefore, one can view them as an already-existing group. While it is often easier to recruit such groups, Crang and Cook (2007) recommend avoiding already-existing groups, because "with members already knowing each other, there may be personal dynamics at work that the researcher will not be aware of which can have a significant bearing on what is said and who says it"(Crang & Cook, 2007, p. 91).

The interviews and the focus groups were conducted online using Zoom, a conference room solution. Zoom's functionality allowed us to record the interviews for later transcription and analysis using the thematic analysis method. One of the disadvantages of video interviews is unstable connection and background noise, which affected the quality of the gathered data. In some cases with rather short inaudible segments, we looked at the syntactic and semantic context to guess what has been said; however, it was not always possible to identify missing words.

5.6 Data analysis

Data analysis is an essential part of the research process because if a researcher does not have a good understanding of the methods he used to analyze the data, it can undermine the trustworthiness of his study (Nowell, Norris, White, & Moules, 2017). This section will explain how I analyzed the qualitative data gathered throughout this study.

5.6.1 Thematic analysis

Nowell et al. (2017) state that thematic analysis is a method that guides the analysis of large qualitative data sets. The main idea behind this method is finding themes or patterns in the analyzed data that provide the answer to the research question.

Each interview we conducted during this study was guided by a set of learning goals or larger themes (Appendix E). These themes were not used for a deductive approach in thematic analysis; their purpose was only to ensure that we cover all the necessary topics during the interviews.

Therefore, when analyzing the transcribed data from the interviews and focus groups, an inductive approach was used. Nowell et al. (2017) explain that the inductive approach is data-driven, where "the themes identified are strongly linked to the data themselves and may bear little relation to the specific questions that were asked of the participants" (Nowell et al., 2017, p. 8). When conducting thematic analysis, I read through the transcribed data several times to become familiar with it. Subsequently, the sections of data with relevance to the study were identified, and an initial set of codes was developed and later used to identify larger themes. Figure 5.3 shows a global theme of *software reuse* identified during the analysis alongside its sub-themes and codes. Figure 5.4 shows some of the codes belonging to the *Component* organizing theme and some of their respective data sections.

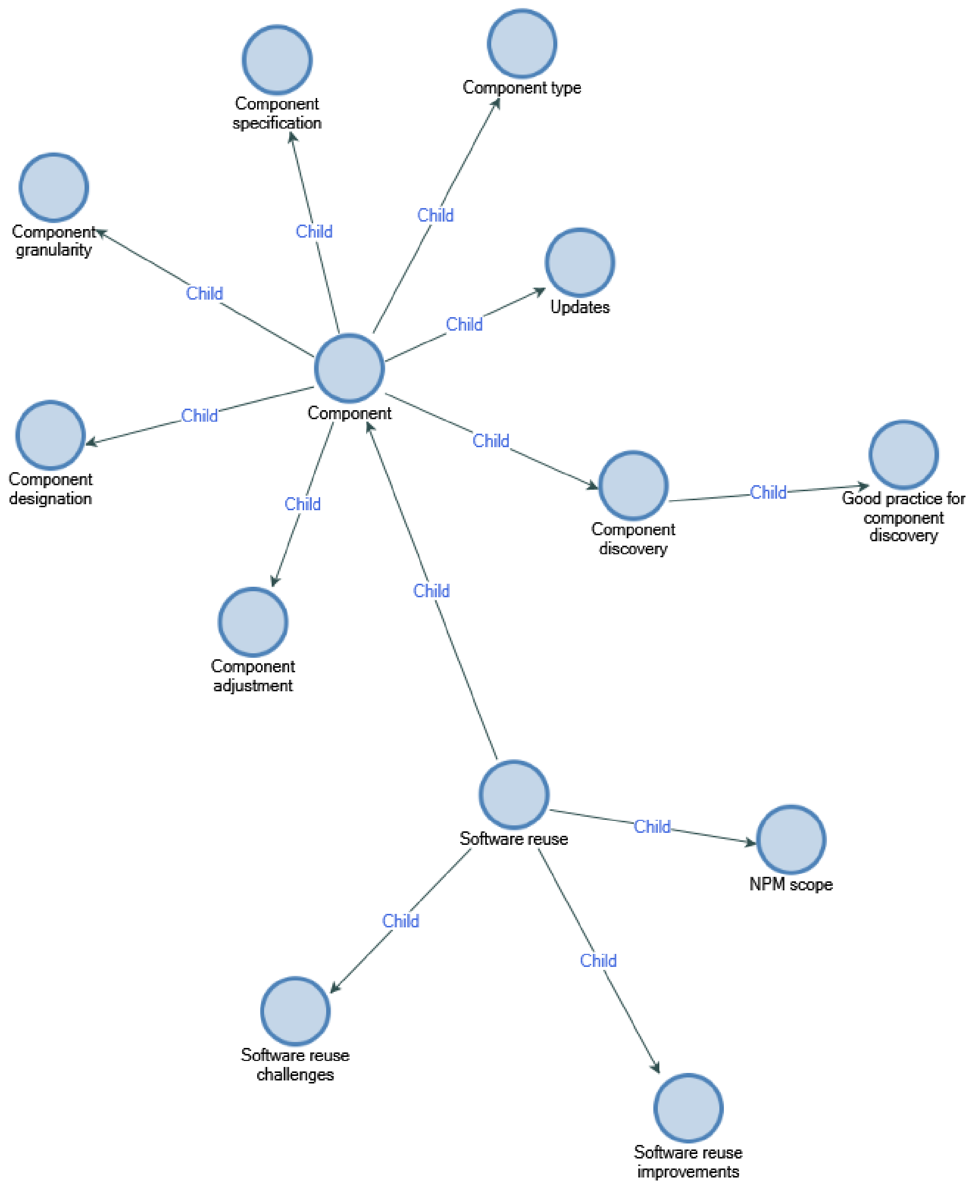


Figure 5.3: A thematic map showing the relationship between the codes and larger themes.

Thematic analysis was conducted using NVivo 12, a qualitative data analysis software package. Once the themes were identified, the data related to the themes most relevant to the SCP were used to guide the design and development of our solution.

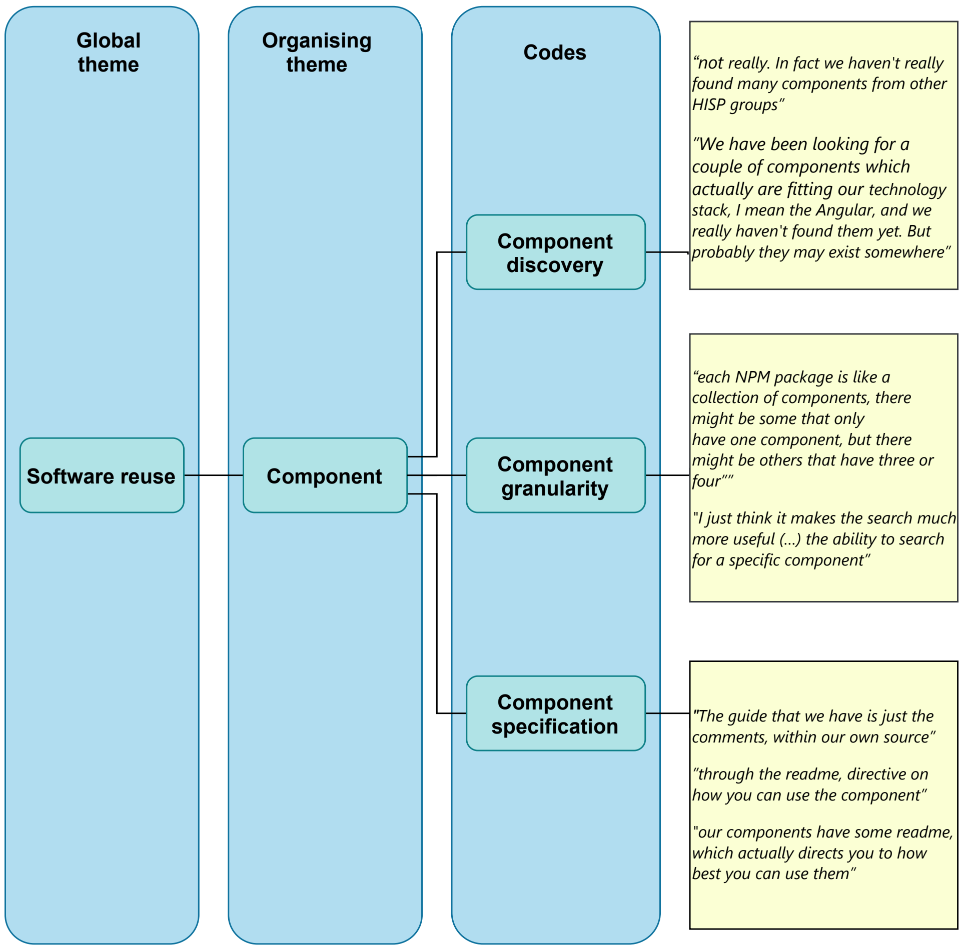

Global theme
Organising theme
Codes
Software reuse
Component
Component discovery
Component granularity
Component specification
“not really. In fact we haven't really found many components from other HISP groups”
“We have been looking for a couple of components which actually are fitting our technology stack, I mean the Angular, and we really haven't found them yet. But probably they may exist somewhere”
“each NPM package is like a collection of components, there might be some that only have one component, but there might be others that have three or four””
"I just think it makes the search much more useful (...) the ability to search for a specific component”
"The guide that we have is just the comments, within our own source”
“through the readme, directive on how you can use the component”
"our components have some readme, which actually directs you to how best you can use them”

Figure 5.4: Example of thematic analysis for the global theme Software reuse, its code and data sections.

5.7 Artifact evaluation

Evaluation of an artifact is an important activity of DSR that aims to provide feedback for further work on the artifact and establish how well the artifact achieves its utility (Venable et al., 2016). There are two important categories of the evaluation discussed in the literature, namely: "(1) *formative evaluation* [and] *summative evaluation* and (2) *ex ante* [and] *ex post* evaluation" (Venable et al., 2016, p. 78). The *summative* and *formative* evaluation is *Why* the evaluation is being done. According to Venable et al. (2016), the purpose of *formative* evaluation is to provide a basis for further improvements of the artifact, while the purpose of the *summative* evaluation is to evaluate the effectiveness and the utility of the artifact. *Ex ante* and *ex post* evaluation refer to the point *When* the evaluation is happening in the DSR cycle. *Ex ante* evaluation happens before the design and development activity in DSR and addresses the artifact's requirement planning. *Ex post* evaluation usually happens after the artifact construction in order to establish "the value of the implemented system" (Venable et al., 2016, p. 79). Venable et al. (2016) explain that *ex ante* and *ex post* evaluations are the two "extremes of an evaluation" (p. 79) activity, while the researchers can perform a set of intermediate evaluations during the artifact's construction (Figure 5.5).

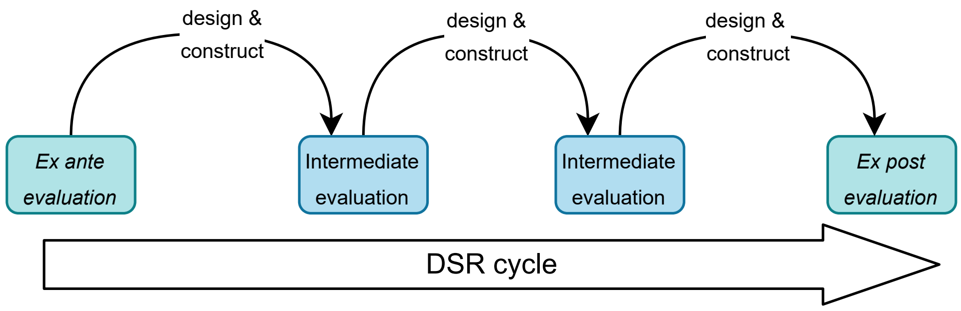


Figure 5.5: Evaluation continuum in DSR.
Note. Adapted from "FEDS: a Framework for Evaluation in Design Science Research", by Venable et al. (2016), *European Journal of Information Systems (2016) 25*, 77–89. 2016 by Operational Research Society Ltd.

Other important aspects of evaluation are *How* to evaluate, i.e., determining the rights methods for evaluation, and *What* to evaluate, i.e., determining the right evaluation criteria. A. Hevner, Prat, Comyn-Wattiau, and Akoka (2018) claim that in DSR, "the value of a design artifact is defined by the goodness of its fit as a solution for the problem or opportunity presented" (p. 3). A. Hevner et al. (2018) propose a hierarchy of DSR goals for socio-technical systems solutions, which is based on Maslow's (1943) hierarchy of human needs (Figure 5.6).

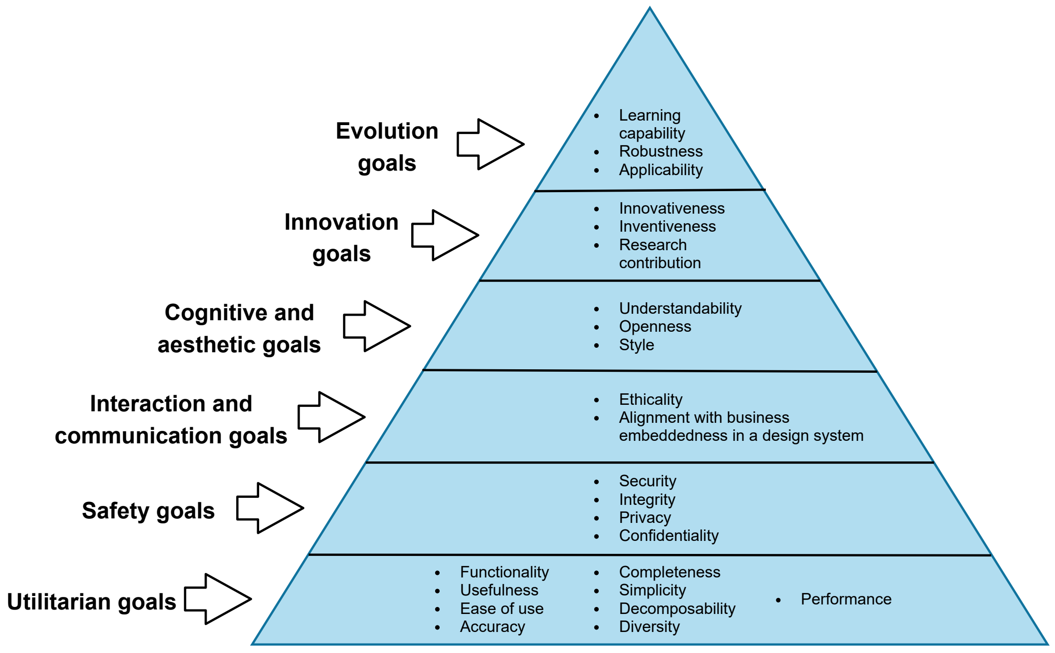


Figure 5.6: Hierarchy of DSR goals and evaluation criteria.
Note. Adapted from "A pragmatic approach for identifying and managing design science research goals and evaluation criteria", by A. Hevner et al. (2018), *AIS SIGPrag Pre-ICIS workshop on "Practice-based Design and Innovation of Digital Artifacts."*

Figure 5.6 shows that the hierarchy comprises six goals with their corresponding evaluation criteria. Utilitarian goals at the lowest position of this hierarchy focus on the artifact's functionality, utility, and benefits. A. Hevner et al. (2018) explain that one should be fulfilling the goals starting at the base of the hierarchy, i.e., the fulfillment of the utility of the artifact has the highest priority, then comes safety, etc. Similarly, when evaluating an artifact, it is most critical to evaluate the criteria associated with goals starting at the hierarchy base.

I establish my evaluation as a *formative* evaluation because SCP is still in the development stage and is not yet ready to be put in production. Therefore the goal of my evaluation, i.e., *Why* to evaluate, is to obtain feedback from the users that can help with further development and improvement of the artifact. With regard to the aspect of *When* to evaluate, I establish that my evaluation is intermediate. This is because it is neither an *ex ante* evaluation, as development has already been started, nor is it *ex post* evaluation, as development has not yet been completed either. To address the aspect of *What* to evaluate, I decided to include the criteria from the utilitarian dimension in Figure 5.6: accuracy, efficacy, usefulness, and performance. Additionally, my evaluation would also address the criterion of openness. Due to time, resource constraints, and difficulties finding the participants for my evaluation, I tailored my evaluation according to the resources available. A complete set of the established evaluation criteria is shown in Table 5.3. The expert evaluation was used to evaluate the application of one of my design principles, and I figured out that it was also possible to address the criteria of openness and performance as part of the expert evaluation. Criteria of usefulness and efficacy are, perhaps, one of the most important criteria in DSR, as it is vital to identify whether the artifact produces its desired effect and is useful to its end users.

Criterion	**Description**
Accuracy	The degree of agreement between the artifact's expected output and the actual output (A. Hevner et al., 2018).
Efficacy	"The degree to which the artifact produces its desired effect considered narrowly, without addressing situational concerns" (A. Hevner et al., 2018, p. 14).
Openness	"The degree to which artifacts are open to inspection, modification, and reuse" (Gill & Hevner, 2013, p. 5:10).
Performance	The degree to which the artifact is able to perform its task within given constraints (ISO/IEC & IEEE, 2017).
Usefulness	"The degree to which the artifact positively impacts the task performance of individuals" (A. Hevner et al., 2018, p. 14).

Table 5.3: SCP evaluation criteria.

Prat, Comyn-Wattiau, and Akoka (2015) recommend to use common evaluation methods and also generate new evaluation methods taking into account the aspects of *What* and *How* to evaluate. The authors argue that even though the choice of evaluation methods is usually guided by *What* of evaluation, pragmatic considerations such as time constraints and unavailability of participants may influence the choice of evaluation methods.

The criterion of accuracy is important for a technical artifact like SCP and impacts its utility because if the artifact does not function according to its technical requirements, it might negatively affect such criteria as usefulness and efficacy. The evaluation results might end up not being truthful. Due to time constraints, I could not measure the accuracy of SCP as a whole. Thus I focus only on the command-line interface module, which was mainly developed by me. I consider unit testing an appropriate method for accuracy evaluation to test that the implemented functionality of the command-line interface works as intended.

The criteria of efficacy and usefulness were addressed during the naturalistic empirical evaluation, interpretive in nature. I consider these criteria essential in DSR, as they can determine the utility of the artifact. Evaluation of the artifact based on the criteria of usefulness and efficacy requires an evaluation from a user's perspective. The component repository is a complex system that supports several CBSE processes and has three distinct groups of users: component providers, component users, and certifiers. Therefore I considered three perspectives of user evaluation:

- Evaluation of SCP from the perspective of component providers
- Evaluation of SCP from the perspective of component users

- Evaluation of SCP from the perspective of certifiers

Evaluating SCP as component providers would include component publishing to SCP and their pre-certification, and submission of the published components for certification. Evaluating SCP as component users would include searching for reusable components on the website and testing various search-related functionality. Lastly, certifiers would need to evaluate SCP Whitelist and the certification pipeline. I considered using a demonstration as an evaluation method, showing the artifact's usefulness and efficacy for a single test case. However, since I have actively been involved in the development and testing of SCP during the design and development activity, the demonstration method would present a bias risk. If I were to conduct a demonstration, I would likely exercise the system in ways that feel natural to me and have already been extensively tested during implementation. Whereas requesting others to evaluate the system independently, without my guidance, would be more likely to result in unexpected and novel usage patterns, which may highlight problems that we did not consider during our design and implementation process. I concluded that the survey method would be appropriate for this type of evaluation, as it allowed me to perform a more naturalistic evaluation.

The criteria of openness and performance focus on the non-functional properties of the artifact and therefore were not evaluated by the users. It is unclear whether someone will take over this project and work on further development, eventually putting it in production. According to Gill and Hevner (2013), openness "encourages further design evolution" of the artifact (p. 5:10). In my opinion, a high degree of openness allows other actors in the HISP community, such as developers in HISP groups and the DHIS2 core team, to contribute to this project, and help with testing, maintenance, and improvements. To evaluate the artifact's openness and performance, I chose the expert evaluation method, which implies inspecting the artifact by a specialist with expertise in software architecture.

Additionally, the evaluation also considered the established design principles, specifically considering how they were applied to the artifact's design.

5.7.1 Evaluation participants

The type of evaluation in this study was guiding the choice of participants. As my evaluation aimed to assess both functional and non-functional characteristics of the artifact, I established that I needed participants who could act like real users of the system and participants with expertise within software architecture for the expert evaluation. The most desirable choice was to include the developers in HISP East Africa and the DHIS2 core team, as we collaborated with them during the study. However, due to time constraints, the busy schedule of the DHIS2 core team, and difficulties in establishing contact with HISP East Africa, I could only include one DHIS2 core team developer as a participant in my evaluation.

He evaluated the component repository from all three perspectives. Another participant for SCP user evaluation was chosen among the members of the DHIS2 Design lab. I considered his knowledge of DHIS2 and experience in web application development. He evaluated SCP from a perspective of component providers and component users. The third participant was chosen based on his extensive experience with CBSE, software architecture, and web application development using React framework. He evaluated SCP's functional characteristics as a user from all three perspectives and acted as an expert evaluator of non-functional characteristics in expert evaluation. The fourth participant has fifteen years of experience with software development, of which ten years was as a full-stack developer developing software for internal and external use. He evaluated SCP Website from the perspective of component users. Table 5.4 shows the participants, their eligibility basis, corresponding evaluation methods they were involved in, and the DSR criteria addressed by their evaluation.

Participant	**Eligibility basis**	**Evaluation methods**	**Criteria evaluated**
Evaluator 1	DHIS2 core team developer, has experience in web application development using React framework.	Survey (as component provider, component user, and certifier)	Efficacy, usefulness.
Evaluator 2	DHIS2 Design Lab member, Master's student, and has experience in web application development using React framework. Developed a web application for DHIS2 as part of the course *IN5320 Development in platform ecosystems* at the University of Oslo.	Survey (as component provider and component user)	Efficacy, usefulness
Evaluator 3	Currently employed as a principal software engineer in a Fortune 500 company, was the lead architect for various systems during his career, one of which was a OLTP provisioning system for mobile subscribers that was deployed at more than four mobile network operators and served more than 60 million subscribers. Has experience with CBSE and React framework.	Expert evaluation, Survey (as component provider, component user, and certifier)	Efficacy, usefulness, openness, performance
Evaluator 4	Experience with CBSE and React framework. Has 15 years of experience with software development, of which 10 years was as a full-stack developer.	Survey (as component user)	Efficacy, usefulness

Table 5.4: SCP evaluation participants.

The sampling technique used to select the evaluation participants is a mix of *convenience sampling* and *purposive sampling*. Lopez and Whitehead (2013) define convenience sampling as a "form of qualitative sampling and occurs when people are invited to participate in the study because they are conveniently (opportunistically) available with regard to access, location, time and willingness" (Lopez & Whitehead, 2013, p. 124). However, when recruiting the evaluators, I considered their experience with software development, specifically web application development, and their knowledge and experience with CBSE and DHIS2.

5.7.2 Evaluation methods

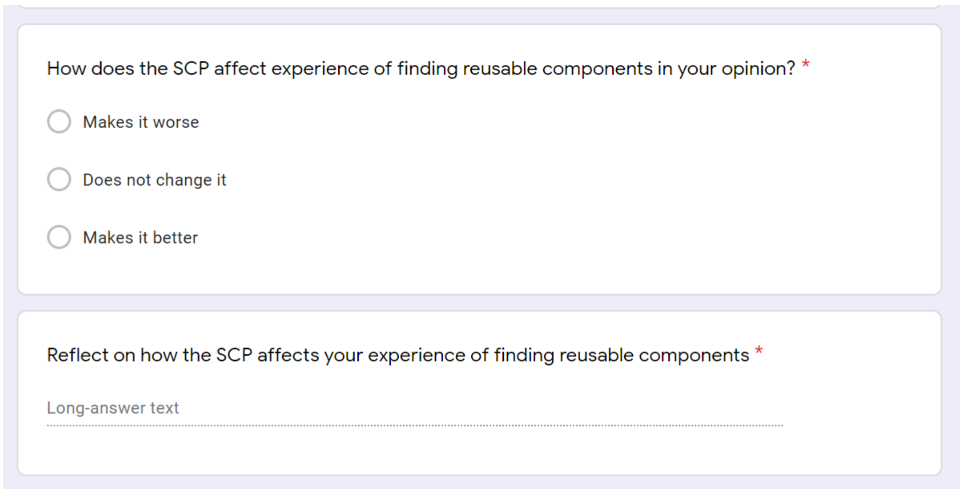
In this section, I discuss the evaluation methods used in this study. The results of the evaluation are presented in Chapter 7.

5.7.2.1 Surveys

I created three questionnaires (Appendix G) to cover the three categories of user evaluation. Additionally, I have written SCP User Documentation (Appendix B) to guide the tasks that users had to perform for this evaluation. Each of the questionnaires contained the following:

- Description of the survey's purpose
- Consent form
- Explanation of the task that needs to be performed before proceeding to answer the survey's questions
- Link to SCP User Documentation
- Set of questions that the participant should answer

It is important to note that the tasks were not detailed; for example, the participants were asked to develop a software component and publish it to SCP, but they were not given the exact steps and instructions to perform the task. The questionnaires contained open-ended questions allowing free-form answers and some close-ended questions where I felt it was necessary that had a fixed set of possible answers. Close-ended questions helped me conduct a more straightforward comparison of data between the participants and get an overall sense of the situation. Open-ended questions let the participants fully express their opinions, give deeper and meaningful insights, and elaborate on chosen answers to close-ended questions. Although, such data is often more challenging to analyze, as some of the participants provide very diverse answers, while others may opt to give very terse and superficial answers. Figure 5.7 shows a combination of a close-ended question and an open-ended question taken from one of the questionnaires.


How does the SCP affect experience of finding reusable components in your opinion? *

- ○ Makes it worse
- ○ Does not change it
- ○ Makes it better

Reflect on how the SCP affects your experience of finding reusable components *

Long-answer text

Figure 5.7: Survey questions example.

This evaluation method allowed the participants to evaluate SCP in a more naturalistic way than if they were a part of a demonstration conducted by me. They engaged in such activities as component creation, publishing, certification, and discovery using the provided user documentation on their own without any interference from my side. Additionally, this method facilitated evaluation of how intuitive the artifact is for the users and the quality of the documentation. Open-ended answers of each survey were thematically analyzed, and the insights gained can be used to guide further development of the artifact.

5.7.2.2 Expert evaluation

Peffers, Rothenberger, Tuunanen, and Vaezi (2012) classify *expert evaluation* as one of the evaluation methods in DSR and define it as an "assessment of an artifact by one or more experts" (p. 402). This study uses this method to assess the artifact's non-functional characteristics, namely coupling and cohesion, and address such evaluation criteria as the artifact's openness and performance.

Evaluator 3 and I have discussed various software coupling metrics such as *Fenton and Melton Software Metric* and *Dhama Coupling Metric* (Singh & Bhattacharjee, 2013). These metrics are mainly quantitative but they require a qualitative assessment of the system's dependencies and classifying them based on the coupling type. However, these metrics are primarily defined for source code level analysis and not for a system-level analysis. To use these metrics on a system level, they would have to be reinterpreted, but such reinterpretation would require further justification, which is outside of this study's scope. Additionally, such an assessment would be resource-intensive and time-consuming. Therefore, we agreed that Evaluator 3 would perform the analysis of the SCP's modules and system-level dependencies to qualitatively determine the degree of coupling and cohesion.

In order to proceed with the expert evaluation, I have prepared the necessary documentation for Evaluator 3. Provided documentation comprised SCP's architecture description (Section 6.4) and SCP User Documentation (Appendix B), published openly on GitHub. Evaluator 3 evaluated SCP without my active participation by first reading the documentation to understand SCP's purpose, functionality, and how to use it. Afterward, he tried out various functions of the component repository, including validation and pre-certification in the command-line interface, publishing of NPM packages with web components to the website, submission of NPM packages for certification, and searching for components on the website. He also read the architectural description of SCP, selectively analyzed the source code of the various parts of the component repository where he felt it was necessary. His evaluation was performed considering that the nature of evaluation is *formative* and that future improvements and development are expected. Evaluator 3 produced a report on the non-functional aspects of SCP. This report was reviewed and discussed by Evaluator 3 and me during a two-hour focus group on January 10, 2021.

5.7.2.3 Unit testing

Sommerville (2011) defines unit testing as "the process of testing program components, such as methods or object classes" (p. 211). As I have mainly worked on the development of one of our artifact's modules - a command-line interface, I used this type of testing to assess its accuracy by validating that the module behaves as expected. The unit tests were implemented using Jest, a JavaScript testing framework. The tests consist of two test suites comprised of 34 unit tests. Each time the tests were run, Jest generated a code coverage report that included the information about the files that were included in testing and code coverage percentage for the following code coverage methods:

- Statement coverage
- Branch coverage
- Function coverage
- Lines coverage

Statement coverage shows how many statements in the code were executed at least once. Branch coverage shows how many decision conditions in the code are executed at least once. Function coverage shows how many functions defined in the code are executed, and line coverage shows how many lines of code are covered by tests. This evaluation method was used by me during the development activity and early in the evaluation activity before SCP was evaluated by the users. I aimed at achieving a test coverage of 70-80%, which was adequate in my opinion considering the time and resources available.

5.8 Paradigmatic limitations

The philosophy of pragmatism is highly criticized for devaluing the notion of 'truth,' or rather for its redefinition of 'truth.' As Russell (2004) explains, if the effects of a certain belief increase happiness, this belief is true. Simply put, the truth is what is useful. One of the challenges pertaining to this assertion is that one cannot claim that one's beliefs are true unless one has determined what a beneficial outcome, for whom it is beneficial, and whether one has succeeded at measuring the utility. In my case, my prescriptive knowledge in the form of the design principles would have no value unless I evaluate the artifact. If a DSR researcher does not continuously measure the artifact's utility during design and implementation activity, there is a risk that he might produce a solution with no utility whatsoever.

5.9 Methodological limitations

In this section, I elaborate on the methodological limitations of this study.

As an alternative to DSR, we could also conduct Action Design Research in a field setting, closely collaborating with HISP groups' developers and working on our prototype. This methodology would also allow us to take into use other qualitative methods, such as participant observation. However, it was not possible due to the COVID-19 pandemic.

To obtain the data to guide the design and development of SCP, we conducted interviews and focus groups. These data collection methods can introduce research and participant bias. The participants may not provide honest answers but rather give the answers they think the researchers want to hear. The researcher's presence may impact the participants' responses as well. We have also not used any other qualitative data collection methods; thus, it was not possible to perform data triangulation to strengthen this study's reliability and eliminate potential biases.

The limitations of surveys are that survey respondents may not be providing honest and accurate answers. One cannot exclude the possibility that some of the questions can be misinterpreted, and some of the questions could be interpreted differently by different survey respondents. I have included many mandatory questions in my questionnaires, as I feared that my participants would skip the questions they did not have to answer. However, when a questionnaire has mandatory questions, respondents may provide answers only because they must and not because they are willing to or know what to answer. Questionnaires with close-ended questions may not be beneficial to a *formative* evaluation, as they lack the depth of feedback that could guide further development of the evaluated artifact.

Limitations of expert evaluation are that such evaluation requires participants with a high level of expertise and can be resource-intensive. Additionally, to ensure the validity of such an evaluation and reduce a single expert evaluator's bias, it is beneficial to recruit several expert evaluators and adopt several evaluation methods. In my case, I did not have the opportunity to ensure the validity of the expert evaluation by recruiting additional expert evaluators. Besides, due to time constraints and resource limitations, the expert evaluator had only used one qualitative evaluation method, while there was a clear potential for quantitative methods.

5.10 Ethical considerations

According to Creswell and Creswell (2018), "research involves collecting data from people, about people" (p. 144); thus, the researchers should consider the importance of the ethical issues related to the conducted research. Creswell and Creswell (2018) argue that a researcher should respect his participants, strengthen their trust, protect them from harm, and nurture the integrity of research.

To ensure that data collection was done ethically and with due consideration to the research participants' privacy, the participants in this study were supplied with a consent form, and they were required to sign it before participating in the research activities. The consent form (Appendix D) included the following components:

- Description of the purpose of the project
- Identification of those responsible for the project (my team and our supervisors)
- Identification of the type of participant involvement
- Notification of the voluntary participation and withdrawal at any time of the project
- Description on how the data is stored
- Guarantee of confidentiality and anonymity
- Participant's rights
- Provision of names to contact

We ensured that all our participants were aware that the participation is voluntary and they are not required to participate if they do not wish to. There would not be any negative consequences for the participants if they were to withdraw from the project. Even though we specified in the consent form that the interviews and focus group discussions would be recorded, we obtained permission from the participants before starting the recording. We also followed NSD's guidelines for research. We did not collect more data than necessary for this project and aimed to collect data as securely as possible. The data was stored on Google Drive, and its access was restricted using Google Drive authentication and authorization system. The data is to be deleted at the end of this project.

Our supervisors helped us establish contact with some of the participants, and, in my opinion, this might have affected the participants' voluntary participation in the research. It is unknown whether they were interested in the project and were willing to collaborate of their own accord or whether they felt obliged to participate. In our consent form, we explicitly say that participation in the project is voluntary, and there will not be any negative consequences for those who choose not to participate. However, we could not guarantee that there would not be any negative consequences imposed by other community actors, such as their superiors, if they choose not to participate.

We considered all data that we collected, regardless of whether the data was in line with the community's best practices. This was done in order to conduct good qualitative research "[reporting] the diversity of perspectives about the topic" (Creswell & Creswell, 2018, p. 152). In this case, it was important to ensure the participants' anonymity, i.e., instead of fully providing each HISP group's country-specific identifiers, I only specify the regional group they are part of.

Lastly, I considered it important to preserve the anonymity of one of my colleagues, providing him with a pseudonym. My thesis has content that can be seen as critical of our group work and team management, and I do not want such critical content to affect him negatively. My other colleague, Håkon André Heskja, has given his approval for using his name in my thesis.

5.11 Team management and group work

Our team can be described as a leaderless self-managing team. Since all of us are Master's students, no team member had any authority over others, and thus it would not be possible for any team member to be a group leader or a manager. It was assumed that each of us would contribute and work towards the project's success without needing a group leader. The approach we adopted was inspired by *Identity Management Method*, which works well in knowledge-oriented teams, and its main idea is to "manage by making people *identify* with the goals you're trying to achieve" (Spolsky, 2008, p.47). Spolsky (2008) states that the goal is to make your team feel like a family, cultivate loyalty and commitment to create motivation to work towards the goals. Each of us had our own reasons for choosing this specific Master's thesis topic, but all of us had a common goal of finishing this project as a requirement to achieve an M.Sc. degree.

We tried to rely on consensus for major decisions with broad and system-wide implications, for example, SCP Whitelist repository architecture, and where consensus was not possible for such decisions, we went with a majority vote. Minor decisions, for example, using TypeScript over JavaScript in the command-line interface, were delegated to the person who has worked the most on a specific module. For example, Heskja had made most minor decisions for the website, while I made most minor decisions for the command-line interface.

We had agreed upon having a group meeting on Zoom at least once a week, where we discussed the requirements that we were to implement and planned various data gathering activities. Slack and Mattermost were used as team communication and collaboration tools. Additionally, Trello was used to keep track of the various tasks. A distributed version control system, Git, was used for cooperation on the source code during the development of SCP. GitHub was used as a code-hosting facility. At first, we did not divide the tasks among the group members, as it was assumed that each of us would take up available tasks and work on their implementation. This approach was changed towards a more formalized process towards the end of the development process due to its inefficiency. Additionally, we felt it was necessary to adopt a Daily Scrum meeting format for our weekly meeting, during which each team member answered the following questions:

1. What did I do since the last meeting?
2. What am I working on now?
3. Do I need help with the tasks?

To make the development process more effective and efficient, we put a lot more focus on the Trello board by creating and prioritizing various tasks and assigning these tasks to each of the group members. Moreover, deadlines were added to each task, and we introduced a meeting to assess the group work. The following measures were taken to motivate group members to deliver their tasks within the deadline:

1. The task that is too large can be split into smaller pieces.
2. The task can be exchanged between the group members.
3. One can ask for help to finish the task in good time before the deadline.

I will further reflect on the team management and group work in Section 8.5.

5.12 Work distribution

In this section, I will give a brief account of my contribution to the implementation of DHIS2 Shared Component Platform. In collaboration with Heskja, I have created a more detailed overview of the work distribution amongst the team members (Appendix A).

5.12.1 Contribution to SCP Website

I have contributed to overall responsiveness of SCP Website using Pure CSS grid system. I did initial prototyping for the page with information about the NPM packages and the landing page (A.1 aspects 2,3). I have refactored the top navigation bar and made it responsive (A.1 aspect 1). Additionally, I have implemented the help page and the about us page with information about the DHIS2 Design Lab and HISP UiO (A.1 aspect 9). I implemented basic functionality for pagination (A.1 aspect 14), reworked and enhanced the package list on the search page, and later, the component grid (A.1 aspects 11,13).

5.12.2 Contribution to SCP Whitelist

I have created and set up SCP Whitelist repository on GitHub to host the list of certified packages (A.2 aspect 1). I have also set up the automated certification workflow using GitHub Actions and provided necessary user documentation (A.2 aspect 2,3).

5.12.3 Contribution to SCP CLI

My contribution to SCP CLI involved the creation of a framework for the implementation of the command-line interface (A.3 aspect 1). Furthermore, I have developed the functionality for `dhis2-component-search` keyword and `dhis2ComponentSearch` property validation (A.3 aspects 2,3). I have also implemented the functionality to support the certification process in SCP Whitelist (A.3 aspects 4,5,6,9). Moreover, I have written unit tests for the functionality I implemented, and provided a user documentation (A.3 aspects 7,8).

5.12.4 User documentation

For my final formative evaluation I have created SCP User Documentation (Appendix B) that explains in detail all the activities done by component providers, component users, and certifiers. I have also outlined the process of publishing reusable components to SCP and submission for certification to accommodate the users that prefer tutorials in addition to the documentation (Appendix C).

Chapter 6

Artifact description

This chapter describes my practical contribution in the form of the DHIS2 Shared Component Platform and my theoretical contribution, a set of design principles established during the design and development activity of the project.

6.1 DHIS2 Shared Component Platform within the DHIS2 platform ecosystem

In this section, I explain the role of SCP in the DHIS2 platform ecosystem.

6.1.1 SCP as a nested transaction platform

At some point during the design activity of SCP, we had to decide on the naming of our artifact. The name DHIS2 Shared Component Platform came to be when we had not quite determined whether it could fulfill the definition of a digital platform. I have already established in Section 3.1 that there are three types of digital platforms - transaction platforms, software platforms, and integration platforms.

A software platform, by definition, has an extensible core that serves as a foundation for complementary products (modules, applications). The components published to SCP do not extend SCP's functionality - they are build to constitute DHIS2 web applications. These, in turn, are the complementary modules that extend the core of the DHIS2. Therefore I conclude, that SCP cannot be defined as a software platform.

However, for transactions platforms, there is no requirement for an extensible core, as there is for software platforms. Instead, there is a focus on the interaction between distinct and mutually interdependent user groups. In the case of SCP, we have three distinct user groups: component users, component providers, and component certifiers. There is an evident interdependence between component users and component providers and the emergence of network effects. Component user do not gain any value from SCP, if there are no reusable components published by component providers, while SCP does not bring any value to component providers, if there are no component users interested in searching for, and reusing, published components. In turn, component certifiers bring value to SCP by increasing the level of component trustworthiness, which greatly impacts software reuse. The proper level of component trustworthiness in SCP may also increase the number of users interested in reusing components.

Even though we distinguish between component users and component providers, all of them have the role of web application developers in the DHIS2 platform ecosystem. Reusable components are used to construct web applications created for the end-users of DHIS2. Therefore, I conclude that SCP enables interaction between businesses, i.e., third-party development, and can be viewed as a nested B2B transaction platform. Given this, to motivate the growth and success of SCP as such a platform, the considerations relevant to these types of platforms should be taken into account. For example, increasing the number of component providers would not be enough to foster the growth and success of the platform, as such a platform's growth and success are predicated on cross-side network effects.

6.1.2 SCP as a boundary resource

Another role that SCP assumes is that of a *boundary resource* (Section 3.2) as it provides design capabilities to third-party developers and to facilitate application development. Given that SCP is a technological resource that offers technical functionality, it can also be more specifically categorized as a *technical boundary* resource. Furthermore, given that SCP is designed to support the third-party application development process, it can be categorized as a *development boundary resource.*

When considering SCP as a *development boundary resource*, it is also important to consider its social context. For this, the actors to consider are the DHIS2 core team, which has the role of platform owners, the HISP groups developers who act as third-party developers, and also the DHIS2 Design Lab, which my team and I are part of. Given that the DHIS2 Design Lab is independent of the DHIS2 core team, I view my team as third-party developers. Given this social context, the development of SCP can be classified as self-resourcing under the "Boundary resource model" presented by Ghazawneh (2012), because SCP is developed by third-party developers in "response to a perceived limitation of existing boundary resources" (Ghazawneh, 2012, p. 56).

6.2 Design considerations

When we started to work on this project, we had limited knowledge of the problem. We knew that there is an interest in building a component repository, but we did not know anything about code reuse practices in the HISP community. Therefore we conducted several interviews and focus groups with the DHIS2 core team and developers in HISP East Africa. We established a set of learning goals to help us obtain the necessary knowledge for technical requirements specification. From the interviews, we have learned about:

- application development practices and various aspects of the application development process
- motivation for software reuse
- current and prospective software reuse practices
- impediments for software reuse
- tooling
- collaboration in co-located teams (i.e., within one HISP group) and geographically dispersed teams (i.e., between different HISP groups)

The main goal of SCP is to facilitate component reuse, which includes component publishing and discovery. It also provides various functions relevant to the component management process, specifically component certification, which supports the CBSE for reuse and CBSE with reuse processes.

In the following sections, I will present the *current* component reuse practices in the HISP community that were explored in this project. I will also explain how they were taken into consideration during the construction of the artifact in the context of CBSE for reuse, component certification, and component acquisition.

6.2.1 SCP's design considerations in the context of CBSE for reuse

One of the practices discerned during the interviews is software reuse through the copying of code. One interviewee explained that their reuse practice involved the creation and maintenance of a skeleton project with the components that they want to reuse:

> What we do - we have a skeleton project where we have the components running, we have a basic project where we have the components, so we go there, it is on our bitbucket repository, we go there and reuse the source at bitbucket. (lead developer in HISP East Africa, personal communication, July 8, 2020)

This skeleton project is then copied and used as the basis for new applications that they want to develop. The interviewee mentioned a plan to create an internal component library, but his team and he were unsure whether it was a necessity for them, as their current copy-paste practice, in their opinion, enables efficient code reuse. Their reusable skeleton with components is hosted on Bitbucket, a git-based source code repository hosting service. The source code in the repository is annotated using comments that guide component identification when the developers look for candidate components for reuse purposes.

Even though the interviewee stated that copy-pasting practice makes the reuse process more effective and can be seen as a practice of code reuse with minimal effort, there are several issues pertaining to this practice. The code might have bugs and security vulnerabilities, and copy-pasting would mean introducing these issues in different applications. Additionally, when the original source is improved and updated with new functionality or a bug fix, the copied code would need to be manually updated, which is an inefficient, error-prone, and time-consuming process. For example, if a component has been modified in an application and the same component was modified in the skeleton, someone could inadvertently override the application modifications when trying and updating it from the skeleton.

Another practice we have encountered during the interviews was the component-based software engineering approach. Some interviewees stated that they use GitHub to store their reusable components, and in addition, they have also published them to NPM Registry as scoped NPM packages (HISP East Africa developers, personal communication, October 9, 2020). NPM Registry is "a public collection of packages of open-source code for (...) front-end web apps, mobile apps, robots, routers, and countless other needs" (*About npm*, n.d.). NPM also provides a package manager that helps developers to publish and install packages. An NPM *package* is defined as "a file or directory that is described by a `package.json` [file]" (*What is a package?*, n.d., para. 2). Such `package.json` file contains metadata relevant to the component, e.g., component's name, description, version, keywords, dependencies, and a link to the location of the component's source code. Metadata such as keywords, name, and description are used by NPM Registry for its search functionality, while metadata such as author, contributors, and homepage credits everyone involved in the development of the component and provides additional information. As I have mentioned earlier, the packages with reusable components published on NPM Registry by the interviewees are scoped. A *scope*, in the context of NPM, is defined as "a way of grouping related packages together and (...) a good way to signal official packages for organizations" (*scope*, n.d., para. 2). One of the advantages of scope is a restriction on who can publish the packages within the scope. To conclude, it is clear that the interviewees adopted CBSE approach and NPM Registry functions as a component repository. While their practice can work well within their HISP group, some of their technology choices can be an impediment for effective code reuse within the HISP community. When asked whether they reuse components developed by other HISP groups, one of the interviewees explained:

> Not really. In fact, we haven't really found many components from other HISP groups. (...) I guess also because for the most

> part we are using technology a little bit not used by many. (...) We have been looking for a couple of components that actually are fitting our technology stack, I mean the Angular, and we really haven't found them yet. But probably they may exist somewhere, I think we would have used them (...). So for the most part, we have used ours. (HISP East Africa developer, personal communication, October 9, 2020)

One can conclude that their use of the Angular framework for the development of reusable components has somewhat isolated them from software reuse within the HISP community because the Angular framework is less prevalent in the community. Diversity of tooling and technology has been mentioned during focus groups and an interview we have had (DHIS2 core team developers, personal communication, February 26, 2020; HISP East Africa lead developer, personal communication, July 8, 2020). We learned that some of the HISP groups use React and Angular frameworks to develop web applications, while the DHIS2 core team developers use React framework to develop their web applications and encourage the community to adopt the same practice. Some interviewees view the concept of diversity as positive, in that it gives freedom of practice, and claim that there is little diversity of web-development practices in the HISP community and that this creates a negative impact on software reuse and collaboration between HISP groups (HISP East Africa developer, personal communication, October 9, 2020).

The first practice discussed in this section is not CBSE-based, as it does not involve the creation of reusable components. Therefore it could not be considered to be the process of CBSE for reuse, and the development of a component repository would not be able to support this practice of code reuse. One can argue that developers could still benefit from a component repository, as they would be able to reuse the components published there by others; however, they cannot take the role of component providers and publish their own components to the repository unless they change their practice.

The other practice based on CBSE approach has made us question whether there is, in fact, a need for the development of a component repository given that NPM Registry is already in place. NPM Registry has recently passed its one millionth package milestone (Tal & Maple, 2019), has a large user base and functionality in place, and is in use by HISP developers and the DHIS2 core team (DHIS2 core team, personal communication, February 26, 2020; HISP East Africa developers, personal communication, October 9, 2020). Given the number of resources dedicated to the development, maintenance, and operation of NPM Registry and its long track record, we realized that we would not be able to develop something better than NPM Registry given our resources and time. We have decided not to develop an entirely new component repository but rather cultivate the installed base by reusing and extending the existing infrastructure, which is, in this case, NPM Registry and GitHub. Our solution's primary goal would be to support and improve the existing CBSE approach by addressing some of the challenges we have encountered with existing technologies, services, and tools. This approach's benefit is that the resources required for the maintenance of our solution would be minimal and that our solution would benefit from future improvements to the installed base that we cultivated. Since we have taken into use the functionality of NPM Registry, which operates on package-level, I will often use the concept of an NPM package when discussing the implementation of SCP as all components that SCP deals with have to be contained in NPM packages.

When we considered the existing CBSE practice detailed above, we encountered three challenges that needed addressing: *component designation*, *component discoverability*, and *framework diversity*.

The *component designation* issue came to our attention during one of the interviews when we asked the interviewees to explain how they look for reusable components. One of them explained that they mainly look for reusable components in NPM Registry and through Google Search, and, in his opinion, it is easier to find reusable components when they include keywords, e.g., `dhis2` (HISP East Africa developer, personal communication, October 9, 2020). It is worth noting that when I have looked at some of the packages within the organizational scope on NPM of this interviewee, some of the packages had keywords, and some packages did not. It was clear that there is no standard way of designating NPM packages as related to DHIS2.

We encountered the *component discoverability* issue during one of the focus groups with the DHIS2 core team developers. A developer explained:

> I just think it makes the search much more useful, (...) the ability to search for a specific component, if I'm developing an application and I want (...) an org unit tree, and I search for org unit tree, and there only comes up with the one, or doesn't even come up with the one from ui-core because that doesn't exist as a package by itself, right, its just a component within a UI package, and there are maybe three other org unit trees that other people have developed that are all part of their kind of utility library, (...) *I don't find it to be super helpful* [emphasis added] to me. (DHIS2 core team developer, personal communication, October 2, 2020).

NPM Registry allows component providers to publish single components within one NPM package, however in many cases a single NPM package contains component libraries with multiple components. DHIS2 core team developer pointed out that NPM Registry search operates on packages and not on individual components within these packages. Therefore, individual components are not made discoverable to component users. DHIS2 core team developer suggested we "find some ways to expand the library and expose the individual components" (DHIS2 core team developer, personal communication, October 2, 2020).

The third issue, *framework diversity*, came to light during the discussion with one of HISP groups detailed above when they noted that different HISP groups use different frameworks. There is also no standard way of specifying what framework the components in an NPM package supports, and thus also no standard way to discover components supporting a specific framework.

To address these challenges we developed an extension to the standard `package.json` file format that NPM packages should adhere to for them to be included in SCP. In the SCP `package.json` extension we address the *package designation* issue by mandating that packages must include `dhis2-component-search` keyword in their `package.json` files. We address the *component discoverability* and *framework diversity* issues by having an explicit machine-readable specification of components contained in an NPM package which also includes an explicit specification of the framework supported by the components. Component providers must specify this information in the `dhis2ComponentSearch` property of their `package.json` files. The `dhis2ComponentSearch` property must include a list of reusable components that are included in the package with the following meta-data: component name, description, export identifier and optional information about the supported DHIS2 versions.

Additionally, we also provide tooling, in the form of a command-line interface (SCP CLI) for automatic validation that an NPM package complies with the SCP `package.json` extension.

To conclude, the following design decisions were made with regard to CBSE for reuse:

1. Our component repository utilized the installed base in the form of NPM Registry and GitHub and addressed the identified challenges pertaining to it.

2. We developed an extension to the standard `package.json` file format to resolve the challenges of component designation and component discoverability. This extension allows us to aggregate components developed for DHIS2 from NPM Registry and then index individual components that are part of component libraries.

3. We provide a command-line interface to s as a means to validate that their NPM packages comply with the SCP `package.json` extension.

6.2.2 SCP's design considerations in the context of component certification

During our first meeting with the DHIS2 core team on February 26, 2020, some of the core team developers have shown interest in implementing component certification for the component repository. While discussing potential functional requirements for SCP, the core team developers stated that it would be nice "to promote the ones [components] that have a certain level of quality, or that they have a certain level of maturity, and testing" (DHIS2 core team developer, personal communication, October 2, 2020).

In CBSE, the certification process is concerned with establishing standards for components, evaluating the components against set standards, and then marking or designating the component as certified. Given this, we can use the term certified component to refer to components with a certain level of quality, maturity, and testing. The request from the DHIS2 core team then translates to promoting certified components. Furthermore, for SCP to promote certified components, it would imply that there are two types of components, certified and not certified. Based on this, our team decided that the component repository would include both non-certified and certified components. Allowing non-certified components would also reduce the barrier to entry for component providers as they would be able to publish to the platform without understanding the certification process. It would also reduce turnaround time as the component providers would be able to release new versions and make new versions available without having to wait for certifiers. Additionally, as discussed in Section 3.3.6, the need for certification is reduced when the source code for components is available to component users, and in most cases, the source code should be available given the nature of the HISP community.

During our development work on certification, I have identified three aspects of certification with regard to its implementation: *certification functionality*, *certification architecture*, and *the role of a certifier*.

Certification architecture is concerned with how component certification should be designed. The design process of the certification architecture was driven by two main requirements: the architecture should support required functionality and it should incorporate the existing infrastructure, specifically NPM Registry and GitHub. Given the usage of NPM, and given that NPM Registry operates on packages, and that component user can only install whole NPM packages, and not individual components from NPM packages, it seemed most practical for certification to operate on NPM packages. Thus, in our case, what would be certified is a whole NPM package and not the individual components inside it. During a focus group with the DHIS2 core team on October 2, 2020, we have reviewed three different component certification design options: *Unipackage*, *Unirepo*, and *Whitelisting* (Table 6.1).

The first two options are relatively similar. Certifiers would be maintaining a GitHub repository with the source code of certified components. Certifiers would perform the certification on the source code of a component, submitted by a component provider. The first option - *Unipackage* would imply having one GitHub repository from which a singular NPM package is published on NPM Registry. To use the components submitted in such a repository, one would need to install all the components, e.g., `npm install @scope/unipackage_name`. This option had clear disadvantages - the users would not be able to install only a subset of the components or use different versions of different components. The user would also not be able to

Certification option	**GitHub repository**	**NPM package**	**Certification designation**
Unipackage	One GitHub repository owned by certifiers.	One NPM package owned by certifiers.	Only certified components are included in the GitHub repository and NPM package.
Unirepo	One GitHub repository owned by certifiers.	Multiple NPM packages owned by certifiers.	Only certified components are included in the GitHub repository.
Whitelisting	Different GitHub repositories owned by component providers.	Different NPM packages owned by component providers.	Only certified components are included in SCP Whitelist.

Table 6.1: Overview of the certification options.

use different versions of different components, all components would have to have a corresponding version, so if a newer version of one component is desired all other components will also have to be upgraded. This would be similar to the radiance-ui component library (*Radiance UI*, 2021) which is a single NPM package published from a single Git repository but containing multiple React UI components. The second option, *Unirepo*, would imply having one GitHub repository containing multiple NPM packages, each published to NPM Registry as individual NPM packages. To use the components within this repository, one would need to install only the NPM packages with these components, e.g., `npm install @scope/component-a @scope/component-b`. Users could install a subset of all the components, as well as different versions of different components. The DHIS2 core team found the first option to be least effective, as the "Git [GitHub] repository quickly becomes large and hard to manage" (DHIS2 core team developer, personal communication, October 2, 2020). The second option was "similar to what we [the DHIS2 core team] do with dhis2 ui at the moment, and that could be a decent option I think" (DHIS2 core team developer, personal communication, October 2, 2020). This would be similar to the Material UI component library, which consists of multiple NPM packages published from a single Git repository, each NPM package contains multiple React UI components (*Material-UI*, 2021).

The third option, *Whitelisting*, was designed and suggested by me as an alternative to the other options. I was dissatisfied with the two first options because of the high maintenance burden it would place on certifiers, as they would potentially have to deal with a large number of certified components. The *Whitelisting* option would lessen the amount of work required of certifiers maintaining GitHub repositories with reusable components. This option would involve having one GitHub repository with a file that contains a list of certified NPM packages with reusable components. Each NPM package could have its own GitHub repository, and users could use a subset of all components and different versions of different components. Thus, the certifiers would only maintain a list of NPM packages instead of maintaining their source code. The DHIS2 core team agreed that *Whitelisting* would be the best choice as it "is much more community-based" (DHIS2 core team developer, personal communication, October 2, 2020). Taking the core team's recommendation into account, we decided to proceed with the implementation of *Whitelisting* and set up of SCP Whitelist repository on GitHub.

There were plans to set up SCP Whitelist under the `dhis2` organization on GitHub; requiring DHIS2 core team's involvement (DHIS2 core team developer, personal communication, October 2, 2020). However, we could not get in touch with them. Due to time constraints, our team decided to implement the repository on our own (internal team discussion, personal communication, October 22, 2020). The task and responsibility of setting up SCP Whitelist repository were assigned to me, and SCP Whitelist repository was set up under the DHIS2 Design Lab organization on GitHub.

During the implementation of SCP Whitelist, we had to make several architectural choices. First, a decision had to be made whether we would certify a specific version of a package or certify a package without regard to its individual versions. The DHIS2 core team said there is no need to implement support for specific versions of a package; however, one of them noted that "it's more work to whitelist specific versions, but it's safer" (DHIS2 core team developers, personal communication, October 2, 2020). On the one hand, certification of specific versions can require more effort and time; on the other hand, certifying a package without considering specific versions could undermine certifiers' credibility if later versions of they turn out to be malicious. Another concern we had was that it would be best if the certifiers had a mechanism to revoke the certification status of a package in a way that component users of the package would notice such revocation. If the only way for component users to notice a revocation of the certification status of a package was to check SCP, this will place a significant burden on them to perform a tedious and even error-prone manual task. One way of addressing this problem would be creating an NPM scope for certified components. Certifiers would own and maintain this scope and would retain exclusive rights to publish packages to this scope. The package's certification status could be revoked by unpublishing the package from the scope or by marking it as deprecated. In this case, a component user that uses the package would immediately become aware of the revocation. One of the core team developers stated that "it's more work and responsibility to publish packages under our own NPM scope. I think a single file with whitelisted/recommended packages is enough" (DHIS2 core team developer, personal communication, October 22, 2020). Another core team member noted that "(...) maintaining an NPM scope for all contributed packages should be avoided for now - we can add it in the future if there is demand" (DHIS2 core team developer, personal communication, November 13, 2020).

I conclude that the DHIS2 core team's decisions are heavily impacted by the availability of resources, such as time and effort. We have considered their advice and decided not to proceed with implementing NPM scope. However, the package versioning support was implemented, as we were aiming to increase component trustworthiness.

The architecture of the certification pipeline in SCP Whitelist was based on suggestions from the DHIS2 core team:

> In terms of the pipeline and the whitelisting and all of that, I think, it would be very cool to do some of the checks *automatically* [emphasis added]. (DHIS2 core team developer, personal communication, October 2, 2020)

Taking this request into consideration, we decided to implement an automated event-driven workflow that runs automated certification checks when a specific component's certification is requested. For this, we used the existing feature of GitHub, a workflow system called GitHub Actions. Our approach's benefit is that it does not require the overheads involved with the maintenance of source code; it also allows independent versioning of components. Besides, it allows component providers to evolve the components independently from the component certifiers. It makes it easy for component providers to rapidly evolve their components and still certify components even after early adopters have already started using them. Another benefit of this approach is its openness. It provides an open certification process where both the component providers and the certifiers can see the certification workflow's output.

Another consideration for certification architecture came to our attention when discussing certification functionality with one of the HISP groups. A senior developer asked us:

> What is the [certification] process, have you mapped out, let's say a checklist of what you go through to decide that package is [certified] or not? (senior developer in HISP East Africa, personal communication, October 9, 2020)

It became clear to us that it is crucial for component providers to have a clear understanding of the certification requirements so that they could create and maintain their components in compliance with certification requirements. One way of addressing this problem was to provide the functionality to component providers that would allow them to perform the same certification checks on their components that the certifiers would do when the components are submitted for certification. To achieve this, we implemented certification functionality in SCP CLI in such a way that it could be used by component providers (hereafter pre-certification), and by both certifiers for certification process in SCP Whitelist.

Certification functionality is another important aspect of component certification. It is concerned with the capabilities of a software product to fulfill the needs of its user. Different options for certification checks were discussed with the DHIS2 core team. The first option would consist of fully automated certification checks, and if all the checks passed, the package would be marked as certified. If at least one of the automated checks failed, the package would not be certified. The second option was a more subjective certification process. It would mean that the output of the automated certification checks would only serve as a basis for decision-making. However, whether the package passes certification or not would be primarily based on human discretion. We decided to adopt the second option for our initial implementation.

In mid-October 2020, I had to decide what certification checks should be implemented. I consulted with the DHIS2 core team before making the final decision and suggested the following checks:

1. Security review check of the dependencies with npm audit, or OWASP dependency check
2. Static code analysis for problematic code patterns detection with SonarCloud, ESLint, or Flow
3. Check whether the package is in active development and is actively maintained, i.e., checking whether the component providers are working on fixing issues
4. Quality-related check to determine test coverage and other code metrics

The DHIS2 core team gave non-specific feedback on these options, with one member saying, "I think you got them [certification checks] all," indicating that these are all indeed good options to consider (DHIS2 core team developer, personal communication, October 22, 2020). Another member chose to not discuss specific checks and stated that "the best option is to start simple and incrementally add functionality as the component [repository] is adopted and used, based on the feedback from the users" (DHIS2 core team developer, personal communication, November 13, 2020). Considering the time constraints, we opted to limit the implementation of certification checks to security review check `npm audit` and static code analysis with ESlint. These checks were automated, producing the output that would serve as a basis for certification assessment by the certifiers.

To determine the nature of a certification process, I propose the Certification Process Classification Model shown in Figure 6.1. This model decomposes the nature of a certification process into two dimensions, one of them is how manual or automated the certification process is, and the other dimension is how objective or subjective the certification process is. Certification checks can be either fully automated, like `npm audit` check and static code analysis with ESLint in the case of SCP. Alternatively, they can be fully manual, meaning that a certifier could manually perform various components' assessments. Additionally, the certification process could be objective, meaning that certification would be performed against a specific, measurable set of requirements. For example, `npm audit` can detect vulnerabilities and assign them a low, moderate, high, or critical audit level. Based on this, certifiers could state that certification would fail if the assessed package has a high or a critical audit level. A subjective certification process would be based on human discretion. For example, the only way to determine whether a component is designed following UI design guidelines, i.e., using the right layout, spacing, typography, and colors, is through a subjective assessment. I have assessed the nature of the SCP certification process and placed it within the Certification Process Classification Model as shown in Figure 6.1. I view the SCP process of certification as manual, as it has a certain level of human discretion despite the checks being automated. Even though the output of `npm audit` and static code analysis with ESLint can be measurable and fully automated, in our case, they only produce an output for certifiers. I also view the SCP process of certification as as highly subjective, as there are no clear certification requirements to guide certifiers on how to analyze the output.

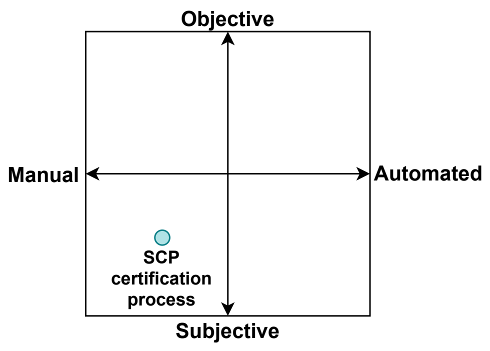


Figure 6.1: Certification Process Classification Model.

A more manual process of component certification would be more costly and time-consuming for certifiers. In contrast, once created, an automated certification process could be much faster than a manual process and run at no additional cost. Additionally, an automated certification process could improve certification accuracy. If the certification process is objective, it can enable the development of pre-certification functionality for component providers, allowing them to assess their components beforehand to make sure they comply with certification requirements. A more subjective certification process complicates the situation for component providers, as they would not know how certifiers would assess their components unless the certifiers are clearly communicating their certification requirements. My Certification Process Classification Model can be used by practitioners to assess the nature of their certification processes, better communicate the nature of their certification process, and it can also provide them with insights into the consequences and implications of their certification process. For example, this model would help practitioners understand that the more manual and subjective the certification process is, the more difficult it becomes to provide a pre-certification mechanism to component providers. It would also help practitioners to understand that more subjective processes preclude more automated processes or that the objectiveness of a certification process is inversely proportional to how automated a certification can be.

The role of a certifier is an important aspect of component certification, especially in a context of a software platform ecosystem, as it has significant implications for the whole software platform ecosystem. The DHIS2 core team has several times shown interest in taking on the role of certifier (DHIS2 core team developers, personal communication, October 2, 2020; DHIS2 core team developers, personal communication, February 26, 2020). However, it would mean that the core team would not be able to certify their own components, as the previous research on component certification specifically states that the components should be assessed by an independent third party. My team and I have not managed to establish who should be responsible for the certification process in SCP; thus, this issue will have to be addressed by someone who will take over this project. An important consideration for the role of a certifier is how the certification requirements will affect the platform ecosystem. Claiming that the certified components have a high level of trustworthiness and quality would make certification requirements a sort of guideline for component providers and impact their practices and autonomy. Given this, one of the challenges is determining who should take the role of a certifier and who would be in a position to judge what characteristics a trustworthy, high-quality reusable component should possess. The DHIS2 core team, as certifiers, could use certification as a tool to gain more control over application development and shift component providers practices in the ecosystem. An example of it could be certification requirements that state that in order to pass the certification, the reusable component should be written in React framework and be designed according to DHIS2 Design System. It would mean that the component providers that use the Angular framework would have no choice but to migrate from Angular to React if they want their components to be certified and promoted on SCP. This was discussed with my supervisor, and he stated:

> I agree that 'certifiers' outside HISP UiO would create more egalitarism [*sic*], but it will also take away some 'control' from HISP UiO. (supervisor, personal communication, February 4, 2021)

The level of subjectivity in the certification process affects how much power and influence an individual certifier would have. If there is a lack of clearly specified certification requirements and the certification process relies on human discretion, the individual certifiers would introduce their own prejudices of what a high-quality component is. When choosing individuals to take on the role of certifiers, one should carefully assess what implications it might have on a governance balance in a platform ecosystem, given the level of human discretion in the certification process.

To conclude, the following design decisions were made with regard to component certification:

1. *Whitelisting* was chosen as the design option for the certification architecture. This design option implies that component certifiers only maintain a list of NPM packages and do not have to spend time and resources maintaining the source code of NPM packages.
2. Certificates are issued for specific component versions to increase component trustworthiness.
3. A command-line interface was developed to perform some automated checks for the purpose of certification. Additionally, this command-line interface is available to be used for the purpose of pre-certification by component providers.
4. The certification pipeline was automated using the GitHub Actions workflow.

6.2.3 SCP's design considerations in the context of component acquisition

SCP Website was developed to provide functionality to component users for component discovery. As I have previously mentioned, we cultivated the installed base by reusing NPM Registry. This means the reusable components are stored in NPM Registry as packages, while SCP Websites acts as an aggregator, indexing these packages and displaying the components according to their specification provided by component providers using the SCP `package.json` extension. *Framework diversity* is accommodated with functionality that enables component users to filter the components based on the framework the components are built with. When discussing the component search functionality on SCP Website, one of the DHIS2 core developers explained that:

> In some cases, a component might be built in a way that only supports 2.28 [DHIS2 version] or something (...), and so we would want to have some way to expose that information in the search, or maybe be able to filter by the DHIS2 version support of those individual components. So that would be something that would be another beneficial enhancement, I think to this component search functionality. (DHIS2 core team developer, personal communication, October 2, 2020)

To accommodate this request, we added functionality to SCP Website to filter components by their DHIS2 version; this version is specified in an NPM package's `package.json` file, again using SCP `package.json` extension. We had two options for what set of components to show on SCP Website, we could show only certified components, or we could show certified and non-certified components and then provide a visual indication to component users for certified components. We decided that it would be best to show certified and non-certified components, as this, in our view, would ease and accelerate the adoption of SCP and make SCP more open to diversity in the ecosystem. We also wanted to give component users the option to make their own decisions whether the quality of a component is satisfactory for them, which they can assess either on trust or by evaluating the source code of a given component if available. Additionally, we did not want to limit the variety of components, especially not this early in the life-cycle of SCP, as we expect the quality standards to evolve as adoption increases. To accommodate component users that are only interested in using certified components, we decided to add functionality to SCP Website that enables them to view only certified components.

To conclude, the following design decisions were made with regard to component acquisition:

1. We developed a website that aggregates designated NPM packages published to NPM Registry. This website indexes the components stored within the packages, displays them, and makes them searchable for component users.

2. We implemented component filtering based on the framework and DHIS2 version these components are compatible with.

3. We implemented a visual indicator to show what component versions are certified.

6.3 Software design approach

A component repository is a part of a component-management process concerned with reusable components storage and cataloging. It must provide support for other CBSE processes. During the design activity, we have established our user roles (component provider, component user, and certifier), and the functional requirements to support the processes of CBSE for reuse, component acquisition, and component certification. In order to reduce the complexity of SCP and assure a high degree of maintainability, we adopted a modular approach to building an orthogonal system, i.e., highly cohesive and loosely coupled.

6.4 The architecture of SCP

The overall architecture of DHIS2 Shared Component Platform (Figure 6.2) consists of the following modules: *SCP Website*, *SCP command-line interface*, and *SCP Whitelist* repository. The platform also uses two external systems: *NPM*, a code-sharing platform, and *GitHub*, a code hosting platform.

NPM Registry stores components published by component providers as packages. Component providers use SCP CLI for the purpose of validation of component metadata and pre-certification. SCP Whitelist includes a list with identifiers and versions of certified components, this module uses SCP CLI for certification workflow. SCP Website is the interface to component users. It fetches the components published to SCP from NPM Registry, then uses UNPKG to retrieve `package.json` file from each of these packages and index individual components inside the NPM packages. This module fetches the certification status from SCP Whitelist repository. This section will discuss each of these modules, their role and function in the overall system design, and what interfaces they use and provide.

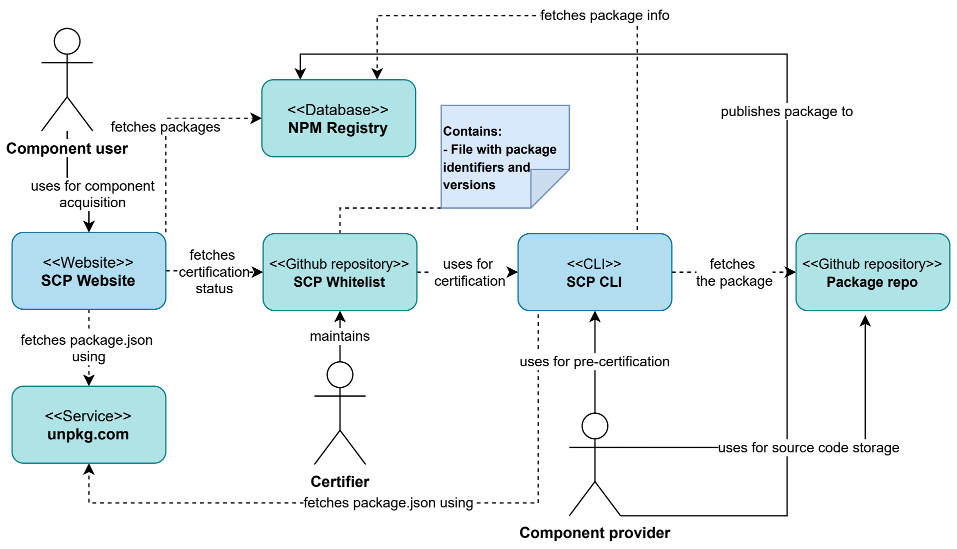


Figure 6.2: Context diagram of the SCP's architecture.

6.4.1 SCP Website

SCP Website serves as the primary interface to component users. Specifically, it allows component users to search through components in NPM packages published to SCP. The search mechanism also provides the following filtering functionality on search result:

- Framework filter
- DHIS2 version filter
- Certified components filter

The framework filter allows filtering components by front-end frameworks, specifically React and Angular. The DHIS2 version filter allows filtering components by selecting a specific DHIS2 version components are compatible with. The certified components filter allows component filtering by its certification status. SCP Website also provides component users with clear and concise information about a component (Figure 6.3), including its name, description, and exported identifier. Besides, it provides information about the package the component belongs to, including its identifier, its keywords, its latest non-certified version, its latest certified version, and a link to its NPM page.

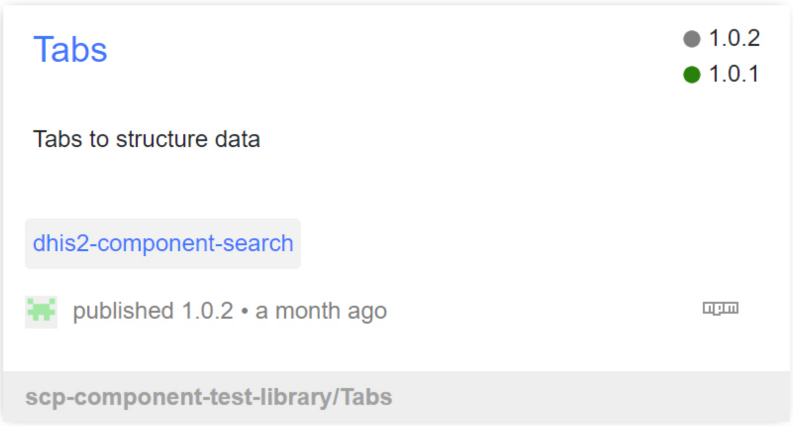


Figure 6.3: Component representation on SCP Website.

SCP Website was built using React, an open-source, front-end JavaScript library. It uses the Redux library for state management. Redux makes it easy to manage state consistently and coherently across different components. Pure CSS framework was used to ensure faster web development and website responsiveness, i.e., proper rendering across multiple devices. Pure CSS is a mobile-first responsive grid system that can be used through CSS class names. We have also used several reusable UI components from React Bootstrap front-end framework.

SCP Website uses the interfaces shown in Table 6.2.

Interface	**Interface provider**
NPM Registry API	npmjs.com
UNPKG API	unpkg.com
SCP Whitelist file format	SCP Whitelist (GitHub repository)

Table 6.2: The SCP Website used interfaces.

NPM Registry API is used by SCP Website to fetch packages with a `package.json` file that contains the `dhis2-component-search` keyword and `dhis2ComponentSearch` property. In the early phase of the project, we utilized npms.io API to retrieve NPM packages published on NPM Registry. However, due to instability of npms.io API, a decision was made to migrate to NPM Registry API.

The UNPKG API is used by SCP Website in order to retrieve an NPM package's `package.json` file for the purpose of extracting component data.

The SCP Whitelist file format is used by SCP Website in order to determine the certification status of an NPM package considering its version. The file conforming to the SCP Whitelist format is fetched from GitHub using the HTTP protocol.

6.4.2 SCP CLI

SCP command-line interface serves as one of the primary interfaces to component providers. It provides functionality to component providers that enables them to validate that their packages are compliant with requirements for publication on SCP. This functionality is provided by the `verify` command and includes the following checks:

- Check to verify whether the `package.json` file includes `dhis2-component-search` keyword.
- Check to verify whether the `package.json` file includes a correctly structured `dhis2ComponentSearch` property.

The `dhis2ComponentSearch` property contains key/value pairs for a chosen framework, which are `react` and `angular` for the current implementation, and specification of the exported components. Each component object requires the exported component's name, description, and the exported component's identifier. The `dhis2Version` property is optional and comprises the DHIS2 versions the components are compatible with. Code listing 6.1 shows an example of `dhis2ComponentSearch` property structure.

```
{
   "dhis2ComponentSearch": {
    "language": "react",
    "components": [
      {
        "name": "Simple Card",
        "export": "SimpleCard",
        "description": "A simple react component
           card",
        "dhis2Version": [
          "31.1.0",
          "32.0.0"
        ]
      }
    ]
   }
}
```

Listing 6.1: Example of `dhis2ComponentSearch` property structure.

SCP command-line interface provides additional checks that are part of certification and pre-certification. First, it lints the code using ESLint, a static code analysis tool for finding problematic code patterns. The other check is running a security audit with `npm audit` that does "an assessment of package dependencies for security vulnerabilities" (NPM, 2020, para. 1).

Moreover, SCP CLI is a part of the SCP Whitelist certification workflow. All the functionality described above is being used for package certification, and the certification workflow is to be assessed by certifiers.

SCP CLI's pull request certification functionality is provided through the `pr-verify` command and includes the following checks:

- Check to verify whether the event is a pull request
- Check to verify that only one file has been modified
- Check that the modified file is `list.csv`
- Check to validate package identifier and its version
- Check to verify that the package's `package.json` file specifies git repository with current structure

When the checks listed above pass, SCP CLI proceeds to perform the checks provided by the `verify` command.

Initially, SCP CLI was built using JavaScript, but later in the development process, I decided to migrate the project to TypeScript mainly because of type safety, null safety, type inference. The benefit of type safety and null safety is that many classes of errors relating to incorrect usage of variables can be detected without running the code. These safety mechanisms need to know the types of variables determined by either explicit type specifications or by type inference. Explicit type specifications are where the user explicitly specifies the types of variables, arguments, and return types. Type inference is "the process of determining the types of expressions based on the known types of some symbols that appear in them" (Mitchell & Apt, 2001, p. 135). It allows the compiler to infer the types of variables, arguments, and return types from the types of other variables, arguments, and return types if the user did not explicitly specify them. I also sought advice from the DHIS2 core team on whether it would be appropriate to use TypeScript. The DHIS2 core team developers have stated that it is fine to use TypeScript if I prefer it, however they do not use it for any of their CLI modules today.

SCP command-line interface uses the interfaces shown in Table 6.3.

The UNPKG API is used by SCP CLI in order to retrieve an NPM package's `package.json` file to extract the package's Git repository URL. Git repository URL is used for the purpose of cloning the source code of a package for the purpose of certification.

Interface	Interface provider
UNPKG API	unpkg.com
GitHub API	GitHub.com
Git protocol	The Git repository host of the NPM package

Table 6.3: The SCP CLI used interfaces.

The GitHub API is used by SCP CLI to verify that the changes in a pull request are only done to one file, namely the `list.csv` file, and determines what changes were made to this file.

The Git protocol is used by SCP CLI to clone an NPM package's source code for the purpose of certification.

The interfaces provided by SCP CLI are shown in Table 6.4.

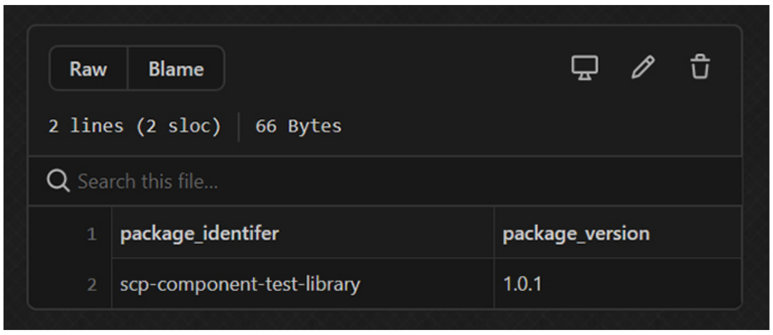
Interface	Interface users
PR verify command interface	SCP Whitelist

Table 6.4: The SCP CLI provided interfaces.

SCP Whitelist uses the PR verify command interface to validate pull requests. This happens automatically when a pull request is created. When invoked through the PR verify command interface, SCP CLI performs validation of the SCP `package.json` extension and runs automated certification checks.

6.4.3 SCP Whitelist

SCP Whitelist serves as the primary interface to certifiers and as one of the interfaces to component providers. It provides the functionality to component providers that allows them to submit a specific version of their package for certification. The repository includes a `list.csv` file (Figure 6.4) that, in turn, includes two columns labeled `package_identifier` and `package_version`. The `list.csv` file in the master branch of this repository is the list of certified components. In order to submit a package for certification, a component provider must create a pull request modifying this file by adding a new line with an NPM package `identifier` and its `version` separated by a comma, for example, `component-card,4.4.1`. To do this, a component provider would need to create a fork of SCP Whitelist repository and edit the `list.csv` in the fork and then create a pull request to merge this fork into the master branch. The GitHub web interface simplifies this process by automatically creating a fork and pull request when someone tries to edit this file via the web interface.



2 lines (2 sloc) | 66 Bytes

1	package_identifer	package_version
2	scp-component-test-library	1.0.1

Figure 6.4: The structure of the `list.csv` file in SCP Whitelist repository.

As a result of pull request creation, a pull request event triggers the automated certification workflow implemented using Github Actions. Github Actions is meant to automate certain tasks, and allows one to run commands after a specific event has occurred. The certification workflow is configured and set up in the `validate-workflow.yml` file. It contains the name of the workflow, environment it runs on (e.g., `ubuntu-latest`), the name of the GitHub workflow event that triggers the workflow, and information about the steps that are run in this job. In the context of the SCP Whitelist workflow, it performs the following steps:

1. Write the pull request event to a file named `event.json` (Code listing 6.2, line 14-16).

2. Get the SCP CLI package and build it (Code listing 6.2, line 17).

3. Run the SCP CLIs `pr-verify` command in the environment of that package, passing `event.json` as an argument (Code listing 6.2, line 17).

```
 1 name: CI-Validate
 2 on:
 3   pull_request_target:
 4     branches: [ main ]
 5
 6 jobs:
 7   build:
 8     name: Validate
 9     runs-on: ubuntu-latest
10     steps:
11       - uses: actions/checkout@v1
12       - uses: ./.github/actions/validate-action
13       - run: |
14           cat - > event.json <<EOF
15           ${{ toJson( github.event ) }}
16           EOF
```

```
          npx -p "https://github.com/haheskja/scp-
            cli#master" dhis2-scp-cli -vvv pr-
            verify event.json
```

Listing 6.2: SCP Whitelist workflow file.

The interfaces used by SCP Whitelist are shown in Table 6.5.

Interface	**Interface provider**
GitHub Actions workflow format	GitHub.com
PR verify command interface	SCP CLI
Git protocol	GitHub.com

Table 6.5: The SCP Whitelist used interfaces.

The GitHub Actions workflow format is used by SCP Whitelist to define the actions that must be taken on a pull request creation. Specifically, it uses this format to specify that the PR verify command interface must be invoked.

The PR verify command interface is used by SCP Whitelist to verify a pull request after being invoked from the GitHub Actions workflow.

The Git protocol is used by SCP Whitelist to retrieve the latest version of SCP command-line interface and run the `pr-verify` command for the purpose of certification.

The interfaces provided by SCP Whitelist are shown in Table 6.6.

Interface	**Interface users**
SCP Whitelist file format	SCP Website

Table 6.6: The SCP Whitelist provided interfaces.

SCP Website uses the SCP Whitelist file format to determine a package's certification status and display it on the website.

6.4.4 NPM Registry

NPM Registry is a package hosting platform and an external module of SCP. It serves as an interface to component providers and provides functionality for publishing packages with reusable components. SCP Website uses NPM Registry to fetch packages that contain the `dhis2-component-search` keyword in their `package.json` file.

NPM Registry provides the interfaces shown in Table 6.7.

NPM Registry API provides other subsystems with functionality to query published packages. Specifically, it allows retrieval of all packages

Interface	**Interface users**
NPM Registry API	SCP Website

Table 6.7: NPM Registry provided interfaces.

with specific keywords. In the current architecture, only SCP Website uses this interface to fetch packages published by component providers.

6.4.5 GitHub

GitHub is "a code hosting platform for version control and collaboration" (Github, 2020, para. 2) and an external module of SCP. It provides developers with Git hosting for their source code. Additionally, it provides functionality for pull request management. It also has functionality for executing automated workflows on specific events, such as on pull request creation, and it provides API that makes it possible for other subsystems to extract information about repositories and events such as pull requests.

Table 6.8 shows the interfaces provided by GitHub relevant to SCP.

Interface	**Interface users**
GitHub API	SCP CLI
Git protocol	SCP CLI and SCP Whitelist
GitHub Actions workflow format	SCP Whitelist

Table 6.8: GitHub provided interfaces.

The GitHub API provides other modules with a mechanism to retrieve information about users, repositories, issues, and pull requests, among other things. In the current SCP architecture, the specific system that uses the GitHub API is SCP CLI. It uses the GitHub API to check that a pull request only changes the `list.csv` file and determine what exact changes were made to this file.

The Git protocol provides other modules with a mechanism to retrieve the content of a Git repository. In the current SCP architecture the specific modules that use the Git protocol is SCP CLI and SCP Whitelist. SCP Whitelist uses it to fetch the latest version of SCP CLI. SCP CLI, in turn, uses it to fetch the source code of the package for certification purpose.

The GitHub Actions workflow format provides a mechanism for specifying actions to perform when specific events occur on GitHub hosted git repositories and provides infrastructure for executing these commands. In the current SCP architecture, the specific module that uses the GitHub Actions workflow format is SCP Whitelist, which uses it to specify that the `pr-verify` command from SCP CLI should be run when a pull request is created.

6.4.6 UNPKG

UNPKG is a global content delivery network for NPM (UNPKG, 2020, para. 1) and an external module of SCP. It is used to retrieve files from the packages hosted in NPM Registry. It is an external service used by SCP Website and SCP CLI. UNPKG provides the following interfaces:

Interface	**Interface users**
UNPKG API	SCP Website and SCP CLI

Table 6.9: UNPKG provided interfaces.

SCP Website uses the UNPKG API to fetch an NPM package's `package.json` file so it can extract the components provided by the package. SCP CLI uses the UNPKG API to fetch `package.json` file and extract the value of its `repository` property to be able to clone the package's repository.

6.5 Design principles

As a theoretical contribution of my research, I establish a set of essential design principles for a component repository that facilitates component reuse in the context of a software platform ecosystem. These design principles can guide the construction of similar artifacts. I define *essential design principles* as principles that are important for a component repository to effectively fulfill its function in the context of a platform ecosystem, as this is the context considered in my research question. In the case of the SCP, the function is to effectively drive component reuse.

When formulating the design principles, I have followed the conceptual schema for prescriptive knowledge formulation for IT-based artifacts proposed by Gregor, Kruse, and Seidel (2020). My design principles have the following structural components: an Aim, a Context, a Mechanism and a Rationale (Table 6.10).

I do not explicitly define the implementers in the design principles, as all of them are targeted towards someone who is to implement a component repository in a similar context. These principles were identified through the analysis of the qualitative data gathered during the interviews with HISP East Africa developers and focus groups with the DHIS2 core team, and during our design and development work. Given this, the empirical data had a primary role in the establishment of the design principles, while design theory and kernel theories played a justificatory role in supporting them.

In this section, I introduce the design principles and explain how they emerged from the empirical data.

Design principle component	**Description**
Aim	The aim achieved by applying the prescriptive knowledge
Context	Boundary conditions, implementation setting
Mechanism	Prescriptive knowledge on how to achieve the aim
Rationale	Descriptive justificatory knowledge providing rationale for the design principle

Table 6.10: Design principle structure.
Note. Adapted from "The Anatomy of a Design Principle" by Gregor et al. (2020), 2020, *Journal of the Association for Information Systems 21(6)*, p. 1622-1652.

6.5.1 Principle of component trustworthiness

The DHIS2 core team expressed their interest in implementing component certification to promote the components with a certain quality level. Relevant literature on component certification that I reviewed in Section 3.3.6 shows that component certification is a method of establishing the trustworthiness and quality of components. Therefore, when implementing a component repository in a software platform ecosystem, one should consider implementing component certification. Consequently, I formulate *Principle of component trustworthiness* as follows:

> **Design Principle (DP) 1 of component trustworthiness:** To increase component trustworthiness in a software platform ecosystem, consider implementing component certification because it is a means to establish component trustworthiness, increase component quality and make people more comfortable reusing components.

6.5.2 Principle of balanced certification

Given the fact that component certification was an initiative of the DHIS2 core team as platform owners, and there were plans for them to take the role of certifiers, certification could be used as a control mechanism over component reuse, and consequently, over web application development. I recommend the researchers and practitioners to look at the role of a component certifier and the nature of a certification process from a holistic perspective, assessing the possible implications of their decisions in assigning this role to balance the interests of the actors involved. Consequently, I formulate *Principle of balanced certification* as follows:

> **Design Principle (DP) 2 of balanced certification:** To increase the likelihood of component repository adoption and

cultivate its growth in a software platform ecosystem, implement component certification with a balance between control and autonomy because the right governance balance is important for the ecosystem's growth and success.

6.5.3 Principle of component discoverability

NPM Registry allows component providers to publish packages that contain only one component and packages that contain multiple components, i.e., component libraries. Landau (2021) explains that developers usually create component libraries because the creation and maintenance of an NPM package requires much effort, resulting in high overhead. Thus, it is more common to create and maintain component libraries. However, as pointed out by the DHIS2 core team, NPM Registry does not expose individual components within such component libraries to component users through its search facility, which has a negative impact on component discovery. This lack of discoverability reduces the utility of components in a component library. Therefore, to prevent a potential loss of utility and increase component discoverability in a component repository, one should provide a mechanism to index the components that are part of component libraries. Given this, I formulate *Principle of component discoverability* as follows:

> **Design Principle (DP) 3 of component discoverability:** To facilitate component acquisition from a component repository in a software platform ecosystem, aim at providing a search mechanism that can index individual components within component libraries because it is important for component discoverability and reusability.

6.5.4 Principle of installed base cultivation

My team and I have engaged with the HISP developers to learn about their software reuse practices and challenges, technology, and tools. We utilized the existing infrastructure in the form of NPM Registry and GitHub and, through our solution, addressed the identified challenges in the existing practices and tooling. According to the kernel theory on installed base cultivation in Section 4.1.2, this approach has beneficial impacts on adoption and reduces learning cost. Given this, I formulate *Principle of installed base cultivation* as follows:

> **Design Principle (DP) 4 of installed base cultivation:** To increase the likelihood of component repository adoption and encourage its growth in a software platform ecosystem, aim at cultivating the installed base (i.e., utilizing existing infrastructure) because it lowers learning cost and increases the likelihood of adoption.

6.5.5 Principle of orthogonality

My work on the project and the kernel theory on software modularity in Section 4.2.2 have led me to conclude that orthogonality is an important aspect of the architecture of a component repository that is to support multiple CBSE processes in the context of a platform ecosystem. My team and I used modularization as a means to achieve orthogonal architecture. It allowed us to work simultaneously on the same system with a high degree of independence from each other and with minimal communication. This choice was also made to make it easier to test, debug and maintain the system, as the different modules could be debugged, maintained, and tested independently. Therefore, in my opinion, it should be considered while building a similar system. I formulate *Principle of orthogonality* as follows:

> **Design Principle (DP) 5 of orthogonality:** To reduce the complexity and increase the maintainability of a component repository that supports multiple CBSE processes in a software platform ecosystem, aim to achieve a high degree of orthogonality, i.e., loose coupling and high cohesion, in the architecture of the component repository because it brings the benefits of modular software design.

6.6 Summary of the design principles

In this section, I present the established design principles (Table 6.11).

#	Design principle	Design principle specification
1	Principle of component trustworthiness	To increase component trustworthiness in a software platform ecosystem, consider implementing component certification because it is a means to establish component trustworthiness, increase component quality and make people more comfortable reusing components.
2	Principle of balanced certification	To increase the likelihood of component repository adoption and cultivate its growth in a software platform ecosystem, implement component certification with a balance between control and autonomy because the right governance balance is important for the ecosystem's growth and success.

Table 6.11 continued from previous page

#	Design principle	Design principle specification
3	Principle of component discoverability	To facilitate component acquisition from a component repository in a software platform ecosystem, aim at providing a search mechanism that can index individual components within component libraries because it is important for component discoverability and reusability.
4	Principle of installed base cultivation	To increase the likelihood of component repository adoption and encourage its growth in a software platform ecosystem, aim at cultivating the installed base (i.e., utilizing existing infrastructure) because it lowers learning cost and increases the likelihood of adoption.
5	Principle of orthogonality	To reduce the complexity and increase the maintainability of a component repository that supports multiple CBSE processes in a software platform ecosystem, aim to achieve a high degree of orthogonality, i.e., loose coupling and high cohesion, in the architecture of the component repository because it brings the benefits of modular software design.

Table 6.11: Established design principles.

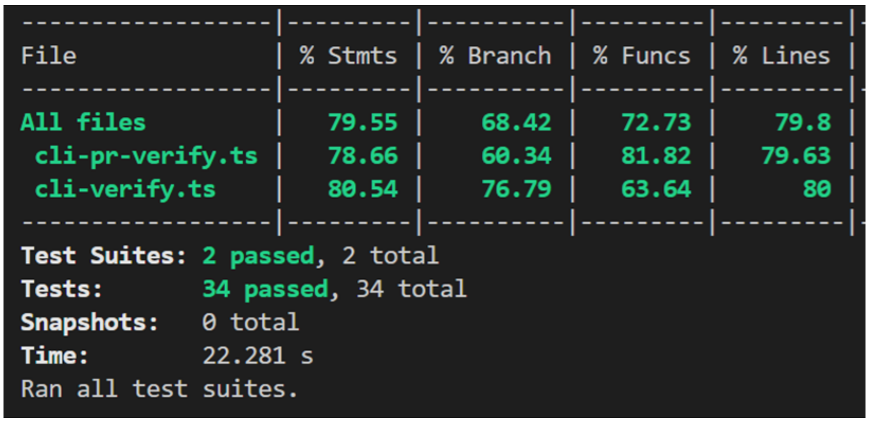
Chapter 7

Evaluation

In this chapter, I present the results of the artifact's evaluation according to the selected evaluation criteria as described in Table 5.3. These criteria are accuracy, openness, performance, efficacy, and usefulness. Additionally, I present findings that address the application of the established design principles.

7.1 Accuracy

Unit testing was used to measure the criterion of the accuracy of SCP CLI. The goal was to reach the test coverage of 70-80%. As seen in Figure 7.1, this goal was partly achieved due to total branch coverage being 68.42%, which is less than the target coverage.



```
------------------|---------|----------|---------|---------|
File              | % Stmts | % Branch | % Funcs | % Lines |
------------------|---------|----------|---------|---------|
All files         |   79.55 |    68.42 |   72.73 |    79.8 |
 cli-pr-verify.ts |   78.66 |    60.34 |   81.82 |   79.63 |
 cli-verify.ts    |   80.54 |    76.79 |   63.64 |      80 |
------------------|---------|----------|---------|---------|
Test Suites: 2 passed, 2 total
Tests:       34 passed, 34 total
Snapshots:   0 total
Time:        22.281 s
Ran all test suites.
```

Figure 7.1: SCP CLI unit test coverage report.

Unit tests were beneficial to error handling, as SCP CLI could properly handle an incorrectly set up `package.json` file during the user evaluation. Evaluator 1 made a typo in his `package.json` extension. When he submitted his package for certification, the SCP certification pipeline rejected his pull request, and the certification workflow failed. This typo was confirmed during the manual inspection of the submitted `package.json` file. The developer explained:

> This was a mistake. I didn't see my typo before your workflow caught the mistake. It's doing its job and also proves its usefulness. (Evaluator 1, personal communication, December 16, 2020)

I conclude that unit tests were beneficial for establishing the accuracy of the technical aspects of the artifact.

7.2 Openness

Evaluator 3 concluded that given all parts of SCP are entirely open-source, it is easy to inspect SCP, as users can download the source code (Evaluator 3, personal communication, January 10, 2020). Additionally, where applicable, such as in the case of the SCP Whitelist certification pipeline, the inputs for decisions are public and available to all parties, and processes are conducted in an "*open and transparent* [emphasis added] way" (Evaluator 3, personal communication, January 10, 2020). Additionally, Evaluator 3 noted that the artifact and its various parts are also open to reuse, given that they are open source, and some of the modules are highly cohesive and loosely coupled. SCP CLI, for example, could be used in a CI pipeline or even a completely different component repository. SCP Website could function without the presence of SCP CLI and SCP Whitelist, meaning that certification-related modules can be removed without having an impact on SCP Website. According to Evaluator 3, the SCP the artifact has a high degree of openness as it is open to inspection, modification, and reuse.

7.3 Performance

During the expert evaluation of SCP, Evaluator 3 remarked upon the current implementation of SCP Website with regard to its performance. He explained that the current approach of client-side indexing, where all packages are indexed for each visitor to SCP Website in their browser, provides a decent solution for a low number of packages (Evaluator 3, personal communication, January 10, 2021). However, it may put a high, or even excessive, load on external services and may also put a high load on end-users' browsers. He explained that if the amount of NPM packages becomes high enough, this indexing will likely have to be done centrally (Evaluator 3, personal communication, January 10, 2021). This could be done by having a central service with a persistent index that is used by SCP Website. This persistent index would have to be updated, either periodically or on some event trigger, to include newly published NPM packages.

7.4 Efficacy and usefulness

When evaluating SCP from the perspective of component providers, the evaluators were asked to assess how SCP affects their reusable component publishing experience. Evaluator 1 and Evaluator 3, who had previously used NPM Registry for component publishing, stated that SCP makes the experience better. Evaluator 1 explained that it is "easier to publish, easier for others to consume and contribute. cli-tool [SCP CLI] makes sure we follow the conventions." Evaluator 3 argued that "people will be more likely to notice components." Evaluator 2 stated that SCP does not change it because the evaluator has "not been a part of the publishing packages [on NPM]."

Evaluator 1 and Evaluator 3 concluded that publishing reusable components for SCP takes more or less the same amount of time compared to their previous experience with publishing reusable components. Evaluator 2 had no relevant or comparable experience. I find this result particularly interesting because component providers are required to do additional work, such as manually setting up `package.json` file in order to publish their components to SCP. However, none of the evaluators stated that it "took a little bit longer" to publish the components. It was clear for all evaluators what components should be listed in `package.json` file, and one of them explained: "it is rather clear to me that I had to list components that I expect others to use."

To investigate whether the evaluators understand the purpose of certification in the component repository, I asked them to describe their understanding of it and its purpose in SCP. Evaluator 1 explained that pre-certification checks the list of components specified in `package.json` file and checks for "typos and missing fields." He also understood that the success of pre-certification would mean that the package would pass SCP Whitelist certification pipeline. However, he did not mention two other certification checks - `npm audit` and ESLint and their purpose in certification. Evaluator 2 stated that the purpose of certification is to "to verify that the packages contains [*sic*] the required fields." Evaluator 3 explained:

> Firstly it checks the basics that is [*sic*] required for the components from my npm package to be searchable on the website. The second thing it does is to check the quality of my package.

Given this, I conclude that we have failed to effectively communicate the purpose of certification and pre-certification in SCP to our users.

Two of the evaluators stated that the component representation on SCP Website was apparent to them, while two others found it to be clear enough (Figure 7.2).

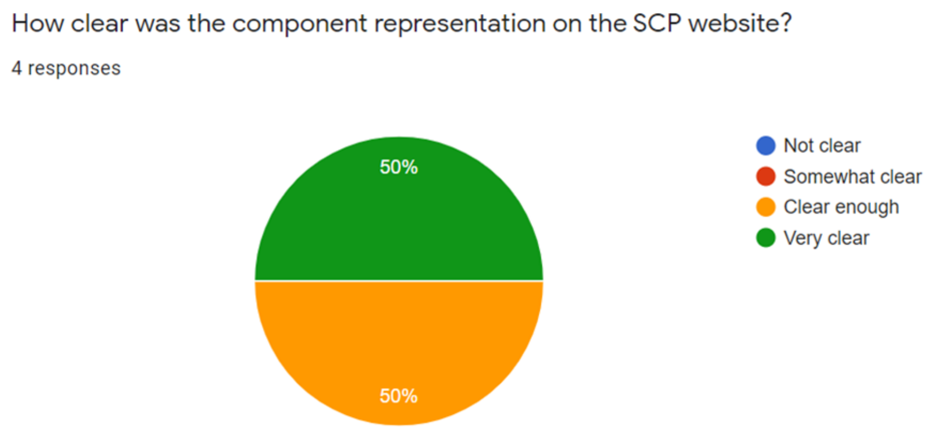


Figure 7.2: Survey result on component representation on SCP Website.

Evaluator 3 noted that "It [representation] is fairly detailed." Evaluator 4 explained that the representation is "good, but largely depending on component creator making a good description." Some component cards link to NPM modules with multiple components, which should maybe have an indication on the card." He makes a good point, because SCP relies heavily on the component descriptions provided by component providers in a `package.json` file, and if a component provider fails to describe his components, it will negatively impact component representation on SCP Website and component users experiences.

The evaluators were asked to describe their understanding of the version indicators on the component cards. Evaluator 1, Evaluator 2, and Evaluator 3 understood that the green dot represents the certified version of the package, while the grey dot represents the non-certified version of the package. However, Evaluator 2 found this representation to be "a little bit confusing." Evaluator 4 explained that "[dot] *with color* [emphasis added] showing 'current' version. I am color blind, so addional [*sic*] indicator would be useful." In my opinion, there is a clear need to re-design the component version indicators on SCP Website, also improving the color accessibility for colorblind users. While the component representation and version indicators are explained in detail in the SCP Documentation, we cannot expect that the users have already read through it when they access SCP Website to find reusable components. Given this, I conclude that we should aim at creating a more intuitive design.

The evaluators were asked to reflect on how SCP Website affects their experience of finding reusable components. Evaluator 1 explained that one "can search for components instead of libraries. more [*sic*] in tact with actual dev work." Evaluator 2 said that "SCP website would make it easier to find components." Evaluator 3 explained:

> I do not think it affects my experience at this point, since I do not think I would use it as it is today. But if more functionality

for filtering was provided as well as showing images of the components, then I think I would use it if I am looking for a new component.

Evaluator 4 expressed a desire for additional search and browse functionality on SCP Website:

Being able to reuse code for common components is a major plus. As component list grows it makes it more challenging to find what you need, so more advanced search/browse functionality would help.

To conclude, most evaluators agreed that SCP makes their component publishing and component discovery experience better and that it takes more or less the same amount of time compared to their previous experiences with component publishing. However, the evaluators either do not quite understand the purpose of certification and pre-certification or fail at explaining their understanding of it. Evaluator 3 seems to have a clear understanding of it, although it is expected because he became familiar with SCP design and architecture while performing the expert evaluation.

7.5 Evaluation of the application of the design principles

The design principles were first introduced in Chapter 6. The purpose of the evaluation was to assess the application of the established design principles by evaluation participants. In this section, I present and discuss the evaluation results of the application of the following design principles: *Principle of orthogonality*, *Principle of component trustworthiness*, and *Principle of component discoverability.* When the evaluation was performed, it was not possible to assess the application of *Principle of balanced certification* because we have not established who would take a role of a certifier, what certification criteria would be used, and what certification requirements would entail. Additionally, the application of *Principle of installed base cultivation* should be assessed by the target user group, the developers from HISP East Africa, which was not possible to do during this formative evaluation due to the difficulties establishing contact.

7.5.1 Application of Principle of orthogonality

To assess the SCP's orthogonality, I have chosen expert evaluation as an evaluation method because it is a non-functional architectural characteristic that cannot be assessed by the users of SCP.

The expert evaluator's assessment of SCP Website's efferent dependencies has shown that the usage of UNPKG REST API and NPM Registry API is not dependent on the specific implementations but only dependent on the protocols which are stable and well-defined (Evaluator 3, personal communication, January 10, 2021). The expert evaluator argues that SCP Whitelist is tightly coupled with GitHub, as it is implemented on top of existing GitHub functionality, specifically the pull request system and GitHub Actions (Evaluator 3, personal communication, January 10, 2021). According to Evaluator 3, there is no separation between SCP Whitelist and GitHub, no interfaces that can be reimplemented. The benefit of this design choice is that it reduces the overall footprint, complexity, and maintenance burden of SCP. Therefore, according to Evaluator 3, it seems like a reasonable or even preferable design choice. Additionally, SCP Whitelist uses SCP CLI by calling it as a command. This can be considered loosely coupled as this usage is not dependent on the SCP CLI implementation and the SCP CLI implementation can be changed without having to change SCP Whitelist repository. The expert's assessment of SCP CLI coupling shows that SCP CLI interacts with UNPKG through REST API, and GitHub through the Git protocol. The usage of these systems is not dependent on the specific implementations but only depends on the stable and well-defined protocols. This design is optimal and qualifies as loose coupling. While assessing the degree of cohesion of SCP CLI, the expert evaluator concludes that all functions of this module relate to validation and certification of an NPM package (Evaluator 3, personal communication, January 10, 2021). They are usable and relevant to component providers in their workflows for developing, maintaining, and publishing NPM packages containing reusable components. Additionally, SCP Website can be considered highly cohesive because its functions fit well together and have a unified purpose, which is to allow component users to find the components published according to the SCP guidelines. Similarly, SCP Whitelist repository can be considered highly cohesive as all of its functions fit well together and have a unified purpose, which is to host the SCP Whitelist and facilitate changes to it.

To summarize stated above, my formative evaluation suggests that we have achieved a moderately high degree of orthogonality in SCP and that this has resulted in a simple and maintainable system.

7.5.2 Application of Principle of component trustworthiness

Evaluation of SCP from the perspective of certifiers was performed by Evaluator 1 and Evaluator 3. First, when asked to elaborate on their understanding of the purpose of component certification, Evaluator 1 stated that "[certification] makes sure only high quality non-malicious convention-following packages are marked as [certified]." Evaluator 3 explained that certification is necessary to "establish that the component had some baseline of quality." Further, the participants were asked to give their opinion on the fact that SCP Whitelist certifies only the individual versions of the packages. Evaluator 1 explained:

> Good approach. Lots of stuff break or gets worse over time. We

> can only verify what we have here and now.

It should be pointed out that earlier, during one of the focus groups with the DHIS2 core team, the core team developers argued that there was no need to implement support for specific versions of a package (Section 6.2.2). Despite this, we have chosen to implement it, and it seems like Evaluator 1, a member of the DHIS2 core team, found this solution acceptable. Evaluator 3 noted that it "makes sense" because it is not clear how one deals with older and newer versions of the packages. He makes a good point because certifying a package without considering its version can be risky, as there is no guarantee that the older or newer versions of the package are of an appropriate level of quality and trustworthiness. Additionally, the participants were asked what certification checks should be part of the certification process. Evaluator 1 has listed the following checks:

- usefulness
- predictable behavior (do the components behave as documented)

While predictable behavior is something that has been mentioned in the reviewed literature on CBSE and is an aspect of component trustworthiness, the criterion of *usefulness* is not a known quality attribute. It is also unclear what metrics can measure this criterion, as it is very subjective. Another concern I have is whether it should be up to certifiers to decide what components are useful and which are not. In my opinion, certifiers do not possess knowledge about the prospective use context of every component and therefore cannot determine their usefulness beforehand. I suggest that component usefulness should be determined by component users when they are engaged in component acquisition. Evaluator 3 suggested such certification checks as dependency analysis or software composition analysis, software management analysis, code analysis, code style analysis, and automated test analysis.

The participants were asked to provide suggestions for the improvements of the certification workflow. Evaluator 1 stated:

> I think it's quite nice. The PR [pull request] *acts as a discussion platform* [emphasis added] when we maintainers want the component [provider] to make changes also puts the whole thing [certification process] *open and public* [emphasis added].

He seemed to like the open nature of the certification process in SCP Whitelist. Additionally, he suggests that there is a possibility for component providers and certifiers to engage in an open discussion as part of the certification process. In my opinion, the openness of the certification process makes certification transparent to both component providers and component users, making it more clear what this process entails. Evaluator 3 noted:

> If there is a way to make it clearer on the pull request pages itself that warnings occurred it would be nice, not sure if Github

> actions provides such a mechanism. Overall though I really like the way it works.

Furthermore, I asked the participants to reflect on the nature of the component certification process. They were asked whether the package certification process should have minimal human discretion and thus be objective or subject to certifiers' opinions. The reported results showed diversity in opinions. Evaluator 1 noted:

> I think it should be subjective. Otherwise we'll end up with many [obscenity] libraries with a [certified] tag. The discussion of approval *should be public* [emphasis added] (the PR thread).

Evaluator 3, on the other hand, suggested minimal human discretion:

> I think it is best to have minimal human discretion. That way [certification] can be quicker and more predictable.

None of the evaluators have experienced any issues during the evaluation. To conclude, the evaluation results show that both evaluators find the certification of individual versions of the packages as a good approach. Furthermore, both evaluators like the architecture of the certification workflow and its openness. However, it is also clear that the certification process and its nature still have some challenges that we have not managed to address adequately.

In summary, my formative evaluation suggests that the certification mechanism in SCP is reasonably good means to establish component trustworthiness and make people more comfortable with reusing components.

7.5.3 Application of Principle of component discoverability

When reflecting on the component discoverability of SCP, Evaluator 1 explained that "when developing you're looking for components that do what you want the component is tied to the library, not the other way around." Evaluator 3 noted that "it is nice to have a place where all components are searchable." Evaluator 4 explained that "depending on use case, but having a uniform way to present information of a specific type across multiple different systems is very valuable." Evaluator 2 chose not to answer this question. Given this, the consensus amongst evaluators seem to be that we managed to address the issue of component discoverability.

In summary, my formative evaluation suggests that the search mechanism in SCP makes it easy to find and subsequently acquire individual components that are part of component libraries.

Chapter 8

Discussion

This chapter will present and discuss the established set of design principles considering the empirical data of this study and evaluation results and elaborate on how the design principles inform the existing research.

8.1 Design principles

8.1.1 Summary of the design principles

The following research question was asked: *What are the essential design principles for implementing a component repository that facilitates component reuse in a software platform ecosystem?* To provide the answer to the research question, I presented a set of the established design principles (Table 6.11). Further in this section, I will discuss each of them to underline the significance of my findings in light of the existing research.

8.1.2 Principle of component trustworthiness

Previous research shows that component trustworthiness is a significant challenge in CBSE and some researchers, inter alios Crnkovic and Larsson (2002), Heineman and Councill (2001), Sommerville (2011), Tiwari and Kumar (2020), indicate the importance of certification as a means for component users to determine the level of quality and trustworthiness of a component.

Given my research question, it is important to establish why this principle addresses the particular context of a software platform ecosystem. In the case of DHIS2, component reuse can take place at an intra-group level, within an individual HISP group, meaning that developers in a certain HISP group create reusable components for their own use. Component reuse can also take place within the greater HISP community, at an inter-group level, meaning that developers from one HISP group use components made by another HISP group or made by the DHIS2 core team. The case when the HISP developers reuse components developed by third-parties that are not part of DHIS2 ecosystem can be viewed as Internet-wide reuse. In the first case, intra-group reuse, more mechanisms are available to control the quality of the components being made and reused, as HISP groups are unified organizations and can institute organizational policies, for example, mandating that code must be reviewed and require automated tests with some specific minimum code coverage. In the second case, inter-group reuse, even though reuse is happening within the HISP community - at the ecosystem level, there can be different quality outcomes because the HISP groups are inherently independent organizations with different policies. The same concern would apply to the case of Internet-wide reuse. The developers in one HISP group may be reluctant to reuse the components developed by a different organization, as indicated by Jalender, N.Gowtham, et al. (2010) in Section 3.3, especially if they are unsure if these components will meet their quality expectations. Therefore, establishing a means to determine the level of component trustworthiness and quality is critical for facilitating component reuse in cases other than intra-group reuse, which includes reuse in a software platform ecosystem.

Considering that certification is still a challenge and that there is a lack of standard procedures and methods for certification, I contribute to the body of knowledge by providing insights and recommendations regarding certification functionality and certification architecture by presenting our implementation of certification.

First, the certification process should consider component versioning. As Szyperski (2002) notes, there is a need for proper component versioning to ensure component compatibility. Additionally, Tiwari and Kumar (2020) argue that a component repository should support different versions of different components and handle proper version updates. Furthermore, our research also showed that taking component versioning into consideration can be useful for certification. Certification done on a version-specific basis may require more time and effort because later versions of components would have to be certified again. However, if certification is done on a per-component basis without considering the version, then regressions of the component quality in future versions of a component can call the certification's validity into question. Even though the practitioners indicated that there is no need to implement version-specific certification given its drawbacks mentioned above, my evaluation showed that version-specific certification is a good and preferable solution. It is also a safer choice for component certifiers, as they cannot guarantee that the older and future versions of a component maintain the same level of quality. Our implementation of version-specific certification also addressed the challenge mentioned by Mohammad (2011) in Section 3.3.6 - component modification after certification. NPM Registry, one of the existing systems that we used as a basis for SCP, prevents published NPM package versions from being changed after they were published (*npm-publish*, n.d.); this restriction is also present in SCP itself, thus ensuring that versions we certify do not change after certification was performed. Given this, it is important to implement version-specific certification when implementing a component repository with certification and ensure that specific component versions are immutable and cannot be changed once published.

Another question raised by Mohammad (2011) in Section 3.3.6 is how component modification affects certification. In our case, given the immutability of component versions, a change in a component results in an entirely new entity that has to be certified again. However, our approach may lead to significantly more certifications being conducted. Practitioners could consider some middle ground between certifying specific component versions and certifying components without consideration of versions. One option would be to certify according to specific elements of a component's version number, provided semantic versioning adheres to (*About semantic versioning*, n.d.). They could for example certify a component for a specific `Major` and `Minor` version elements, such as 1.3, and then all `Patch` versions with the same `Major` and `Minor` version elements would be considered certified, so 1.3.0, 1.3.5 and 1.3.50 would all be certified. However, a problem with this approach is that there is no guarantee that component providers correctly follow the semantic versioning guidelines, and this approach also has no protection against malicious actors which may intentionally break semantic versioning guidelines.

Furthermore, one should consider implementing functionality that gives certifiers a mechanism to directly integrate with the development and continuous integration flow of component users. This is especially important when certifiers want to revoke a component's certification status and make component users aware of the revocation. In our case, we considered creating an NPM scope maintained by certifiers. Certified components would be republished within such scope, guaranteeing that the certified components have not been modified after certification. This would also give certifiers a way to revoke the certification status of components so that component users would notice revocation without requiring the use of specialized tools or procedures such as checking SCP Whitelist. Usage of an NPM scope or other similar functionality could also provide a mechanism to ensure the immutability of component versions. This insight contributes to the future research suggested by Mohammad (2011) discussed in Section 3.3.6.

Additionally, one should consider implementing a certification process that is open and transparent to component providers, and even component users. SCP Whitelist certification workflow is open to anyone who is registered on GitHub and, as a DHIS2 core team developer has mentioned during evaluation, it acts as a platform for discussion for the involved actors. It can also be beneficial to component users, as they can gain knowledge on how the certification requests are being assessed and what assessment metrics are used as a basis for certification. This openness can provide all users with much deeper insight and practical knowledge that they could not easily obtain from the documentation. Openness could also lead to more innovation as a deeper awareness and understanding in the community could give users new ideas and inspirations on improving and optimizing various aspects of the system.

Certification requirements should be communicated to component providers to ensure they can create the components that comply with these requirements. I suggest that certifiers clearly define quality attributes and metrics; they should also analyze the nature of the metrics, i.e., examine the level of human discretion the metrics require and how much automation can be done for assessing the metrics. Automated checks are faster and more accurate if implemented correctly, and they also require fewer of certifier's resources. These automated checks with minimal human discretion allow the implementation of a more objective certification process, enabling the possibility to provide more rigorous and accurate pre-certification functionality for component providers. This means component providers could perform the same certification procedures locally to determine if their components are compliant with certification requirements before they proceed with publishing and submission for certification. Metrics that require some level of human discretion and cannot be fully automated and implemented as part of pre-certification should be communicated to component providers in another manner, appropriate to the nature of the metrics. One good option would be to clearly document the metrics that require human discretion so that component providers have explicit knowledge of these criteria and give them the ability to perform a self-assessment for these criteria. In order to communicate the certification process's nature, the Certification Process Classification Model (Figure 6.1) can be used.

To sum up, when implementing a component certification, one should issue the certificates for specific component versions, and these versions should be immutable. Additionally, the implemented certification process should be open and transparent and also automated where it is possible.

8.1.3 Principle of balanced certification

A certifier's role is an aspect of the certification process that should be considered in the context of a software platform ecosystem. Certifiers specify quality attributes and metrics and define what characteristics a trustworthy high-quality component should possess in order to be certified. These decisions have implications for component reuse and web application development in a software platform ecosystem. First, as several researchers point out, certification should be performed by independent assessors. Given this, because the HISP groups are independent organizations, one can assume that certifiers from one group could certify the components developed by another HISP group. However, ecosystem actors might pursue their self-interest, and it is unclear how these can be set aside in the context of certification. For example, the DHIS2 core team could act as an independent assessor of the components developed by one of the HISP groups in HISP East Africa while pursuing their own interests with regard to development practices in the community. One way the DHIS2 core team could pursue their own interest is by limiting the diversity of practices by promoting the components built with the web development framework preferable to them.

Since it was the DHIS2 core team, as platform owners, who expressed their interest in component certification and taking the role of certifier, I will now bring my own critical perspective on this situation and discuss it in the light of theory on governance and control in a software platform ecosystem.

Platform owners cannot exercise control over application development in the ecosystem using development boundary resources in the same manner it can be done through application boundary resources. SCP is inherently a development boundary resource, meaning that it does not provide any direct interaction with the platform core, and our end-users are not required to use it to access the functionality of the DHIS2 platform. If component providers perceive that certification requirements limit the diversity of their practices and affect their development experiences negatively, they might avoid submitting their components for certification, and, in the worst-case scenario, abandon SCP. This would also negate any potential benefits that component users could gain from SCP as some level of certification is better than no certification.

The role of a component repository as a development boundary resource can be further explored with regard to governance and control in the ecosystem. Tiwana (2014) explains that third-party application developers in an ecosystem are independent of a platform owner; therefore, good governance in a platform ecosystem should respect developers' autonomy and should be done through shaping and influencing, rather than direct control. Tiwana (2014) distinguishes between two types of decision rights in a platform ecosystem. The first is *platform decision rights*, which cover the decisions pertaining to the platform, while *app decision rights* cover the decisions pertaining to the applications that extend the platform (Tiwana, 2014). It should be noted that *app decision rights* do not sit exclusively with app developers; some of these rights can be held by platform owners (Tiwana, 2014). SCP, as a development boundary resource, is meant to support web application development; therefore, platform owners could use certification as a means to extend their *app decision rights*. Further, according to Tiwana (2014), decision rights can be either *strategic* or related to *implementation*. Tiwana (2014) explains that *strategic* decision are meant to set a direction, while *implementation* decision rights are about functionality, UI, features pertaining to applications. Governance over the reusable components can thus further be categorized as *implementation* decision rights.

Tiwana (2014) argues that platform owners can take in use three formal control mechanisms, which are *gatekeeping*, *process control*, and *metrics*. The first control mechanism, gatekeeping, is app-centric, meaning that platform owners decide what applications can be allowed to a platform ecosystem. This type of control mechanism could be exercised in a component repository, for example, by limiting component storage only to certified components. According to Tiwana (2014), there are three pre-requisites for this control mechanism:

1. Platform owners must have a certain level of competency to judge
2. Evaluation should be fair and fast
3. This control mechanism should be accepted by application developers

Given this, it is not guaranteed that this control mechanism will produce expected results as platform owners have no control over component reuse and what type of components are used in web applications. Application developers do not have to accept this control mechanism. However, this control mechanism could potentially be effective if application developer interests are taken into considerations.

The second control mechanism, process control, refers "to the degree to which a platform owner rewards or penalizes app developers based on the degree to which they follow prescribed development methods, rules, and procedures that it believes will lead to outcomes desirable from a platform owner's perspective" (Tiwana, 2014, p. 124). Component certification process could allow platform owners to exercise this control mechanism, for example, by issuing component certificates to the components that adhere to a certain level of quality. Search functionality in a repository could be constructed in such a way that promotes certified components by making them more visible to component users. This mechanism has two pre-requisites:

1. Platform owner must have the expertise to be able to instruct application developers
2. Work conducted by application developers has to be monitored and observed

Given this, with regard to component reuse, platform owners must have knowledge about component reuse practices in the ecosystem and be able to give instructions that could improve this process. It would be resource-intensive to monitor CBSE for reuse process, but certification can be used to verify whether the components were developed according to a set of requirements provided by platform owners.

The third, metric control mechanism is defined as "the degree to which the platform owner rewards or penalizes app developers based on the degree to which the outcomes of their work achieve predefined target performance metrics" (Tiwana, 2014, p. 124). Tiwana (2014) discusses mainly app-level metrics, such as performance and memory utilization, and market-oriented metrics, such as downloads and user ratings. With regard to component reuse, it would be possible for platform owners to introduce component-oriented metrics for the purpose of certification. They would be similar to app-oriented metrics, just at a different level of abstraction. Tiwana

(2014) claims that for this mechanism to be viable, those metrics must be set by platform owners, they should be pre-defined and objectively measurable. In my opinion, there are similarities with *gatekeeping*, as this control mechanism would not be viable without clearly specified objectively measurable metrics.

Furthermore, Tiwana (2014) argues that there is a difference between "control attempts" (p. 126) by platform owners and their actual ability to realize these control attempts. Tiwana (2014) claims that for a control attempt to be successful, two requirements must be met. First, prerequisites pertaining to each control mechanism must be met. Second, the established control mechanism must be perceived by application developers as "legitimate, fair, and reasonable" (Tiwana, 2014, p. 126) and require "the consent of the governed" (Tiwana, 2014, p. 126).

Considering the aspects discussed above and the fact that platform owners cannot exercise direct control over web application development, and specifically component reuse in our case, I conclude that platform owners should adopt rewarding techniques as a way to influence component reuse practices rather than trying to enforce direct control through penalization. I suggest that platform owners, who have an interest in being in charge of component certification, design the process in collaboration with component providers, and define clear and objective metrics to perform a fair certification.

8.1.4 Principle of component discoverability

This study's empirical data showed that the component repository that is already in use by some parts of the HISP community, namely NPM Registry, allows components to be stored individually or as component libraries containing multiple components. However, NPM Registry does not provide a mechanism for searching through individual components within component libraries, which means it is not properly cataloged as explained in Section 3.3.5. Component discoverability is essential for the component acquisition process that includes component search and component selection. If a component repository's search functionality does not make it possible to search through all the available components in a repository, including the ones within component libraries, then component users might end up with fewer candidate components to choose from. This has a negative impact on component reuse by reducing the visibility of potentially suitable components.

When component reuse is happening in-house, i.e., within one organization, it is easier for component users and component providers to have an overview of the components that are available for reuse. An organization, in our cause, an individual HISP group, could use an established set of tools and technology for component management and follow some common standard for designating components made by the organization to facilitate component discovery. For example, as was discussed in Section 6.2.1, one of the HISP groups we collaborated with uses GitHub to store their source code and an organizational scope on NPM Registry for component storage and designation. On the contrary, when component reuse happens on the ecosystem level, developers do not possess the same knowledge about the components developed by others and do not necessarily know what designation mechanisms the other groups use; thus, they have to rely on search functionality provided by component repositories or search engines. Similar problems exist on an Internet-wide level of component reuse. For example, a popular React UI framework Material-UI has a vast number of reusable components, such as grids, button groups, sliders, progress bars, etc. However, one cannot discover those individual components using the current search functionality of NPM Registry. To discover them, I would need to open the Material-UI homepage and look for components in there. To conclude, this principle is relevant for component reuse in general but has higher importance for component reuse happening within larger organizations with distributed teams, within platform ecosystem, and Internet-wide, as these contexts are where discoverability is the biggest problem.

8.1.5 Principle of installed base cultivation

While platform owners are in control over some specific parts of a platform ecosystem, such as a platform core, they do not have the ability to directly control third-party application development and mandate what development boundary resources are to be used. Given this, the adoption of a component repository in a platform ecosystem cannot be done through a planned and controlled action, and therefore needs a different approach to innovation, such as installed base cultivation presented in Section 4.1.2. Our practical aim was to build a component repository to support the existing CBSE process in the DHIS2 platform ecosystem. These existing processes, such as component reuse, are performed by people, shaped by practices, governed by standards, and supported by some specific technology and tooling. Within a platform ecosystem, these aspects constitute an installed base; therefore, when designing and developing a component repository, this installed base should be considered.

We applied this principle during the design and implementation of SCP. First, in the initial stage of our project, we established our target user group, which are two HISP groups in HISP East Africa and the DHIS2 core team. To drive the adoption of SCP, we developed it in such a way that addresses the challenges we had identified during the interviews and focus groups, which was explained in detail in Chapter 6, thus making the artifact beneficial to them and *designing initially for usefulness*, as suggested by Hanseth and Lyytinen (2010). Furthermore, we have considered the technology and tooling that is already in use and *built SCP upon the existing installed base*, because, according to Hanseth and Lyytinen (2010), it decreases adoption barriers and learning costs. These strategies aimed to address the bootstrap problem discussed in Section 4.1.2.

To address the adaptability problem and accommodate the future growth of SCP we have adopted modularization. According to Hanseth and Lyytinen (2010), such an approach helps to achieve stability in the system, reducing error propagation between the modules. Our modular architecture with small modules and simple interfaces also makes it simple for future maintainers to organize their IT capabilities.

8.1.6 Principle of orthogonality

When designing SCP, we aimed to achieve an orthogonal system with logical architecture to embrace the benefits offered by modular software design discussed in Section 4.2.2. This approach was beneficial to us during our development process, allowing us to work independently on different modules of SCP, lowering the risk of collisions and error propagation. Additionally, modularization made it possible for us to deepen our specialization in specific parts of the system without requiring us to be familiar with all the other modules of SCP. It allowed us to be more efficient overall when making a contribution.

Our goal was to ensure that each module of SCP would be loosely coupled and have a single and well-defined purpose. According to Ingeno (2018), the functionality inside each module should be related and contribute to its established purpose. SCP Website's primary purpose is to aggregate reusable components and provide an interface for component acquisition by component providers. SCP CLI module provides package validation and pre-certification functionality to component providers, and this module is also used in SCP Whitelist for the purpose of certification by certifiers. The purpose of SCP Whitelist is to store the list of certified components, and facilitate component certification using GitHub Actions. Ingeno (2018) discusses that modules with low cohesion serving many different purposes are less likely to be reused, while "a module that works together as a logical unit with a clear purpose is more likely to be reused" (p.172). This is in line with our experience when developing the artifact as it seemed natural to us to reuse the SCP CLI in two contexts that required the same functionality that was in line with the highly cohesive purpose we intended for it.

When developing a component repository for a software platform ecosystem, one problem that must be addressed, according to the literature reviewed in Section 4.1.2, is the adaptability problem. Specifically, this problem is that solutions must be technically and socially flexible to adapt to a growing user base. Hanseth and Lyytinen (2010) prescribe modularization as a way to address the adaptability problem. Designers are encouraged to develop loosely coupled modules to accommodate the growth of the system. A high degree of orthogonality has a significant impact on the system's evolution, as each of the modules can evolve in a decentralized way, i.e., the modules can be modified, updated, and removed independently of each other. This provides technical flexibility, but it also enables social flexibility. An orthogonal solution could allow ownership and maintenance of various modules to be allocated to different ecosystem actors. For example, in the case if SCP, while SCP Website could be maintained by third-party application developers, the DHIS2 core team could act as certifiers and take over maintenance the certification related modules.

The expert evaluation showed that the modules of SCP are highly cohesive. SCP Website and SCP CLI are also loosely coupled, as they are dependent on the stable and well-defined protocols. However, SCP Whitelist and its certification workflow are tightly coupled with GitHub, a design choice we have made considering all the benefits of GitHub's functionality.

8.2 Incentives for component reuse in the DHIS2 ecosystem

Szyperski et al. (2011) explain that the creation of reusable components is a demanding and time-consuming task because, contrary to bespoke solutions, components have to be generalized in order to be used in different contexts. Additionally, a component provider has to create a proper component specification, documentation, and various tutorials. Given this, "components are thus viable only if the investment in their creation is returned as a result of their deployment" (Szyperski et al., 2011, p. 17). I will now discuss component reuse in the HISP community and elaborate on the possible return of investments.

Component reuse in the HISP community can take place on three levels. The first level is in-house reuse, which means that components are created and reused within one organization. In our case, this would be reuse within an individual HISP group. As Szyperski et al. (2011) note, in-house reuse brings the benefits of component-based software engineering, such as reduced time to market, increased maintainability, and configurability. Additionally, with in-house component reuse, the incentives of component providers and component users are aligned, as the benefits of component-based software engineering accrue to the same organization that invests in creating reusable components. Furthermore, with in-house reuse, trustworthiness is less of a problem than it is with the reuse of components created by third parties because it is easier to implement uniform standards and quality control within an organization. As Sommerville (2011) notes, some companies create their own internal component repositories and exclusively use the components they have developed, avoiding third-party components. One of the HISP groups we collaborated with publish their components within an organizational scope on NPM Registry. Using a component repository for in-house reuse facilitates better component distribution, cataloging, and discovery. There are, of course, large organizations with distributed teams. However, contrary to ecosystems, it is typical for organizations to exercise a command-and-control management strategy, making it easier to facilitate component reuse even within large organizations.

The second level of component reuse occurs at the ecosystem level. This happens when one organization within an ecosystem reuses components created by another organization within the same ecosystem. I will refer to this level of reuse as inter-group reuse, as these organizations are HISP groups in the case of the HISP community. This study aimed to create a component repository to facilitate component reuse on the ecosystem level, aiming to increase the utility of components developed by individual HISP groups by making them more accessible to other developers that are part of the DHIS2 ecosystem. Previous literature on CBSE sees component markets as a means to gain return on investment (Crnkovic & Larsson, 2002; Kotonya et al., 2003; Seacord, 1999; Szyperski et al., 2011). The potential for profit from selling components on such markets can incentivize component providers to invest their resources in the creation of reusable components. However, if the components are open source, as is the case for the DHIS2 ecosystem, this incentive is no longer applicable. Strictly defined markets involve the exchange between parties, whether this exchange is for money or something else (CFI, 2021). However, many markets involve exchanging goods for money and are closely tied to a monetary profit incentive. By virtue of being a B2B transaction platform, our artifact is inherently a marketplace for component exchange, just without an exchange for money and thus without a direct monetary profit incentive for component providers. Szyperski et al. (2011) argue that components within markets will exist only when component providers and component users will "join forces (...) to reach a 'critical mass'" (p. 17). What he discusses is essentially indirect network effects, one of the important features of transaction platforms. Tiwana (2014) notes that if a platform is two-sided from the start, one has to attract both user groups to get the platform "off the ground" (Tiwana, 2014, p. 145). Given this, to drive the success of a component repository, one has to attract both user groups. This, in turn, raises the question, what are the incentives for component reuse on the inter-group level compared to the intra-group level?

Component users would gain access to a larger number of components, which could positively impact the process of component selection, and make web application development process more effective, as fewer components would be written from scratch. Inter-group reuse is more demanding for component providers because they would spend more time creating a component specification, documentation, and tests. Although, component providers would benefit from it if their components are exposed to and used by a larger number of developers. It increases the chance for component users to notice bugs and other issues, report these issues to component providers, and even help with bug fixing, maintenance, and further improvements of a component, thus reducing the maintenance burden and improving the quality of the components. This is, of course, predicated on component users also taking a role as component providers, i.e., not just reusing components, but also developing them.

Another incentive for component providers in inter-group reuse is developer reputation. Cai and Zhu (2016) state that a large number of studies explore developers' motivation within an open-source project and some of these motivations are altruism, career opportunities, enjoyment, and a "desire for reputation" (p. 103). *Reputation* is defined as "a distribution of opinions, estimations, or evaluations about an entity, such as people, item and organization, in an interest group" (Cai & Zhu, 2016, p. 103). Cai and Zhu (2016) explain that it is beneficial for developers in an open-source community to work with someone intelligent and experienced. Several aspects can increase a developer's reputation: his coding performance and code quality, community experience, and collaboration experience (Cai & Zhu, 2016). Developers that create high-quality components and expose these components to other developers in a platform ecosystem could gain an increase in reputation both within a community and more universally among all software developers. Community interaction and collaboration experiences, such as participating in discussions and helping other developers, and participating in various inter-group projects, could also benefit developers. As Cai and Zhu (2016) mention, collaboration experience can also increase trust between developers, indirectly affecting component trustworthiness. Component trustworthiness is a particular challenge in CBSE, and even more so on the ecosystem level, as the developers may not be willing to reuse the components developed by third parties. If a component provider becomes known in the ecosystem for delivering high-quality code and has been successfully collaborating with other developers, components developed by him would be more trusted.

The third level of reuse, Internet-wide reuse, happens when the developers reuse components developed by third parties that are not part of the DHIS2 ecosystem. NPM Registry could be viewed as a component repository part of the Internet-wide reuse, as it is entirely open to anyone who wants to publish their components and contains a large number of components that were not developed for DHIS2. The incentives and challenges of Internet-wide reuse for component users and component providers are similar to the incentives offered by inter-group reuse, with some notable differences. The advantage of Internet-wide reuse is that component users gain access to an even larger number of components to choose from. However, many of these components are created for different domains and may not be suitable for reuse in web applications in the context of a health management information system. Moreover, different component providers may make use of different services to store their reusable components, and it may become difficult for component users to discover and acquire these components. As I have mentioned in Section 6.2.1, the developers we collaborated with mainly look for components through the NPM Registry, a component repository they use in their practice, and through Google Search. The fact that they use Google search shows that the HISP community neither has a common component repository nor a common designation system for their components, i.e., a means of making it clear that their components were developed for DHIS2. Additionally, with such a large number of components, the success of the component acquisition process is largely dependent on the functionality provided by NPM Registry and the quality of specification of the published components. A component developed by HISP that is poorly documented and specified could become lost among all the other components in large Internet-wide repositories like NPM Registry. Additionally, component trustworthiness is even more of a challenge in Internet-wide reuse. From the point of view of component providers, there is more potential for sharing the maintenance burden in Internet-wide reuse as there are more developers on that level than on the inter-group level, but most of these developers are not likely to be interested in components designed for DHIS2 or components relating to health management information systems. What matters to component providers is not just that their components are available to the largest possible audience, but that it is available to those operating in the domain the components were developed for, and that they stand out more inside that domain. SCP thus creates a greater incentive for component providers than the one that exists on Internet-wide level reuse.

To summarize what has been said, component reuse can occur on three levels: in-house level reuse (intra-group reuse), ecosystem-level reuse (inter-group reuse), and Internet-wide reuse. It is easier to make components trustworthy within one organization, as more strategies are available to organizations for implementing uniform standards and quality assurance. Ecosystem-level reuse can be more demanding for component providers, as the creation of high-quality components is time-consuming and resource-intensive. On the other hand, ecosystem-level reuse exposes components to a larger audience within the same domain, potentially leading to increased quality and reduced maintenance burden. Thus, ecosystem-level reuse could have positive implications for developers' reputations. Internet-wide reuse has the potential for similar benefits as ecosystem-wide reuse; however, this level has drawbacks for more domain-specific components due to poor cataloging functionality in some internet-wide component repositories.

8.3 Research validity

Design Science Research establishes the evaluation of DSR criteria as a mechanism to assess the validity of the contribution. My evaluation shows that the artifact we built through the application of the established design principles is efficacious, useful, open, sufficiently performant for its current life-cycle phase, and accurate in aspects evaluated for accuracy.

8.4 Research limitations

In this section, I will discuss the identified limitations of my research.

8.4.1 Lack of time

As students, we had limited time to conduct our research. DSR includes many activities for researchers to engage in, often simultaneously. We had to arrange data gathering activities, collect and analyze the gathered data, and design and develop our artifact. As a consequence of the COVID-19 pandemic, digital collaboration with practitioners slowed us down, as it sometimes took weeks before we would get any reply. Additionally, it took a long time to arrange the interviews and focus groups. We had to stop the development process earlier than planned so that we could perform an evaluation and then communicate our research. DSR is an appropriate research methodology when researchers aim at solving real-world practical problems; however, due to limited time and resources, we were unable to go through all the activities in the DSR cycle, i.e., putting the artifact in production and conducting an *ex post* evaluation.

8.4.2 Limited access to data

We collaborated with two HISP groups in HISP East Africa, and they were our target user base during this study. We tried to reach out to several other HISP groups, but did not manage to recruit more participants. Considering that the HISP community is large, we were not able to cover all the practices and aspects with regard to software reuse prevalent in the community. Besides, digital collaboration excluded the possibility for conducting fieldwork and the possibility to use other qualitative methods, such as participant observations.

8.4.3 Design principles credibility

When assessing the validity of Design Science Research, it is essential to consider not only the truth and trustworthiness of the findings that guide the design and development of an artifact but also the utility of the artifact (Iivari, Hansen, & Haj-Bolouri, 2018). An ideal outcome of the DSR project is putting the artifact into production and assessing its utility in a naturalistic setting with the target user base. Due to the lack of time and resources, we could not reach this goal; thus, this could limit the credibility of the established design principles. As I have previously mentioned in Chapter 7, it was not possible to assess the application of the installed base cultivation strategy. It would require assessing the artifact in use by our target user group, i.e., how it integrates with the existing technology and practices and whether we have succeeded in addressing the previously identified problems. Additionally, there were plans that the DHIS2 core team would be involved in the implementation of the certification process and take the role of certifier. Unfortunately, these plans were not fulfilled. Therefore I discuss what implications the DHIS2 core team being certifiers have based on my interpretation of the gathered data during focus groups and evaluation.

8.4.4 Design principles limitations

DHIS2 is open source software developed on a non-profit basis and is targeted at a global user base. Additionally, DHIS2 is an enterprise software platform with multiple instances which are subject to in-country ownership. The design principles I established are intended to be relevant to practitioners developing a component repository in the context of a platform ecosystem. However, these principles were developed in the context of DHIS2. As such, they may be biased towards this context and may have limited applicability to other types of software platform ecosystems.

8.5 Reflection on team management and group work

This section provides a reflection on team management and is partly based on the discussions I had with Håkon André Heskja. First, I will reflect on the aspects of the group work that, in my opinion, turned out well.

Successful collaboration during the COVID-19 pandemic. Because of the coronavirus, many universities worldwide had to establish new study routines and rely heavily on digital teaching. The coronavirus affected our daily activities, and we had to find good ways of working together and tailor the communication practices of our team to make sure the coronavirus would have minimal impact on our project and its overall quality. We collaborated on code using Git and GitHub, we arranged our meetings on Zoom, and used Slack and Mattermost for daily communication. Our data collection activities had to be remote due to coronavirus-related restrictions and relied heavily on digital tools. Despite these challenging times, we have done a significant amount of work and completed the planned activities.

Modularization in software development. We adopted modularization to achieve a high degree of independence. This approach also allowed individual team members to make decisions that are limited in scope to the implementation of a module without the other team members' involvement. Separation of the SCP modules into different GitHub repositories helped us to reduce and avoid collisions during the development of SCP.

I have assessed our group work, and the following challenges were identified: *self-managing team and lack of leadership*, *avoidance of accountability*, *lack of formalized development process*, and *limited use of collaboration and communication tools*. I will now discuss these challenges and explain what measures were taken to overcome them.

Self-managing team and lack of leadership. In my opinion, having flat-structured teams is a good approach, as it empowers each individual and involves everyone within a team in a decision-making process. It cultivates diversity of thought by allowing each member to realize his ideas, increasing productivity because individual team members can make decisions independently and without involving the other members or team leaders. Although, major decisions cannot be made by individual members, and if no person has clear authority for making major decisions, they have to be made in slower ways. However, my experience has shown that a flat structure does not produce value when teamwork expectations are misaligned, and some members are unwilling to contribute equally and put as much effort as the other members do. If there are no or insufficient incentives or positive consequences for contribution and no, or insufficient negative consequences for lack of contribution, some team members may be underperforming. If this occurs, it can result in unequal contribution, missed deadlines, and resentment among team members with different performance. One way in which negative consequences and positive consequences could be imposed is by a leader with authority for imposing them; however, for our Master's project, it is not clear that we could assign such a leader ourselves.

A leaderless team can bring a dysfunction of a team called *avoidance of accountability*. Lencioni (2002) defines *avoidance of accountability* as "the unwillingness of team members to tolerate the interpersonal discomfort that accompanies calling a peer on his or her behavior and the more general tendency to avoid difficult conversations" (p. 212-213). The author suggests that the best solution to overcome this dysfunction is peer pressure. One way of doing it is to set up clear goals for the team and clarify what needs to be done to achieve them. Lencioni (2002) also indicates the importance of regular peer reviews, where each team member gives feedback on his teammates' performance. In my experience, there was a case of unequal contribution and avoidance of accountability in our project. We did not pay much attention to the Trello board's tasks and did not divide them between team members because we thought it was an unnecessary effort. Additionally, we did not assess and did not keep track of our performance. Adopting the Daily Scrum meeting format for our weekly meeting and group work assessment helped to detect underperformance in the team. Once we detected underperformance, we decided to improve the situation by the formalization of our development process. This formalization involved more frequent and attentive usage of the Trello board and more explicit task distribution between group members. However, there was still a lack of leadership to impose negative consequences for underperformance, so the measures had limited impact.

Lack of formalized development process. We did not adopt any formal process for our meetings, breakdown, and division of tasks, and enforcement of deadlines. The lack of clearly delineated tasks made it difficult for us to assess our progress and contributions of individual team members and made it difficult to determine whether we would meet deadlines.

Limited use of collaboration and communication tools. Lomas, Burke, and Page (2008) argue that "good technology (...) should allow users to extend the boundaries of what they are able to achieve and, at the very least, help people to perform better" (p. 3). In my opinion, collaboration and communication tools improve group work performance only if all the members in a team frequently use these tools. During our development process, the tasks on our three-column Trello board were seldom moved or updated, and thus it was unclear what is under implementation and what is finished. To increase usage of the Trello board in our team, we have established that a task would be counted as completed if its respective Trello card has not been marked as completed. Due to the COVID-19 pandemic, our decision-making process was conducted mainly on Slack. Therefore it was important for each group member to check Slack frequently to participate in conversations. Unfortunately, some member(s) of the group did not frequently use Slack, and as a consequence, these members were left out in some sense, as they did not participate or contribute to the decision-making process and brainstorming.

To conclude, I would recommend that teams engaging in similar development activities adopt a more formal development process from the start. This process should include more frequent group meetings, possibly in a Daily Scrum meeting format. The process should also make it easier to assess project status and progress and individual team members' contributions. It would also be advantageous to have a clear authority figure, possibly a supervisor, that can motivate underperforming team members. The use of a Kanban board to track the backlog, tasks in progress, and completed tasks, is also helpful, as it gives a visual representation of the project progress and status.

Chapter 9

Conclusion and future work

This chapter summarizes the work done with regard to the study aim and the research question and presents some suggestions for future research.

9.1 Conclusion

Design Science Research allowed us to conduct engaged research in the HISP community, exploring the current state of component reuse. Our findings show that even though web application developers use component-based frameworks such as React and Angular, they do not necessarily adopt component-based software engineering in their daily practice.

The practical aim of this project was fulfilled by developing a component repository, DHIS2 Shared Component Platform, that utilizes the existing infrastructure - NPM Registry, a repository of open-source software packages, and GitHub, a code hosting platform.

Design and development of a component repository in a software platform ecosystem was a difficult task. First, because we had to address and overcome the general challenges of CBSE, such as proper component storage, distribution, discoverability, trustworthiness, and certification. Secondly, because there were challenges brought by the context the repository was built within, a software platform ecosystem. We built a small and simple solution beneficial to our target user group and took the existing infrastructure into consideration. Our target user group was relatively small considering the number of the HISP groups and the number of developers involved in web application development in the HISP community. Given the diversity of practices and tooling identified throughout this study, I assume there are many practices, tooling, and technology in the community that we have not considered in our solution and which are yet to be explored. However, the orthogonal design that we adopted should reduce the resources required to add support for these additional technologies. Another challenging aspect of a component platform is its multisidedness, as it is vital to attract component providers and component users to achieve critical mass. Component users do not gain value from a component platform if it lacks reusable components, and component providers will be less interested in making an effort to publish their components to a repository that is not being used by component users. What makes this issue even more complex is the fact that repository adoption cannot be done through a command-and-control management model, as there is no single authority that could mandate what development boundary resources are to be used by web application developers.

The research question of this thesis was formulated as follows: *What are the essential design principles for implementing a component repository that facilitates component reuse in a software platform ecosystem?* To answer this research question and fulfill my theoretical research aim, I presented the following design principles: *Principle of component trustworthiness*, *Principle of balanced certification*, *Principle of component discoverability*, *Principle of installed base cultivation*, and *Principle of orthogonality* (Table 6.11). The established design principles are abstract and aim to guide the researchers and practitioners who are to implement a component repository in a software platform ecosystem.

9.2 Research contribution

My design principles address challenges pertaining to component-based software engineering identified in the reviewed literature and provide valuable insights on how to facilitate component reuse in a software platform ecosystem.

Section 6.2 is a contribution to the HISP community and the DHIS2 Design Lab participants. It provides insights on component reuse practices and challenges we have identified and describes how they were addressed in our solution. Chapter 6, and specifically, Section 6.4 can be used by practitioners who are to build a component repository and wish to gain insight into its architectural design.

I also communicated my research results in two conferences for students and young scientists. My conference abstracts can be found in Appendix H and Appendix I.

9.3 Future work

First, future research should continue to explore the social aspect of software reuse, such as software reuse practices and knowledge, motivation, and impediments to reuse, because component repositories such as SCP are essentially built to improve software reuse practices which are inherently social in nature.

Second, future research should further explore component trustworthiness and certification process, which showed to be a challenge by the CBSE body of knowledge and was challenging for the practitioners in this study. The identified aspects concerning the implementation of certification discussed in Section 6.2.2 can be considered, as well as the proposed Certification Process Classification Model in Figure 6.1.

Lastly, there is some research on the role of boundary resources in a platform ecosystem with regard to resourcing and securing, but this research does not explore these aspects considering the differences between application boundary resources and development boundary resources. Specifically, whether they could be used by platform owners to directly or indirectly affect practices in the ecosystem. Platform owners could provide resources exclusively for a particular framework, for example, React, making it more attractive for the community to adopt. While not providing resources for other frameworks, for example, Angular, which could make these less attractive.

9.4 Suggestions for further work on SCP

I suggest that the developers who will take over this project keep working on the improvements of the component repository considering the results from the evaluation I have conducted. I also suggest a discussion on whether it would be feasible to allocate resources to the certification process at this stage, while it is not clear whether the repository will be adopted by developers. One potential pathway would be putting the repository in production and populating it with reusable components, i.e., making it attractive to component users. Additionally, one should continue to explore software reuse practices in the other HISP groups to attract more users. With regard to certification, it would be interesting to explore further what component certification metrics the DHIS2 core team would adopt, and whether their perception of components quality differs from the perceptions of web application developers who engage in CBSE.

References

About npm. (n.d.). https://www.npmjs.com/about.

About semantic versioning. (n.d.). Retrieved 2021-04-12, from https://docs.npmjs.com/about-semantic-versioning

Adu-Gyamfi, E., Nielsen, P., & Sæbø, J. (2019, 11). The dynamics of a global health information systems research and implementation project. In *The dynamics of a global health information systems research and implementation project.*

Baskerville, R. L., Kaul, M., & Storey, V. C. (2015). Genres of inquiry in design-science research: Justification and evaluation of knowledge production. *MIS Quarterly*, *39*(3), 541–564. Retrieved from https://www.jstor.org/stable/26629620

Bianco, V. D., Myllarniemi, V., Komssi, M., & Raatikainen, M. (2014, April). The Role of Platform Boundary Resources in Software Ecosystems: A Case Study. In *2014 IEEE/IFIP Conference on Software Architecture* (pp. 11–20). Sydney, Australia: IEEE. Retrieved 2020-11-22, from http://ieeexplore.ieee.org/document/6827094/ doi: 10.1109/WICSA.2014.41

Boland, R., Lyytinen, K., & Yoo, Y. (2007, 08). Wakes of innovation in project networks: The case of digital 3-d representations in architecture, engineering, and construction. *Organization Science*, *18*, 631-647. doi: 10.1287/orsc.1070.0304

Booch, G., Engle, M. W., Maksimchuk, R. A., Conallen, J., Young, B. J., & Houston, K. A. (2001). *Object-oriented analysis and design with applications.* Addison-Wesley Professional.

Bourque, P., Fairley, R. E., & IEEE Computer Society. (2014). *Guide to the software engineering body of knowledge.* IEEE Software. (OCLC: 973217192)

Braa, J., & Sahay, S. (2017, 04). The dhis2 open source software platform: Evolution over time and space..

Briggs, R., & Schwabe, G. (2011). On expanding the scope of design science in is research. In *Desrist.*

Brocke, J. v., Hevner, A., & Maedche, A. (2020, 09). Introduction to design science research. In (p. 1-13). doi: 10.1007/978-3-030-46781-4_1

Brown, A. (2000, 01). Large-scale, component-based development / a.w. brown. *OAI*.

Cai, Y., & Zhu, D. (2016, November). Reputation in an open source software community: Antecedents and impacts. *Decision Support Systems*, *91*, 103–112. Retrieved 2021-04-30, from https://linkinghub.elsevier.com/retrieve/pii/S0167923616301403 doi: 10.1016/j.dss.2016.08.004

Capretz, L. (2005, 01). Y: A new component-based software life cycle model. *Journal of Computer Science*, *1*. doi: 10.3844/jcssp.2005.76.82

CFI. (2021). *What is a market?* Retrieved 2021-05-02, from https://corporatefinanceinstitute.com/resources/knowledge/economics/market/

Chesbrough, H. W. (2003). *Open innovation: the new imperative for creating and profiting from technology.* Boston, Mass: Harvard Business School Press.

Christiansson, B. (2002). Software Components — difficulties with acquisition. *Karlstads universitet*, 12.

Constantinides, P., Henfridsson, O., & Parker, G. G. (2018, June). Introduction—Platforms and Infrastructures in the Digital Age. *Information Systems Research*, *29*(2), 381–400. Retrieved 2021-02-18, from http://pubsonline.informs.org/doi/10.1287/isre.2018.0794 doi: 10.1287/isre.2018.0794

Crang, M., & Cook, I. (2007). *Doing ethnographies.* Los Angeles: SAGE.

Creswell, J. W., & Creswell, J. D. (2018). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.*

Crnkovic, I., Hnich, B., Jonsson, T., & Kiziltan, Z. (2002, October). Specification, implementation, and deployment of components. *Communications of the ACM*, *45*(10), 35–40. Retrieved 2021-02-04, from https://dl.acm.org/doi/10.1145/570907.570928 doi: 10.1145/570907.570928

Crnkovic, I., & Larsson, M. (Eds.). (2002). *Building reliable component-based software systems.* Boston: Artech House.

Crotty, M. (1998). *The foundations of social research: meaning and perspective in the research process.* London ; Thousand Oaks, Calif: Sage Publications.

Darlington, Y., & Scott, D. (2020). *Qualitative research in practice Stories from the field* (1st ed.). Routledge. Retrieved 2021-03-10, from https://www.taylorfrancis.com/books/9781000250862 doi: 10.4324/9781003117025

de Reuver, M., Sørensen, C., & Basole, R. C. (2018, June). The Digital Platform: A Research Agenda. *Journal of Information Technology*, *33*(2), 124–135. Retrieved 2020-05-07, from http://journals.sagepub.com/doi/10.1057/s41265-016-0033-3 doi: 10.1057/s41265-016-0033-3

DHIS2 (Ed.). (2021a). *About dhis2.* Retrieved 2021-01-26, from https://www.dhis2.org/about

DHIS2 (Ed.). (2021b). *Global hisp network.* Retrieved 2021-04-07, from https://dhis2.org/hisp-network/

DHIS2 (Ed.). (2021c). *Technology platform.* Retrieved 2021-04-07, from https://dhis2.org/technology/

Dresch, A., Lacerda, D. P., & Antunes Jr, J. A. V. (2015). *Design Science Research: A Method for Science and Technology Advancement.*

Cham: Springer International Publishing. Retrieved 2020-11-28, from http://link.springer.com/10.1007/978-3-319-07374-3 doi: 10.1007/978-3-319-07374-3

Eder, J., & Schrefl, M. (1995, 05). Coupling and cohesion in object-oriented systems. *IBM Systems Journal*.

eHealth, P. (Ed.). (2021). *Best practice: The health information system with dhis2 in cauca.* Retrieved 2021-01-19, from https://www.paho.org/ict4health/index.php?option=com_content&view=article&id=192:good-practice-the-health-information-system-with-dhis2-in-cauca&Itemid=204&lang=en

Evans, P. C., & Gawer, A. (2016). The Rise of the Platform Enterprise. *The Emerging Platform Economy Series No. 1*, 30.

Gare, K. (2010). Formative Infrastructure for IT-Adoption Understanding the Dynamics of IT-Use in SME's. *AIS Electronic Library (AISeL)*, 37.

Gawer, A. (2009). *Platforms, Markets and Innovation.* Edward Elgar Publishing. Retrieved 2020-05-07, from http://www.elgaronline.com/view/9781848440708.xml doi: 10.4337/9781849803311

Ghazawneh, A. (2012). *Towards a boundary resources theory of software platforms.* Jönköping: Jönköping International Business School. (Diss. (sammanfattning) Jönköping : Högskolan i Jönköping, 2012)

Ghazawneh, A., & Henfridsson, O. (2010). Governing third-party development through platform boundary resources. In R. Sabherwal & M. Sumner (Eds.), *Proceedings of the international conference on information systems, ICIS 2010, saint louis, missouri, usa, december 12-15, 2010* (p. 48). Association for Information Systems. Retrieved from http://aisel.aisnet.org/icis2010_submissions/48

Ghazawneh, A., & Henfridsson, O. (2013, 03). Balancing platform control and external contribution in third-party development: The boundary resources model. *Information Systems Journal*, *23*. doi: 10.1111/j.1365-2575.2012.00406.x

Gill, T., & Hevner, A. (2013, 08). A fitness-utility model for design science research. *ACM Transactions on Management Information Systems*, *4*, 5-24. doi: 10.1145/2499962.2499963

Goldkuhl, G. (2012). Design Research in Search for a Paradigm: Pragmatism Is the Answer. In M. Helfert & B. Donnellan (Eds.), *Practical Aspects of Design Science* (Vol. 286, pp. 84–95). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2021-03-05, from http://link.springer.com/10.1007/978-3-642-33681-2_8 (Series Title: Communications in Computer and Information Science) doi: 10.1007/978-3-642-33681-2_8

Gregor, S., & Hevner, A. R. (2013, February). Positioning and Presenting Design Science Research for Maximum Impact. *MIS Quarterly*, *37*(2), 337–355. Retrieved 2020-11-16, from https://misq.org/positioning-and-presenting-design-science-research-for-maximum-impact.html doi: 10.25300/MISQ/2013/37.2.01

Gregor, S., Kruse, L., & Seidel, S. (2020, November). Research Perspectives:

The Anatomy of a Design Principle. *Journal of the Association for Information Systems*, *21*, 1622–1652. Retrieved 2021-02-11, from https://aisel.aisnet.org/jais/vol21/iss6/2/ doi: 10.17705/1jais.00649

Guides, G. (Ed.). (2020). *Hello world.* Retrieved 2020-12-07, from https://guides.github.com/activities/hello-world/

Hanseth, O., & Bygstad, B. (2018, 06). Platformization, infrastructuring and platform-oriented infrastructures. a norwegian e-health case. *Working papers in Information Systems*.

Hanseth, O., & Lyytinen, K. (2010, March). Design Theory for Dynamic Complexity in Information Infrastructures: The Case of Building Internet. *Journal of Information Technology*, *25*(1), 1–19. Retrieved 2021-04-06, from http://journals.sagepub.com/doi/10.1057/jit.2009.19 doi: 10.1057/jit.2009.19

Hein, A., Schreieck, M., Riasanow, T., Setzke, D. S., Wiesche, M., Böhm, M., & Krcmar, H. (2020, March). Digital platform ecosystems. *Electronic Markets*, *30*(1), 87–98. Retrieved 2021-02-18, from http://link.springer.com/10.1007/s12525-019-00377-4 doi: 10.1007/s12525-019-00377-4

Heineman, G. T., & Councill, W. T. (Eds.). (2001). *Component-based software engineering: putting the pieces together.* Boston: Addison-Wesley.

Hevner, A., & Chatterjee, S. (2010). *Design Research in Information Systems* (Vol. 22). Boston, MA: Springer US. Retrieved 2021-03-29, from http://link.springer.com/10.1007/978-1-4419-5653-8 doi: 10.1007/978-1-4419-5653-8

Hevner, A., Prat, N., Comyn-Wattiau, I., & Akoka, J. (2018). A pragmatic approach for identifying and managing design science research goals and evaluation criteria. *HAL*, 16.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (n.d.). Design Science in Information Systems Research. *MIS Quarterly*, 33.

Hornby, A. S. (Ed.). (1995). *Oxford advanced learner's dictionary of current english.* Fifth edition. Oxford, England : Oxford University Press, 1995.

Hunt, A., & Thomas, D. (2000). *The pragmatic programmer: from journeyman to master.* Addison-Wesley.

Hänninen, M. (2020, March). Review of studies on digital transaction platforms in marketing journals. *The International Review of Retail, Distribution and Consumer Research*, *30*(2), 164–192. Retrieved 2020-11-22, from https://www.tandfonline.com/doi/full/10.1080/09593969.2019.1651380 doi: 10.1080/09593969.2019.1651380

Iivari, J., Hansen, M. R. P., & Haj-Bolouri, A. (2018). A Framework for Light Reusability Evaluation of Design Principles in Design Science Research. *DESRIST 2018*, 16.

Iivari, J., & Venable, J. R. (2009). Action research and design science research - Seemingly similar but decisively dissimilar. *AISeL*, 13.

Ingeno, J. (2018). *Software architect's handbook: become a successful soft-*

ware architect by implementing effective architecture concepts. Birmingham, UK: Packt Publishing. Retrieved 2021-01-23, from http://proquestcombo.safaribooksonline.com/9781788624060 (OCLC: 1078373548)

ISO/IEC, & IEEE. (2017, September). *ISO/IEC/IEEE 24765:2017 Systems and software engineering — Vocabulary.* ISO/IEC/IEEE.

ISO/IEC 25010. (2011). *ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models.*

Jacobides, M. G., Cennamo, C., & Gawer, A. (2018, August). Towards a theory of ecosystems. *Strategic Management Journal*, *39*(8), 2255–2276. Retrieved 2021-02-18, from https://onlinelibrary.wiley.com/doi/abs/10.1002/smj.2904 doi: 10.1002/smj.2904

Jalender, B., Govardhan, D., & Premchand, P. (2010, 01). Breaking the boundaries for software component reuse technology. *International Journal of Computer Applications*, *13*. doi: 10.5120/1782-2458

Jalender, B., N.Gowtham, Kumar, K.Praveen, K.Murahari, & K.Sampath. (2010, 11). Technical impediments to software reuse. *International Journal of Engineering Science and Technology*, *2*.

Kaur, K., & Singh, H. (2009, December). Candidate Process Models for Component Based Software Development. *Journal of Software Engineering*, *4*(1), 16–29. Retrieved 2021-02-15, from https://www.scialert.net/abstract/?doi=jse.2010.16.29 doi: 10.3923/jse.2010.16.29

Koskinen, K., Bonina, C., & Eaton, B. (2019). Digital Platforms in the Global South: Foundations and Research Agenda. In P. Nielsen & H. C. Kimaro (Eds.), *Information and Communication Technologies for Development. Strengthening Southern-Driven Cooperation as a Catalyst for ICT4D* (Vol. 551, pp. 319–330). Cham: Springer International Publishing. Retrieved 2021-02-18, from http://link.springer.com/10.1007/978-3-030-18400-1_26 (Series Title: IFIP Advances in Information and Communication Technology) doi: 10.1007/978-3-030-18400-1_26

Kotonya, Sommerville, & Hall. (2003). Towards a classification model for component-based software engineering research. In *Proceedings of the 20th IEEE Instrumentation Technology Conference (Cat No 03CH37412) EURMIC-03* (pp. 43–52). Belek-Antalya, Turkey: IEEE. Retrieved 2021-01-28, from http://ieeexplore.ieee.org/document/1231566/ doi: 10.1109/EURMIC.2003.1231566

Landau, T. (2021). *How to install a single component from any ui library with npm + bit.* Retrieved 2021-04-14, from https://blog.bitsrc.io/how-to-install-a-single-component-from-any-ui-library-8d0519544d7b

Lau, K.-K. (Ed.). (2004). *Component-based software development: case studies* (No. v. 1). New Jersey: World Scientific. (OCLC: ocm55849387)

Lau, K.-K. (Ed.). (2018). *An introduction to component-based software development: 3.* World Scientific.

Lencioni, P. M. (2002). *The five dysfunctions of a team.* Jossey-Bass Inc.,U.S.

Li, M. (2019). An Approach to Addressing the Usability and Local Relevance of Generic Enterprise Software. *AISeL*, 16.

Li, M. (2020). *Facilitating collaborative meta-design: Building a shared component library for web-app development.* Retrieved 2021-01-23, from https://www.mn.uio.no/ifi/studier/masteroppgaver/is/dhis2-design-lab/dhis2-design-lab-component-library.html

Lieberherr, K. J., & Holland, I. M. (1989). Assuring good style for object-oriented programs. *IEEE Software*, *6*(5), 38-48. doi: 10.1109/52.35588

Lomas, C., Burke, M., & Page, C. L. (2008). Collaboration Tools. *ELI paper*, 12.

Lopez, V., & Whitehead, D. (2013, 01). Sampling data and data collection in qualitative research. In (p. 123-140).

Lyytinen, K., Yoo, Y., & Boland, R. (2015, 01). Digital product innovation within four classes of innovation networks. *Information Systems Journal*, *26*, n/a-n/a. doi: 10.1111/isj.12093

Material-ui. (2021). Retrieved 2021-04-14, from https://github.com/mui-org

Merriam-Webster. (2019). Component. In *Merriam-webster.com dictionary.* Merriam-Webster, Incorporated. Retrieved 2021-02-03, from https://www.merriam-webster.com/dictionary/component

Mitchell, J. C., & Apt, K. (2001). *Concepts in programming languages.* USA: Cambridge University Press.

Mohammad, M. S. (2011). *A formal component-based software engineering approach for developing trustworthy systems.* (Unpublished doctoral dissertation). Library and Archives Canada = Bibliothèque et Archives Canada, Ottawa. (ISBN: 9780494634172 OCLC: 769258225)

Moon, K., & Blackman, D. (2014, October). A Guide to Understanding Social Science Research for Natural Scientists: Social Science for Natural Scientists. *Conservation Biology*, *28*(5), 1167–1177. Retrieved 2021-03-04, from http://doi.wiley.com/10.1111/cobi.12326 doi: 10.1111/cobi.12326

Mpande, L. C. (2018). *Simple user management app for water point users.* Retrieved 2021-01-26, from https://github.com/hisptz/user-mananger

Msiska, B., & Nielsen, P. (2018, April). Innovation in the fringes of software ecosystems: the role of socio-technical generativity. *Information Technology for Development*, *24*(2), 398–421. Retrieved 2020-05-09, from https://www.tandfonline.com/doi/full/10.1080/02681102.2017.1400939 doi: 10.1080/02681102.2017.1400939

Nelson, M. L. (1996). Barriers to Software Reuse and the Projected Impact of World Wide Web on Software Reuse. *CiteSeer*, 10.

Nielsen, P. (2006). *A conceptual framework of information infrastructure building* (Unpublished doctoral dissertation). University of Oslo.

Nielsen, P. (Ed.). (2021). *Hisp groups.* Retrieved 2021-04-07, from https://www.mn.uio.no/ifi/english/research/networks/hisp/groups/

Nowell, L. S., Norris, J. M., White, D. E., & Moules, N. J. (2017, December). Thematic Analysis: Striving to Meet the Trustworthiness Criteria. *International Journal of Qualitative Methods*, *16*(1), 160940691773384. Retrieved 2021-03-11, from http://journals.sagepub.com/doi/10.1177/1609406917733847 doi: 10.1177/1609406917733847

NPM. (2020). *Auditing package dependencies for security vulnerabilities.* Retrieved 2020-12-04, from https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities

npm-publish. (n.d.). Retrieved 2021-04-12, from https://docs.npmjs.com/cli/v6/commands/npm-publish

Peffers, K., Rothenberger, M., Tuunanen, T., & Vaezi, R. (2012). Design Science Research Evaluation. In D. Hutchison et al. (Eds.), *Design Science Research in Information Systems. Advances in Theory and Practice* (Vol. 7286, pp. 398–410). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2020-11-26, from http://link.springer.com/10.1007/978-3-642-29863-9_29 (Series Title: Lecture Notes in Computer Science) doi: 10.1007/978-3-642-29863-9_29

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2008, December). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. Retrieved 2020-11-16, from https://www.tandfonline.com/doi/full/10.2753/MIS0742-1222240302 doi: 10.2753/MIS0742-1222240302

Prat, N., Comyn-Wattiau, I., & Akoka, J. (2015, July). A Taxonomy of Evaluation Methods for Information Systems Artifacts. *Journal of Management Information Systems*, *32*(3), 229–267. Retrieved 2021-03-25, from http://www.tandfonline.com/doi/full/10.1080/07421222.2015.1099390 doi: 10.1080/07421222.2015.1099390

Radiance ui. (2021). Retrieved 2021-04-14, from https://www.npmjs.com/package/radiance-ui

Rehman, A. A., & Alharthi, K. (2016). An Introduction to Research Paradigms. *nternational Journal of Educational Investigations*, 10.

Russell, B. (2004). *History of western philosophy.* Taylor & Francis. Retrieved from https://books.google.no/books?id=81Zz27fQuvsC

Schwandt, T. A. (2001). *Dictionary of qualitative inquiry* (2nd ed. ed.). Thousand Oaks, Cal: Sage.

scope. (n.d.). https://docs.npmjs.com/cli/v7/using-npm/scope.

Seacord, R. C. (1999). Software Engineering Component Repositories. *Proceedings of the International Workshop on Component-Based Software Engineering*, 7.

Singh, V., & Bhattacharjee, V. (2013, 08). Identifying coupling metrics and impact on software quality. *International Journal of Engineering and Technology*, *5*, 3433-3438.

Sommerville, I. (2011). *Software engineering* (9th ed ed.). Boston: Pearson. (OCLC: ocn462909026)

Spolsky, J. (2008). *More joel on software.* Berkeley, Calif: Apress.

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, *13*(2), 115-139. doi: 10.1147/sj.132.0115

Szyperski, C. (2002). *Component software: Beyond object-oriented programming* (2nd ed.). USA: Addison-Wesley Longman Publishing Co., Inc.

Szyperski, C., Gruntz, D., & Murer, S. (2011). *Component software: beyond object-oriented programming* (2nd ed ed.). London: Addison-Wesley. (OCLC: 838151413)

Tal, L., & Maple, S. (2019, June 4). *npm passes the 1 millionth package milestone! what can we learn?* https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/.

Tiwana, A. (2014). *Platform ecosystems: aligning architecture, governance, and strategy.* Amsterdam ; Waltham, MA: MK.

Tiwari, U., & Kumar, S. (2020). *Component-based software engineering: Methods and metrics.* Chapman and Hall/CRC. doi: 10.1201/9780429331749

UNPKG. (2020). *Hello world.* Retrieved 2020-12-07, from https://unpkg.com

Venable, J., Pries-Heje, J., & Baskerville, R. (2016, January). FEDS: a Framework for Evaluation in Design Science Research. *European Journal of Information Systems*, *25*(1), 77–89. Retrieved 2020-11-28, from https://www.tandfonline.com/doi/full/10.1057/ejis.2014.36 doi: 10.1057/ejis.2014.36

vom Brocke, J., & Maedche, A. (2019, September). The DSR grid: six core dimensions for effectively planning and communicating design science research projects. *Electronic Markets*, *29*(3), 379–385. Retrieved 2021-04-06, from http://link.springer.com/10.1007/s12525-019-00358-7 doi: 10.1007/s12525-019-00358-7

What is a package? (n.d.). https://npm.github.io/how-npm-works-docs/theory-and-design/what-is-a-package.html.

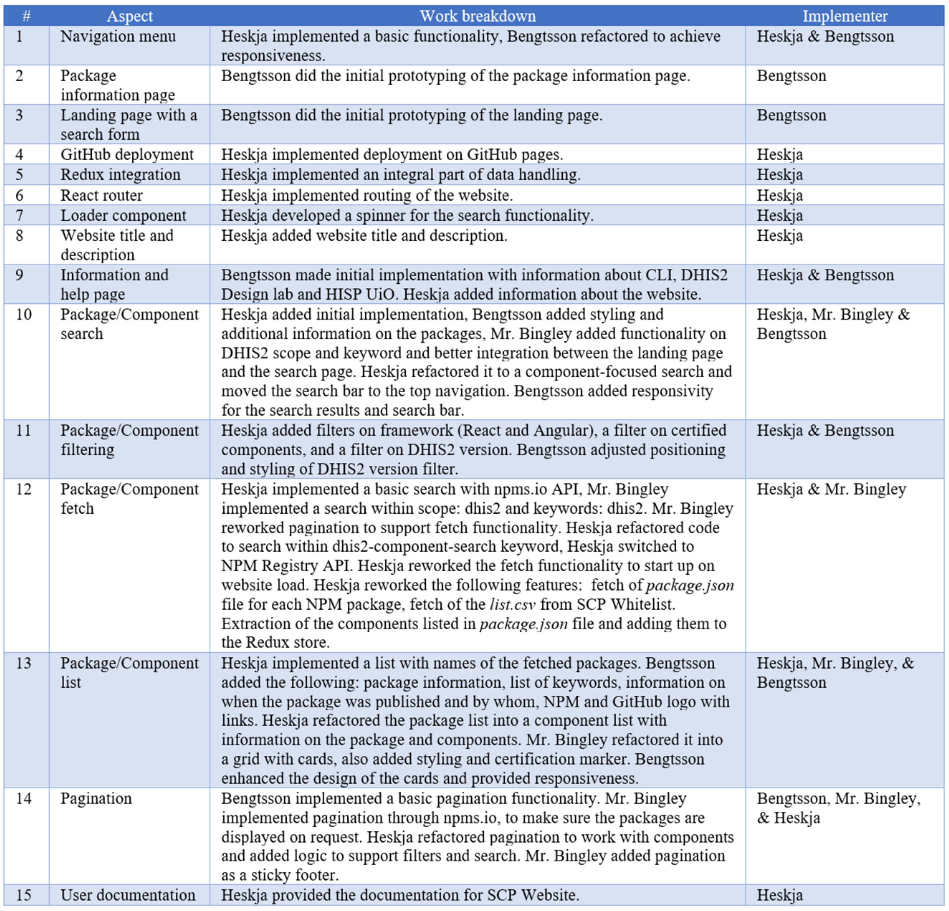
Zittrain, J. (2005). The Generative Internet. *Harvard Law Review*, *119*.

Appendices

Appendix A

Detailed work distribution

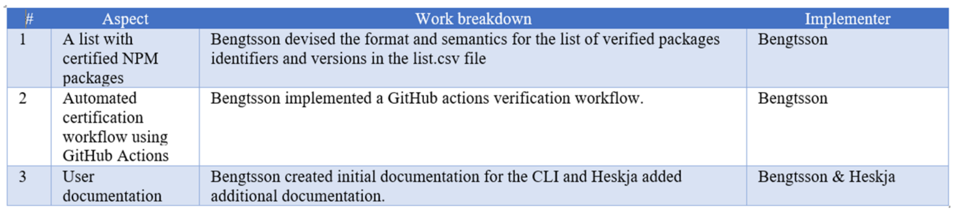
Detailed work distribution during the design and development of SCP [1].



#	Aspect	Work breakdown	Implementer
1	Navigation menu	Heskja implemented a basic functionality, Bengtsson refactored to achieve responsiveness.	Heskja & Bengtsson
2	Package information page	Bengtsson did the initial prototyping of the package information page.	Bengtsson
3	Landing page with a search form	Bengtsson did the initial prototyping of the landing page.	Bengtsson
4	GitHub deployment	Heskja implemented deployment on GitHub pages.	Heskja
5	Redux integration	Heskja implemented an integral part of data handling.	Heskja
6	React router	Heskja implemented routing of the website.	Heskja
7	Loader component	Heskja developed a spinner for the search functionality.	Heskja
8	Website title and description	Heskja added website title and description.	Heskja
9	Information and help page	Bengtsson made initial implementation with information about CLI, DHIS2 Design lab and HISP UiO. Heskja added information about the website.	Heskja & Bengtsson
10	Package/Component search	Heskja added initial implementation, Bengtsson added styling and additional information on the packages, Mr. Bingley added functionality on DHIS2 scope and keyword and better integration between the landing page and the search page. Heskja refactored it to a component-focused search and moved the search bar to the top navigation. Bengtsson added responsivity for the search results and search bar.	Heskja, Mr. Bingley & Bengtsson
11	Package/Component filtering	Heskja added filters on framework (React and Angular), a filter on certified components, and a filter on DHIS2 version. Bengtsson adjusted positioning and styling of DHIS2 version filter.	Heskja & Bengtsson
12	Package/Component fetch	Heskja implemented a basic search with npms.io API, Mr. Bingley implemented a search within scope: dhis2 and keywords: dhis2. Mr. Bingley reworked pagination to support fetch functionality. Heskja refactored code to search within dhis2-component-search keyword, Heskja switched to NPM Registry API. Heskja reworked the fetch functionality to start up on website load. Heskja reworked the following features: fetch of *package.json* file for each NPM package, fetch of the *list.csv* from SCP Whitelist. Extraction of the components listed in *package.json* file and adding them to the Redux store.	Heskja & Mr. Bingley
13	Package/Component list	Heskja implemented a list with names of the fetched packages. Bengtsson added the following: package information, list of keywords, information on when the package was published and by whom, NPM and GitHub logo with links. Heskja refactored the package list into a component list with information on the package and components. Mr. Bingley refactored it into a grid with cards, also added styling and certification marker. Bengtsson enhanced the design of the cards and provided responsiveness.	Heskja, Mr. Bingley, & Bengtsson
14	Pagination	Bengtsson implemented a basic pagination functionality. Mr. Bingley implemented pagination through npms.io, to make sure the packages are displayed on request. Heskja refactored pagination to work with components and added logic to support filters and search. Mr. Bingley added pagination as a sticky footer.	Bengtsson, Mr. Bingley, & Heskja
15	User documentation	Heskja provided the documentation for SCP Website.	Heskja

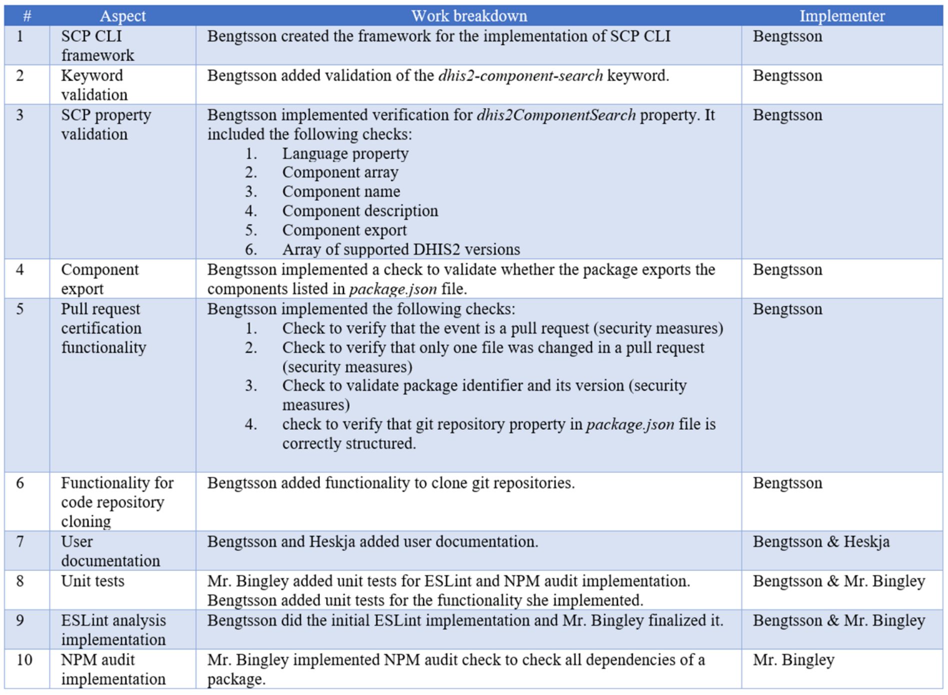
Figure A.1: SCP Website work distribution.

[1]Written in collaboration with Håkon André Heskja.



#	Aspect	Work breakdown	Implementer
1	A list with certified NPM packages	Bengtsson devised the format and semantics for the list of verified packages identifiers and versions in the list.csv file	Bengtsson
2	Automated certification workflow using GitHub Actions	Bengtsson implemented a GitHub actions verification workflow.	Bengtsson
3	User documentation	Bengtsson created initial documentation for the CLI and Heskja added additional documentation.	Bengtsson & Heskja

Figure A.2: SCP Whitelist work distribution.



#	Aspect	Work breakdown	Implementer
1	SCP CLI framework	Bengtsson created the framework for the implementation of SCP CLI	Bengtsson
2	Keyword validation	Bengtsson added validation of the *dhis2-component-search* keyword.	Bengtsson
3	SCP property validation	Bengtsson implemented verification for *dhis2ComponentSearch* property. It included the following checks: 1. Language property 2. Component array 3. Component name 4. Component description 5. Component export 6. Array of supported DHIS2 versions	Bengtsson
4	Component export	Bengtsson implemented a check to validate whether the package exports the components listed in *package.json* file.	Bengtsson
5	Pull request certification functionality	Bengtsson implemented the following checks: 1. Check to verify that the event is a pull request (security measures) 2. Check to verify that only one file was changed in a pull request (security measures) 3. Check to validate package identifier and its version (security measures) 4. check to verify that git repository property in *package.json* file is correctly structured.	Bengtsson
6	Functionality for code repository cloning	Bengtsson added functionality to clone git repositories.	Bengtsson
7	User documentation	Bengtsson and Heskja added user documentation.	Bengtsson & Heskja
8	Unit tests	Mr. Bingley added unit tests for ESLint and NPM audit implementation. Bengtsson added unit tests for the functionality she implemented.	Bengtsson & Mr. Bingley
9	ESLint analysis implementation	Bengtsson did the initial ESLint implementation and Mr. Bingley finalized it.	Bengtsson & Mr. Bingley
10	NPM audit implementation	Mr. Bingley implemented NPM audit check to check all dependencies of a package.	Mr. Bingley

Figure A.3: SCP CLI work distribution.

Appendix B

SCP User Documentation

DHIS2 Shared Component Platform (SCP) Documentation

Documentation for DHIS2 Shared Component platform website, SCP command line interface and SCP Whitelist repository

1. About DHIS2 Shared Component Platform (DHIS2 SCP)

DHIS2 Shared Component platform is a platform for sharing and reuse of software components created by HISP community. The components hosted on this platform are React or Angular UI components and can be used as building blocks for a web application.

Why do we need reusable components? * More effective development of Front end web application * Consistency in web applications * Smaller code base to maintain

What characterizes reusable components? * Easy plug-able components * Can be published directly to NPM * Extendable template * Web-browser independent

Some of the examples of reusable component libraries: Material UI, Dhis2/ui, Semantic UI.

DHIS2 Shared Component Platform consists of three distinct modules:

- SCP website
- SCP command line interface
- SCP whitelist repository

Use the SCP website to discover components within NPM packages. Run the SCP command line interface from the terminal to check if your package is set up properly, so that your components become discoverable on the SCP website. Apply for the package quality assurance in the SCP whitelist repository and let your package be assessed by the team of maintainers.

1.2 Getting started

If you want to publish your package with components for further reuse, see section **2. Publishing your package**. If you want to verify your package locally, see section **3. Verifying your package locally**. If you want to apply for package verification, see section **4. Applying for package verification**. If you are here to learn how to find components, see section **5. Finding reusable**

components. If you are part of the maintainer team and want to learn about the verification workflow, see section **6. Maintaining package verification workflow**.

2. Publishing your package

We assume you already have a package with reusable components that you want to publish on NPM and make available on the SCP website. However if you do not know how to create and publish such a package, you can read more about it here.

To make your package searchable on the SCP website, two conditions must be met:

- Your `package.json` file must include `dhis2-component-search` keyword. (see section **3.1.1 Keyword**)
- Your `package.json` file must include `dhis2ComponentSearch` property with correct values and structure (see section **3.1.3 The `dhis2ComponentSearch` property**).

However, there are more conditions that must be met if you plan to submit your package for verification and pre-verify it locally. If this is the case, read the section **3. Verifying your package locally** and **4. Applying for package verification**.

When you have finished working on your `package.json` structure, you can proceed to publishing your package on NPM.

3. Verifying your package locally

SCP command line interface helps you to create NPM packages with React components that can be used to build DHIS2 apps or other components.

You can install the SCP CLI with the following command:

```
npm install -g https://github.com/haheskja/scp-cli#master
```

This package provides a command line interface `dhis2-scp-cli` with various commands.

- `dhis2-scp-cli verify`: This command will check the quality of your NPM package.

The command will do the following:

- Verify that the package's `package.json` has the `dhis2-component-search` keyword (see section 3.1.1).
- Verify that the package's `package.json` has the `dhis2ComponentSearch` property with correct values and structure (see section 3.1.3).
- Run `npm audit` check and output the result. (See npm audit)
- Run `eslint` and output the result. (See ESLint)

This command should be run inside the directory of your npm package.

To see additional options for this command run `dhis2-scp-cli verify --help`. For verbose output run with `dhis2-scp-cli verify -vvv`.

The SCP CLI also provides a command for verifying pull requests: `dhis2-scp-cli pr-verify`.

This command can be used by component owners to check their pull requests, for example:

```
dhis2-scp-cli pr-verify --pull-request-url "https://api.github.com/
repos/dhis2designlab/scp-whitelist/pulls/14"
```

It can also be used by component owners to check a specific package identifier and version without there being any pull request, for example:

```
dhis2-scp-cli pr-verify --fake-package-data "@dhis2/ui-core,5.7.3"
```

3.1 Verification prerequisites

3.1.1 Keyword

Your `package.json` file must include `dhis2-component-search` keyword as follows:

```
{
  "keywords": [
      "dhis2-component-search"
  ]
}
```

3.1.2 Repository

Currently we only support packages hosted on Github. Your `package.json` file must include `repository` property, that includes key/value pairs for repository type and url. Inside your `package.json`it would look like this:

```
{
   "repository": {
        "type" : "git",
        "url" : "https://github.com/npm/cli.git"
    }
}
```

The URL should be a publicly available and you must specify the repository type and url. Note that you cannot use shortcut syntax, e.g. `"repository": "github:user/repo"`, and you must use `https://`.

3.1.3 The `dhis2ComponentSearch` property

The `dhis2ComponentSearch` property of a `package.json` file includes the information about the package that is relevant to SCP and its search functionality.

The `dhis2ComponentSearch` property must include key/value pairs for framework (using `language` key, we currently support `react` and `angular`), and components. The `component` property, in turn, takes an array of component objects. Each component object must include following information defined as key/value pairs:

Required: * The `name` property contains the name of the exported component * The `export` property contains the identifier of the exported component. The expectation is that if the package is loaded as a commonJS module, that the component exist as a property with this identifier on the commonJS module. * The `description` property contains the description of the exported component

Optional:

- The `dhis2Version` optional property contains the DHIS2 versions supported by the exported component, the versions must be specified as an array of strings.

Inside your `package.json`, the `dhis2ComponentSearch` property may look something like this:

```
{
   "dhis2ComponentSearch": {
    "language": "react",
    "components": [
      {
        "name": "Organizational Unit Tree",
        "export": "OrgUnitTree",
        "description": "A simple OrgUnit Tree",
        "dhis2Version": [
          "32.0.0",
          "32.1.0",
          "33.0.0"
        ]
      }
    ]
   }
}
```

3.1.4 Components as commonJS modules

Our verification process requires your components to be distributed as commonJS modules. The command `npm install` should result in a valid commonJS module. One good way to achieve this is with the help of create-react-library CLI, as it bundles `commonjs` and `es` module formats.

3.1.5 NPM and Github

Your package must be published on NPM and have a public Github repository. Since we verify a specific version of the package, a git tag with this version should be added. When tagging releases in a version control system, the tag for a version must be `vX.Y.Z` e.g. `v1.0.2`.

4. Applying for package verification

DHIS2 SCP whitelist repository contains a list of verified NPM packages with reusable components that comply with SCP. Pull requests to this repository will be validated with a GitHub actions workflow.

If you want your package verified and added to the list of verified packages, you need to submit your package for verification. For that you would need to modify `list.csv` file by adding a new line containing your NPM package `identifier`and its `version` separated by a comma, e.g. `lodash,4.17.14`. Since you do not have a write access to the repository, a change in this file will write it to a new branch in your fork, so you can make a pull request. Fill in a title and description and create your pull request, which in turn, will trigger the verification workflow on your package.

The verification workflow:

- Checks for verification prerequisites defined in section **3.1 Verification prerequisites**
- Lints the code with ESLint

- Runs npm audit

The output of the verification workflow serves as a basis for acceptance of the pull request, but the final decision for acceptance is up to the maintaners.

5. Finding reusable components

SCP website provides necessary functionality for finding reusable components registered with SCP - both verified and unverified. It fetches and lists all the exported components within all packages published on NPMjs that contain the `dhis2-component-search` keyword in `package.json` file and `dhis2ComponentSearch` property with correct values and structure.

The website provides the following filters to narrow search result: * Framework * All: Shows all React and Angular components * React: Shows only React components * Angular: Shows only Angular components

- DHIS2 Version : allows filtering on supported DHIS2 version
- Show only verified components: shows the components within the packages that have their latest version verified.

Each component is represented by a component card that contains the following information: * Component name * Component description * Component export * Package identifier * Package keywords * Information on latest published version * Link to the package on NPM * Latest unverified package version * Latest verified package version

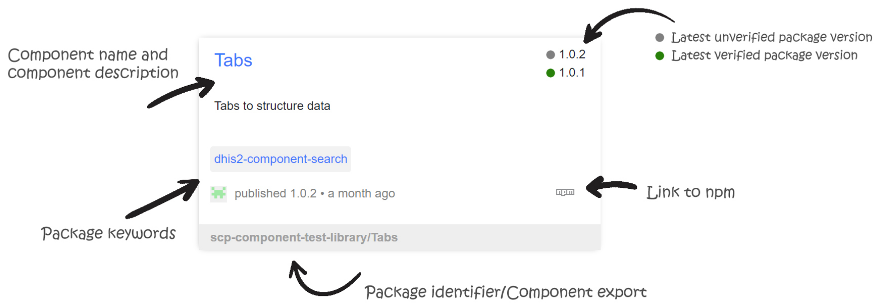

Component name and component description
Tabs
Tabs to structure data
● 1.0.2
● 1.0.1
● Latest unverified package version
● Latest verified package version
dhis2-component-search
published 1.0.2 • a month ago
Link to npm
Package keywords
scp-component-test-library/Tabs
Package identifier/Component export

Figure 1: GitHub Logo

6. Maintaining the SCP whitelist

DHIS2 SCP whitelist repository contains a list of verified NPM packages. Pull requests to this repository will be validated with a GitHub actions workflow.

If you are part of the The SCP whitelist maintaining team, you need to make sure you have the right access rights to be able to manage pull-requests.

A user submits his package for verification by modifying `list.csv` file, adding a new line containing his NPM package `identifier`and its `version` separated by a comma, e.g. `lodash,4.17.14`.It will result in a pull-request that will trigger the automated verification workflow on the added package.

The verification workflow:

- Checks for verification prerequisites defined in section **3.1 Verification prerequisites**
- Lints the code with ESLint and outputs the result
- Runs npm audit and outputs the result

The output of the verification workflow serves as a basis for acceptance of the pull request, but the final decision for acceptance is up to the maintaners.

Appendix C

SCP Tutorial

Process for creating, publishing and submitting for verification

This is a rough process of how to create, publish and submit a npm package for verification. Some details here may be different for you, the commands are just examples.

1. Create a GitHub repo (don't add a README, .gitignore or choose a license). e.g. https://github.com/goudbes/d2scp-test-components
2. Create a project locally with `create-react-library` `npx create-react-library --template default --manager npm --repo https://github.com/goudbes/d2scp-test-components goudbes-d2scp-test-components`
3. Follow the instructions at github for "push an existing repository from the command line".
4. Run local verification `npx -p "https://github.com/haheskja/scp-cli#master" dhis2-scp-cli -vvv verify`
5. Tag to the version in package.json `git tag v1.0.0`
6. Update the version in package.json `"version": "1.0.1"`
7. Commit package.json with the new version
8. Push the created tag `git push --tags`
9. Login to npm with `npm login`
10. Checkout the tag `git checkout v1.0.0`
11. Publish the tag to npm `npm publish`

Verification:

1. Go to https://github.com/dhis2designlab/scp-whitelist/blob/main/list.csv
2. Create a pull request for adding the package (`goudbes-d2scp-test-components,1.0.0`)

Appendix D

Consent form

Are you interested in taking part in the research project *"Shared Component Platform"*?

This is an inquiry about participation in a research project where the main purpose is designing, developing, and evaluating a platform that facilitates the sharing of reusable web components for use in web-based app development for the DHIS2 software platform. In this letter, we will give you information about the purpose of the project and what your participation will involve.

Purpose of the project
Practically, this research project is about designing, developing, and evaluating a platform that facilitates this sharing of reusable web-based components. Features that may be relevant are typical aspects from open source community projects such as the ability to upload components, facilitate automatic peer-reviews, and motivational mechanisms such as badges, rewards, and rankings based on contributions. Theoretically, knowledge about the process and result of the practical work can contribute with interesting findings to research on generic software implementation, open-source communities, and software platform design.
This research project will serve as a basis for the master's thesis of master's students at the University of Oslo.

Who is responsible for the research project?
The Department of Informatics at the University of Oslo is responsible for the project. The project is undertaken by [redacted] These master's students are supervised by [redacted]

Why are you being asked to participate?
You are selected for participation in the interview on the basis of your role and position, and activities related to the development of DHIS2 applications.

What does participation involve for you?
If you choose to take part in the project, this will involve ongoing cooperation, including, but not limited to:

- Interviews
- Prototype evaluations
- Discussions

Depending on the situation, some or all of these meetings will be recorded electronically. The cooperation period will start from the signing of this consent form and last until the 1st of July 2021. The period can be prolonged upon negotiation by both parties.

Participation is voluntary
Participation in the project is voluntary. If you choose to participate, you can withdraw your consent at any time without giving a reason. All information about you will then be made anonymous. There will be no negative consequences for you if you choose not to participate or later decide to withdraw.

Your personal privacy – how we will store and use your personal data

We will only use your personal data for the purpose(s) specified in this information letter. We will process your personal data confidentially and in accordance with data protection legislation (the General Data Protection Regulation and Personal Data Act).
The recorded interview will be transcribed and the transcription will be limited to the Component Team and their supervisors.
The data that includes PII will be stored on Google drive and access to it will be restricted using Google Drive authentication and authorization system. The transcript of the interview will be anonymized by replacement of the names of the participants with pseudonyms and care will be taken to ensure that other information in the interview that could identify the participant is not revealed. The anonymized interview transcript may be used in academic publications which include the Component Team's master's theses.

What will happen to your personal data at the end of the research project?
The project is scheduled to end before the 1st of January 2022. Any data containing PII will only be stored until the 1st of January 2022.

Your rights
So long as you can be identified in the collected data, you have the right to:
- access the personal data that is being processed about you
- request that your personal data is deleted
- request that incorrect personal data about you is corrected/rectified
- receive a copy of your personal data (data portability), and
- send a complaint to the Data Protection Officer or The Norwegian Data Protection Authority regarding the processing of your personal data

What gives us the right to process your personal data?
We will process your personal data based on your consent.

Based on an agreement with the University of Oslo, NSD – The Norwegian Centre for Research Data AS has assessed that the processing of personal data in this project is in accordance with data protection legislation.

Where can I find out more?
If you have questions about the project or want to exercise your rights, contact:
[redacted]
- The University of Oslo [redacted]
- Our Data Protection Officer: Roger Markgraf-Bye by email: personvernombud@uio.no or by telephone +47 90 82 28 26.
- NSD – The Norwegian Centre for Research Data AS, by email: (personverntjenester@nsd.no) or by telephone: +47 55 58 21 17.

Yours sincerely,

The Component Team and their supervisors

Consent form

I have received and understood information about the project *Shared Component Platform* and have been given the opportunity to ask questions. I give consent:

- ☐ to participate in the project
- ☐ for my personal data to be processed outside the EU

I give consent for my personal data to be processed until the end date of the project, approx. *1st of January, 2022.*

(Signed by participant, date)

Appendix E

Learning goals

Learning goals

1. Understanding when and why web-based DHIS2 apps are developed.
2. Understanding the general DHIS2 web app development practices and process.
    - Understand technical aspects of the app development process
    - Understanding social aspects of the app development process (collaboration in co-located teams and between geographically dispersed teams), project member roles, and level of prioritization.
    - Understanding the standards and principles you must or want to comply with.
3. Understanding the process of making apps generic (relevant beyond a single implementation/ user organization).
4. Understanding if, how, why, and what web components, frameworks, tools, repositories, etc. take part in their app development practices.
5. Understanding current and prospective reuse practices (e.g., use of react/npm components).
    - Understanding motivations for reuse.
    - Understanding pertinent improvements that could be made and their plausibility.
    - What are the impediments for reuse?
6. Understanding how we can collaborate further on app development resources/ component libraries.

Appendix F

Interview guide

Interview Guide

1. Your role in the organization and what are you working with?
2. What is the structure of the org/group?
    a. Developer background
    b. Academic influence
    c. Hierarchy
3. Please take us through the process of app development, focusing both on technical and social aspects. This may include:
    - Project management framework - Agile, different teams,
    - Tools you're using for development and collaboration
    - Is everything written from scratch or do you reuse components?
    - What kind of components are you interested in reusing?
    - Where do you store and share components?
    - What design principles and standards do you comply with?
        - Does the client want you to adhere to certain standards?
        - Who is enforcing the standards? Internally? Externally?
        - DHIS2 core standards, apps similar to generic apps.

    How do you experience the design infrastructure set up for DHIS2?
4. Can you explain how a project is set up?
    a. Are there many project members?
    b. Are there any ongoing projects at the same time?
    c. Do you collaborate and talk across projects?
5. Can you tell us a bit about your typical workday?
    a. When do you start/end? Breaks? Social events?
6. How do you currently communicate with?
    a. Hisp groups?
    b. Internally?
    c. What about the core team?
    d. Which barriers, if any, impede this communication?
    e. From your perspective, what could be done about this?
7. Why do you want to participate in this project? From your perspective; what improvement do you consider plausible?
8. Do you have anything else you want to share that you feel relevant to our study?

- Can you take us through the process from starting a new project/app and to completion?
- More technical, can you explain how you start developing? What do you focus on? Do you use any tools?
- Do you have any mechanisms for trying to reuse code across apps?

Appendix G

Surveys

DHIS2 Shared Component platform evaluation survey for component owners

This survey is for the evaluation of the DHIS2 Shared Component Platform and is part of my master's thesis at the University of Oslo. The tasks and questions in this survey are meant to cover the process of sharing reusable components and making them available on the SCP.

Participation in the project is voluntary. If you choose to participate, you can withdraw your consent at any time without giving a reason. All information about you will then be made anonymous. There will be no negative consequences for you if you choose not to participate or later decide to withdraw. By filling out this survey, you agree to participate in this project.

Before proceeding with the survey, you need to make sure you have an account on https://github.com (to store your code) and an account on https://www.npmjs.com (for publishing npm packages).

*Required

Component owner evaluation

DHIS2 Shared Component Platform is a platform for sharing and reuse of software components created by the HISP community. The components hosted on this platform are React or Angular UI components and can be used as building blocks for a web application.

In this task you take the role of a component owner - a developer who creates reusable ui components and shares them for further reuse.
Your goal is to publish one or several reusable components and make them appear on the SCP website.
Additionally, you have to use the DHIS2 SCP command line interface to verify your package locally, and then submit your package for verification on the DHIS2 Whitelist repository.

Link to the SCP website: https://dhis2designlab.github.io/scp-website/
Link to the SCP command line interface: https://github.com/dhis2designlab/scp-cli
Link to the SCP whitelist repository: https://github.com/dhis2designlab/scp-whitelist

You can proceed with this task by looking at the SCP documentation available here:
url

After you have finished, you can begin answering this survey.

1. What is your occupation? *

Mark only one oval.

- ◯ Student at the UiO
- ◯ DHIS2 core team member
- ◯ HISP groups local developer
- ◯ Other: ____________________

2. How much experience do you have in software development? *

Mark only one oval.

- ◯ < 1 year
- ◯ 1-2 years
- ◯ 2-5 years
- ◯ > 5 years

3. How much experience do you have with the DHIS2? *

Mark only one oval.

- No experience
- Very little experience
- Some experience
- A lot of experience

4. Give a brief account of your previous reusable component creation experience *

Your answer should include the nature of the components (e.g. web components, backend components), what process you used and what tools, frameworks and platforms you used.

5. What aspects of the reusable component creation experience do you find the most challenging and why? *

6. Was it clear to you what part of the DHIS2 SCP documentation covered your task and your role as a component owner? *

Mark only one oval.

- Not clear
- Somewhat clear
- Clear enough
- Very clear

7. Optionally, provide some reflection on the clarity of the documentation and suggest improvements

8. How important is documentation to a system like SCP, in your opinion? *

Mark only one oval.

- ◯ Not important
- ◯ Somewhat important
- ◯ Important
- ◯ Very important

9. How intuitive was the process of listing the components in the package.json file? *

Mark only one oval.

- ◯ Not intuitive
- ◯ Somewhat intuitive
- ◯ Intuitive enough
- ◯ Very intuitive

10. Optionally, provide feedback on how intuitive the process of listing the components in package.json file was

11. How does the DHIS2 SCP affect the reusable component publishing experience in your opinion? *

Mark only one oval.

- ◯ Makes it worse
- ◯ Does not really change it
- ◯ Makes it better

12. Reflect on how the DHIS2 SCP affects the reusable component publishing experience *

13. Compare the time spent publishing reusable components to the DHIS2 SCP with your previous experience with publishing reusable components *

Mark only one oval.

- No relevant or comparable previous experience
- Publishing the components for the SCP took significantly longer
- Publishing the components for the SCP took a little bit longer
- More or less the same amount of time
- Publishing the components for the SCP was a little bit faster
- Publishing the components for the SCP was significantly faster

14. Was it clear to you what components you should list in package.json file? *

Mark only one oval.

- Not clear
- Somewhat clear
- Clear enough
- Very clear

15. Optionally, provide some reflection on the clarity on what components to list in the package.json and suggest improvements

16. The SCP focuses on web components as the level of abstraction (specifically React and Angular web components) as opposed to focusing on NPM packages like npmjs.com does. Do you think the focus on web components is the appropriate level of abstraction? Reflect on it. *

17. How easy was it to perform local verification? *

Mark only one oval.

- Not easy
- Somewhat easy
- Easy enough
- Very easy

18. Describe any issues you have experienced while perfoming local verification.

19. Describe your understanding of verification in the DHIS2 SCP and its purpose. *

20. Describe your understanding of the purpose of local verification with the SCP CLI. *

21. How easy was it to apply for package verification in the SCP Whitelist repository? *

Mark only one oval.

- Not easy
- Somewhat easy
- Easy enough
- Very easy

22. Optionally, describe any issues you have experienced while applying for verification in the SCP whitelist.

____

____

____

____

____

23. How clear are the requirements for package verification to you? *

Mark only one oval.

- ◯ Not clear
- ◯ Somewhat clear
- ◯ Clear enough
- ◯ Very clear

24. How confident are you that your package will be verified if it meets all the verification requirements? *

Mark only one oval.

- ◯ Not confident
- ◯ Somewhat confident
- ◯ Confident enough
- ◯ Very confident

25. How important is it to have clear and objective verification requirements for a system like the DHIS2 SCP? *

Objective verification requirements would imply that the requirements are not open to different interpretations or the opinion of any specific person.

Mark only one oval.

- ◯ Not important
- ◯ Somewhat important
- ◯ Important
- ◯ Very important

26. Reflect on the processing of applying for verification in the SCP whitelist and how it could be improved. *

27. In your opinion, how standard and pervasive are the technologies that the DHIS2 SCP requires you to use when publishing reusable components? *

The technologies would be Git, npm, Node.js, GitHub, etc...

Mark only one oval.

- Not standard and pervasive
- Somewhat standard and pervasive
- Standard and pervasive enough
- Very standard and pervasive

28. Optionally, provide some reflection on the technology that the DHIS2 SCP requires you to use when publishing reusable components.

29. In your opinion, is it important to use standard and pervasive technologies in a platform like the DHIS2 SCP? *

Mark only one oval.

- Not important
- Somewhat important
- Important
- Very important

30. Optionally, reflect on the importance of the use of standard and pervasive technologies in a platform like the DHIS2 SCP.

31. Is there anything else you would want to mention?

32. Do you have any comments on this survey?

Thank you for your participation

DHIS2 Shared Component platform evaluation survey for component consumers

This survey is for the evaluation of the DHIS2 Shared Component Platform and is part of my master's thesis at the University of Oslo. The tasks and questions in this survey are meant to cover the process of finding reusable components.

Participation in the project is voluntary. If you choose to participate, you can withdraw your consent at any time without giving a reason. All information about you will then be made anonymous. There will be no negative consequences for you if you choose not to participate or later decide to withdraw. By filling out this survey, you agree to participate in this project.

DHIS2 Shared Component Platform is a platform for sharing and reuse of software components created by the HISP community. The components hosted on this platform are React or Angular UI components and can be used as building blocks for a web application.

For this survey you take a role of a component consumer - a developer who uses reusable ui components to build web-applications. The SCP website will show all available ui components submitted to SCP.

Before proceeding with the survey, you should browse through the SCP website:
https://dhis2designlab.github.io/scp-website/

and look through the documentation

Evaluate the website as if you are using it to find components for reuse in a web-application you are developing.

*Required

1. What is your occupation? *

   Mark only one oval.

   - ⬭ Student at UiO
   - ⬭ DHIS2 core team member
   - ⬭ HISP group local developer
   - ⬭ Other: ______________________

2. How much experience do you have in software development? *

   Mark only one oval.

   - ⬭ < 1 year
   - ⬭ 1 - 2 years
   - ⬭ 2 - 5 years
   - ⬭ > 5 years

3. How much experience do you have with the DHIS2? *

Mark only one oval.

- No experience
- Very little experience
- Some experience
- A lot of experience

4. Give a brief account of your previous experience of finding and using reusable components *

5. What aspects of finding and using reusable components you find most challenging and why? *

6. Was it clear to you what part of the DHIS2 SCP documentation covered your task and your role as a component consumer? *

Mark only one oval.

- Not clear
- Somewhat clear
- Clear enough
- Very clear

7. Optionally, reflect on the clarity of the documentation and suggest improvements

8. How important is the documentation of a system like SCP for component consumers in your opinion? *

Mark only one oval.

- Not important
- Somewhat important
- Important
- Very important

9. How easy was it to find components on the SCP website? *

Taking into account that currently there is a limited amount of the components available.

Mark only one oval.

- Not easy
- Somewhat easy
- Easy enough
- Very easy

10. Optionally, provide additional feedback on how easy it was to find components on the SCP website.

11. How clear was the component representation on the SCP website? *

Mark only one oval.

- Not clear
- Somewhat clear
- Clear enough
- Very clear

12. Optionally, reflect on the clarity of the component representation of the SCP website.

13. How can we improve the SCP website? *

14. Describe your understanding of the purpose of the "Show only verified components" filter on the SCP website *

15. Describe your understanding of the version indicators on the component cards. *
If this question is unclear, see the documentation.

16. Please provide some feedback on your component search experience on the SCP website. *

17. Please provide some feedback on the filtering experience on the SCP website. *

18. Please provide some feedback on the clarity of the information presented on the SCP website. *

19. How does the SCP affect experience of finding reusable components in your opinion? *

Mark only one oval.

- Makes it worse
- Does not change it
- Makes it better

20. Reflect on how the SCP affects your experience of finding reusable components *

21. Describe any issues you have experienced while using the SCP website?

22. The SCP focuses on web components as the level of abstraction (specifically React and Angular web components) as opposed to focusing on NPM packages like npmjs.com does. Do you think the focus on web components is the appropriate level of abstraction? Reflect on it.

If you have previously answered this question in a different survey, you can skip it.

23. Is there anything else you would want to mention?

24. Do you have any comments on this survey?

Thank you for your participation ♡

DHIS2 Shared Component platform evaluation survey for whitelist maintainers

This survey is for the evaluation of the DHIS2 Shared Component Platform and is part of my master's thesis at the University of Oslo. The tasks and questions in this survey are meant to cover the process of maintaining the SCP whitelist.

Participation in the project is voluntary. If you choose to participate, you can withdraw your consent at any time without giving a reason. All information about you will then be made anonymous. There will be no negative consequences for you if you choose not to participate or later decide to withdraw. By filling out this survey, you agree to participate in this project.

DHIS2 Shared Component Platform is a platform for sharing and reuse of software components created by the HISP community. The components hosted on this platform are React or Angular UI components and can be used as building blocks for a web application.

For this survey you take a role of a whitelist maintainer - a person responsible for processing component verification submissions.

Before proceeding with the survey, you should read the SCP documentation, that you can find here https://github.com/goudbes/scp-evaluation/blob/master/documentation/documentation.md

You should also look at the SCP whitelist repository https://github.com/dhis2designlab/scp-whitelist

and check the open pull requests and the verification output.
For example: https://github.com/dhis2designlab/scp-whitelist/pull/17/checks?check_run_id=1546505461

If you have answered any of the questions before, just state that in the answer instead of answering it again.

*Required

1. What is your occupation? *

Mark only one oval.

- ◯ Student at UiO
- ◯ DHIS2 core team member
- ◯ HISP group local developer
- ◯ Other: ____________________

2. How much experience do you have in software development? *

Mark only one oval.

- ◯ <1 year
- ◯ 1-2 years
- ◯ 2-5 years
- ◯ >5 years

3. How much experience do you have with the DHIS2? *

   Mark only one oval.

   - No experience
   - Very little experience
   - Some experience
   - A lot of experience

4. Give a brief account of your previous experience with component reuse *

5. Describe your understanding of the purpose of package verification *

6. DHIS2 SCP verifies individual versions of the packages instead of verifying only a package without any specific version. What do you think of this approach? *

7. In your opinion, what checks would be important in package verification? *

8. Describe any issues you have experienced while assessing the whitelist and the verification workflow *

9. Do you have any suggestions for improvements of the verification workflow? *

10. How clear it for you whether a package verification request should be accepted or rejected? *

Mark only one oval.

- Not clear
- Somewhat clear
- Clear enough
- Very clear

11. What assessment criteria (that is not implemented in the SCP) would you use to decide whether or not a package should be added to the list of verified packages? *

12. What additional checks should the verification pipeline perform in your opinion? *

13. Do you think that the package verification process should have minimal human discretion and thus be objective, or should it be subject to the opinions of whitelist maintainers? Please reflect on it. *

Objective verification example in the SCP: check that package.json file includes the keyword. If it does not, the verification pipeline fails. Subjective verification examples in the SCP: npm audit. Another examples of subjective verification: considering the experience of the component owner, looking through the issues on the package's github repository without any well-defined criteria on what would be acceptable or not. Subjective would also imply that different whitelist maintainers might have a different opinion whether a package should be whitelisted or not.

14. Was it clear to you what part of the DHIS2 SCP documentation covered your task and your role as a whitelist maintainer? *

Mark only one oval.

- Not clear
- Somewhat clear
- Clear enough
- Very clear

15. Optionally, reflect on the clarity of the documentation and suggest improvements

16. How important is the documentation of a system like SCP for whitelist maintainers in your opinion? *

Mark only one oval.

- Not important
- Somewhat important
- Important
- Very important

17. In your opinion, how standard and pervasive are the technologies that the DHIS2 SCP requires you to use when verifying the packages? *

Technologies: Github and Github Actions, CSV and etc.

Mark only one oval.

- Not standard and pervasibe
- Somewhat standard and pervasive
- Standard and pervasive
- Very standard and pervasive

18. Optionally, provide some reflection on the technology that the DHIS2 SCP requires you to use when verifying the packages

19. In your opinion how important is it to use standard and pervasive technologies in a platform like SCP? *

Mark only one oval.

- Not important
- Somewhat important
- Important
- Very important

20. Optionally, reflect on the importance of the use of standard and pervasive technologies in a platform like the DHIS2 SCP

21. Is there anything else you would like to mention?

22. Do you have any comments on this survey?

Thank you for your participation

Appendix H

Conference abstract

Anastasia Bengtsson

(University of Oslo, Oslo)

FACILITATING SOFTWARE COMPONENT REUSE IN THE DHIS2 PLATFORM ECOSYSTEM

В данной работе рассматривается разработка репозитория для хранения и сертификации компонентов для повторного использования кода в экосистеме платформы и информационной системы здравоохранения DHIS2.

There is a growing trend towards using the component-based software engineering (CBSE) approach in development of web applications. Software reuse is the central part of this approach and the main idea behind it is development of applications by reusing configurable software components. However, there are several barriers for component reuse, and one of them is poor cataloguing, distribution of reusable software components and a lack of component certification. This will have considerable impact on component discovery and trustworthiness and make the process of component reuse less effective.

This study focuses on DHIS2, a generic web-based health management information system platform and the process of development of web applications on top of it, as an extension of its functionality. The practical aim of this project was to design, implement, and evaluate a component repository to improve component reuse in the DHIS2 ecosystem. The project involved a close collaboration with DHIS2 developers who provide support to DHIS2 in East Africa region, as well as with the members of the DHIS2 core team at the University of Oslo. The component repository consists of a website (built using React) that aggregates reusable components, and two other modules that support the process of component certification: a command line interface (built using TypeScript) to provide functionality for local certification, and a GitHub repository with an automated certification workflow using GitHub Actions workflow. The component repository aims to increase productivity of DHIS2 developers and shorten the development life cycle. Component certification improves component trustworthiness and thus, improves quality and reliability of the developed web applications.

The process of design and development of the component repository was guided by the Design Science Research methodology. The theoretical aim of the research was to establish a set of theoretically and empirically grounded design principles that contribute to the knowledge base of CBSE on how to implement component repositories in a platform ecosystem.

Appendix I

Conference abstract

FACILITATING SOFTWARE COMPONENT REUSE IN THE DHIS2 PLATFORM ECOSYSTEM

A. Bengtsson

University of Oslo

Oslo, Norway

Scientific advisors P. Nielsen and M. Li

There is an increase in the development of generic software systems that are developed to serve multiple organizations and used for different purposes. Some examples of generic software are the Microsoft Office 365 suite, Adobe Photoshop, and DHIS2 - a generic web-based Health Management Information System (HMIS) platform, which is the focus of my study. The purpose of HMIS is to routinely manage and generate health information data that would serve as a basis for management decisions to foster improvements in health service provision. DHIS2 is currently the world's largest health management information system, and it is in use by 73 low- and middle-income countries **[1]**. HISP is a global network that develops and supports the DHIS2 platform. The network is comprised of HISP groups – organizations based in developing countries, providing support to DHIS2.

One way of contributing to the DHIS2 is through the development of additional modules or web applications on top of generic software, which are extensions of the user interface and the functionality in the case of the DHIS2. Building these web applications from scratch can be time-consuming. It is also not resource-efficient if different HISP groups are developing similar modules. One way of addressing this problem is by building software from existing components using a component-based software engineering (CBSE) approach. Software reuse is the central focus of this approach, and the main idea is a development of applications by reusing configurable software components. However, there are several barriers to component reuse, and one of them is the poor cataloging and distribution of reusable software components. This has a considerable impact on component discovery and makes the process of component reuse less effective.

This study aimed at attaining two goals — a practical one and a theoretical one. The practical goal was to conduct engaged research with the HISP community exploring the possibility of creating a component repository that facilitates component reuse in web-based application development. Therefore, a primary focus of the work in this project was the design, implementation, and evaluation of such a repository in collaboration with the DHIS2 core team and members of HISP groups involved in application development work. The theoretical goal of my research was to identify and establish a set of theoretically and empirically grounded design principles for implementing a component repository that facilitates component reuse in a software platform ecosystem. These design principles are a theoretical contribution to the knowledge base on how component repositories can be designed and developed. They are prescriptive in nature and are meant to give value beyond local practice. Given the above, the paper addresses the following research question: *What are the essential design principles for implementing a component repository that facilitates component reuse in a software platform ecosystem?*

Guided by the nature of the research problem, this study was situated within the pragmatic research paradigm. Software reuse is a socio-technical activity, as it clearly has some social aspects in addition to technical aspects. For example, the specification of metadata for a component is highly technical, as it must be exactly specified and machine-readable to ensure proper component cataloging in a component repository. There are, however, also highly social aspects, for example, the developers' attitude towards software reuse, which could be influenced by social factors such as trust and understanding. Given this, I could see a clear application for the pragmatic research paradigm that advocates embracing the approach that gives most utility in the circumstances. I have chosen Design Science Research (DSR) as an overarching methodology to guide the design and development of the component repository. Contrary to other methodologies that have a goal of understanding reality, DSR is a problem-solving approach with the aim of changing situations to a better or more desirable state. DSR offers a cyclical process model that includes such activities as problem identification, definitions of the objectives, design and development of the artifact, demonstration, evaluation of the artifact, and, finally, communication of the conducted research.

To identify the problem, my team and I have conducted focus group discussions with the members of the DHIS2 core team at the University of Oslo. Additionally, we conducted a set of interviews with developers in HISP East Africa. Our goal was to learn about application development practices, motivation for software reuse, current and prospective reuse practices, impediments for reuse, tooling, and collaboration in co-located teams (i.e., within one HISP group) and geographically dispersed teams (i.e., between different HISP groups). Analysis of the gathered data has shown that there is diversity in technology, tooling, and software reuse practices. One of the practices discerned during the interviews is software reuse through the copying of code, and while it can be seen as code reuse with minimal effort, there is a number of issues pertaining to such a practice. The code might have bugs and security vulnerabilities, and copy-pasting would mean introducing these issues in different applications. Another practice we have encountered during the interviews was CBSE, which involved the creation of reusable components which were stored on Github and as NPM packages in NPM Registry. This has made us question whether there is, in fact, a need for the development of a component repository given that NPM Registry is already in place. We have decided not to develop a completely new component repository but rather cultivate the installed base by reusing and extending the existing infrastructure. The main goal of our solution would be to support and improve the existing CBSE approach by addressing some of the challenges we have encountered with existing technologies, services, and tools.

As a practical contribution to this study, a component repository called the DHIS2 Shared Component Platform (SCP) was developed. The component repository consists of a website (built using React) that aggregates reusable components and two other modules that support the process of component certification: a command-line interface (CLI, written in TypeScript) to provide functionality for local certification, and a GitHub repository with an automated certification workflow using GitHub Actions workflow that invokes the command line interface. During the development phase, SCP was evaluated by the DHIS2 core team members with the intention to improve SCP's functionality and develop a higher quality artifact. SCP aims to increase the productivity of DHIS2 developers and shorten the development life cycle. Component certification improves component trustworthiness and thus, improves the quality and reliability of the developed web applications. The established set of design principles, a theoretical contribution of this study, attempts to address the challenging aspects of the implementation of a component repository that facilitates component reuse in a software platform ecosystem. These principles can serve as guidance for the construction of a similar artifact.

The first design principle, *Principle of installed base cultivation*, advocates the utilization of the existing infrastructure to increase the likelihood of component repository adoption. The process of design and development should not start from scratch; it must consider the existing infrastructure, e.g., attitude towards software reuse, software reuse practices and process, technology, and tooling. Instead of creating a radical change, one should cultivate the installed base towards better practice.

The second design principle, *Principle of component trustworthiness*, advocates the implementation of component certification as an integral part of software reuse in order to increase component trustworthiness and make developers more comfortable reusing software. The review of the previous literature on CBSE has shown that component certification is an important aspect of CBSE, and the DHIS2 core team has also expressed the need for certification functionality to promote components with a certain level of quality. When implementing certification, one must take into consideration the level of human discretion in the certification process. A certification process with a low level of human discretion can be automated and more accurate, while a manual process with a high level of human discretion can be subjective, time-consuming, and less accurate.

The third design principle, *Principle of balanced certification*, emphasizes the importance of governance balance in a software platform ecosystem when choosing individuals for the role of component certifiers. If the DHIS2 core team, as platform owners, takes this responsibility, it might have a significant impact on the autonomy of third-party developers. If the team of certifiers is entirely comprised of third-party developers, it brings more egalitarianism to the platform ecosystem but reduces the platform owners' control over the platform.

The fourth design principle, *Principle of component granularity,* advocates providing the right level of component granularity in a component repository as it has a high impact on a component's discoverability and usability. NPM packages have an arbitrary level of component granularity, i.e., some packages might contain only one reusable component, while some packages act as component libraries and contain multiple components. This has a negative effect on the component discovery, as NPM registry does not search for components within packages. SCP addresses this challenge by indexing reusable components inside the packages and thus, improves their discoverability.

The fifth design principle, *Principle of orthogonality*, guides the researchers and developers in their work on architecting and implementing a component repository. A component repository is part of the component-management process and must provide support for other processes such as component publishing, component acquisition, and certification. Adopting a modular approach with the aim of building an orthogonal system, i.e., highly cohesive and loosely coupled, can reduce the complexity of the system and increase its maintainability. A high degree of orthogonality has a significant impact on the system's evolution, as each of the modules can evolve in a decentralized way (i.e., the modules can be modified, updated, and removed independently from each other).

Bibliography

1 About DHIS2 [Electronic resource]. – 2021. – Access mode: https://dhis2.org/About/. – Accessed: 14.02.2021.