Enabling software component reuse in enterprise software platform ecosystems

Design principles for component management systems

Håkon André Heskja

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

Department for Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

May 2021

Enabling software component reuse in enterprise software platform ecosystems

Design principles for component management systems

Håkon André Heskja

Abstract

A persistent challenge for vendors who design enterprise software is to make it usable across different contexts through a process of localization. Increasingly, more vendors are adopting software platform strategies in business-to-business markets in an attempt to address this challenge, which has transitioned them into curators of ecosystems where actors collaboratively develop and commercialize a shared technology. Based on the heterogeneity of these ecosystems, localization tasks are consigned to regional implementation groups with greater experience in their native markets. One approach to localization that provides a high level of design flexibility is developing apps, but such an approach is resource-intensive. Because of this, implementation groups create other self-resourced initiatives such as software components to aid themselves. A challenge is that software components created by implementation groups to aid them during localization are not reused. As many of these implementation groups face the same challenges during localization, reusing these software components can be valuable. However, existing literature provides no prescriptive knowledge on how to accomplish this. By conducting a design science research project and collaborating with the vendor of an enterprise software platform and two implementation groups, this thesis explores developing a component management system to enable software component reuse. The practical contribution comes in the form of a component management system that enables reusing software components. The theoretical contribution from this project is four design principles regarding component management systems in enterprise software platform ecosystems: the *Principle of an interface for component discovery*, the *Principle of component certification*, the *Principle of infrastructure for hosting and distribution*, and the *Principle of the package-to-component abstraction*.

Acknowledgment

First, I would like to thank my supervisor Petter Nielsen for guiding me through the process of writing this thesis with invaluable feedback and support. I would also like to thank my co-supervisor Magnus Li for initiating this study and always being available for discussions.

Thanks to the collaborating members of the DHIS2 Core Team, HISP Mozambique, and HISP Tanzania for their participation in this study. I want to offer a special thanks to Austin McGee for the ideas and insights throughout the process. Then I would like to thank the members of the DHIS2 Design Lab, and especially its coordinators, Anders Erik Brustad and Hanna Kongshem, for their efforts in creating a positive learning environment.

Special thanks to my colleague Anastasia Bengtsson for the hard work on the project, for the motivation, and for the countless insightful discussions. Also, thanks to my colleague Simeon Andersen Tverdal for the cooperation.

Last, I would like to thank my friends and family for all their support and motivation.

Table of Content

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface.
CBSE	Component-Based Software Engineering.
CLI	Command-Line Interface.
DGPG	Digital Global Public Goods.
DHIS2	District Health Information Systems version 2.
DSR	Design Science Research.
DSRM	Design Science Research Methodology.
ERP	Enterprise Resource Planning.
IS	Information Systems.
IT	Information Technology.
HIS	Health Information System.
HISP	Health Information Systems Programme.
NGO	Non-Governmental Organization.
NPM	Node Package Manager.
SCP	Shared Component Platform.
SDK	Software Development Kit.
UiO	University of Oslo.

Chapter 1: Introduction

A persistent challenge for vendors who design enterprise software is to make it usable across different contexts. Increasingly, more vendors are adopting software platform strategies in business-to-business markets in an attempt to address this challenge (Wareham et al., 2013). Tiwana (2013, p. 5) defines a software platform as a "software-based product or service that serves as a foundation on which outside parties can build complementary products or services." This transition to platform strategies has turned vendors into curators of ecosystems where businesses collaboratively develop and commercialize a shared technology (Foerderer et al., 2019). Based on the extreme heterogeneity that characterizes such ecosystems, these vendors consign localization tasks to "regional [implementation groups] with far greater expertise in their native markets" (Wareham et al., 2013, p. 1196). Further, Wareham et al. (2013) refer to such localization tasks as specific modifications that adapt the platform to local needs. One way of tackling localization tasks is by extending the functionality of the software platform through third-party apps (Li & Nielsen, 2019). Li & Nielsen (2019) explain that such third-party apps provide a high level of design flexibility but are resource-intensive. To support implementation groups in this process, vendors of enterprise software platform ecosystems provide boundary resources such as application programming interfaces (APIs) and software development kits (SDKs) (Ghazawneh & Henfridsson, 2013). Additionally, Li & Nielsen (2019) documented that some implementation groups create component libraries for use in their projects and attempt to distribute them to other implementation groups in the ecosystem. However, these are not efficiently reused, causing implementation groups to develop components that already exist somewhere else in the ecosystem.

In this thesis, I have worked with the vendor of the District Health Information Software 2 (DHIS2), an enterprise software system increasingly following a platform strategy. This vendor collaborates with several local implementation groups who specialize in modifying the generic platform to local needs. Additionally, the vendor increasingly sees that implementation groups create third-party apps to address localization tasks, as it gives them a high level of design flexibility. Based on this, Li (2019a) proposed a research project to explore how these components can be reused.

Based on DHIS2's adoption of React as their primary framework for developing web interfaces, much of the ecosystem has already started to follow, which brings this inherent focus on components with it. Reusing components can reduce the app's time in development, make it easier to maintain, and raise its quality (Sommerville, 2016). Furthermore, it allows the implementation groups to compose apps from reusable components instead of developing everything from scratch. Unfortunately, even though implementation groups create component libraries, having no simple, predetermined way of sharing them with others in the DHIS2 ecosystem means most of them are only reused by those who created them. This thesis explores creating a component management system to enable reusing software components created by these implementation groups with others in the DHIS2 ecosystem.

The presented research problem can be described on three progressively narrower levels. The first overarching level is that generic enterprise software does not work optimally if it is not localized (Li & Nielsen, 2019; Wareham et al., 2013). The second level is that developing apps as an approach to localization is resource-intensive for implementation groups (Li & Nielsen, 2019; Wareham et al., 2013). The third level and this thesis' case is that software components created by implementation groups to aid them during localization are not reused. By enabling the reuse of software components created by implementation groups, developing apps can become less resource-intensive, thereby supporting the localization of enterprise software. In the end, this can help make generic enterprise software work more optimally. The hope is that contributing to the third level can make it easier to localize. This thesis focuses on the problem on the third level, that software components created by implementation groups to aid them during localization are not reused. This thesis explores creating a component management system that enables the reuse of software components by enabling implementation groups to publish their components and reuse others'.

1.1 Component-based software engineering

Literature on component-based software engineering (CBSE) adds a relevant vocabulary to discuss the component management system and add justificatory knowledge used during its construction. Furthermore, it adds context to analyze implementation groups' existing software component reuse processes and potential barriers to successful reuse.

CBSE "is the process of defining, implementing, and integrating or composing" components into systems (Sommerville, 2016, p. 465) and is a proven way of efficiently reusing software (Kumar & Tiwari, 2020). Even though CBSE promises more efficient development processes, there are also several known barriers to efficient component reuse (B et al., 2010). Some barriers include poor distribution of components, poor cataloging, and lack of certification metrics. CBSE consist of two high-level processes (Kotonya et al., 2003; Kumar & Tiwari, 2020; Sommerville, 2016). Development for reuse is concerned with the development of reusable components by component producers and their subsequent storage through a component management system. Development with reuse is concerned with composing already available components into applications by component users.

The primary focus of this thesis is on the "development for reuse" process and how components can be collected, certified, cataloged, maintained, and distributed through a component management system so that they are available for reuse.

1.2 Research context

The object of this study is the DHIS2 ecosystem. DHIS2 is an enterprise software platform developed at the University of Oslo (UiO) from the 1990s and to this date (Adu-Gyamfi et al., 2019). DHIS2 falls under the Health Information Systems (HIS) category and is developed under the Health Information Systems Programme (HISP) coordinated by the UiO. It was created after the fall of apartheid in South Africa and is now implemented and used in more than 100 developing countries (Adu-Gyamfi et al., 2019). The implementation groups that primarily implement DHIS2 are called HISP groups. The developers of the DHIS2 platform, the vendor, are referred to as the DHIS2 Core Team. The HISP groups work on implementing the DHIS2 software in their context. This implementation involves leveraging DHIS2s boundary resources to make the platform locally relevant and usable. Even though the HISP groups use many of the same resources, it is a diverse ecosystem. For example, the HISP groups are geographically dispersed and at different stages of maturity. Some HISP groups have highly competent development teams, while others are still in their inception. Additionally, some engage in the creation of resources that can be reused by them or others. During this project I have engaged with two HISP groups in Sub-Saharan Africa to understand their current and future practices in relation to app development and collaborated and discussed with the DHIS2 Core Team.

This thesis was conducted through the DHIS2 Design Lab, a part of the HISP project at UiO. Most of the practical parts of the process, such as design, development, and most data collection, have been done in collaboration with two other master's students from the DHIS2 Design Lab: Anastasia Bengtsson and Simeon Andersen Tverdal. My two fellow master's students will be referred to as my' colleagues,' and the three of us the 'research team.' The DHIS2 Design Lab explores issues related to how one can improve the usability and the utility of DHIS2 by engaging in implementation projects within the DHIS2 ecosystem (Li, 2019b). Members of the DHIS2 Design Lab meet and discuss their problems, challenges, and findings related to their projects and how to communicate this through their work. Being part of the DHIS2 Design Lab has allowed for greater access to the DHIS2 ecosystem and more avenues for learning.

1.3 Research aim

The aim of this project is both practical and theoretical. The practical aim is to explore, develop and evaluate a component management system within the DHIS2 ecosystem to enable the reuse of software components created by the implementation groups. Subsequentially, developing apps can become less resource-intensive, thereby supporting the localization of generic enterprise software. As such, the practical contribution comes in the form of a component management system that enables the reuse of software components by enabling implementation groups to publish their components and reuse others'. Theoretically, the thesis aims at identifying a set of design guidelines in the form of empirically and theoretically grounded design principles. The design principles are generic and offer guidance to enterprise software vendors in developing component management systems.

1.4 Research question

The following research question is used to guide the research and exploration of the features of a component management system in this thesis:

- *What are design principles for component management systems within enterprise software platform ecosystems?*

To answer the research question, a component management system was developed following the Design Science Research (DSR) methodology. DSR is a problem-solving paradigm that lends itself to creating and evaluating artifacts (Hevner et al., 2004). The following main objectives were performed by building on this paradigm and occurred in the following order:

1. Position and motivate the problem through internal discussion in the DHIS2 Design Lab and review of platform theory. Interviews with two HISP groups to get an overview of their app development and component reuse process.
2. Define objectives and goals through focus groups with the DHIS2 Core Team, the project supervisors, and the HISP groups by evaluating and discussing the component management system.

3. Design and develop the Shared Component Platform (SCP), the implemented component management system, by designing its architecture and programming its necessary functionality to support the reuse of software components created by the HISP groups. Create the initial set of design principles.
4. Perform formative evaluations on the outcome of the development process to measure the component management system's ability to solve the problem and the feasibility to create, operate, and maintain it.
5. Elaborate and justify the design principles based on evaluation results and theory and communicate these back to literature.

1.5 Expected contributions

The expected practical contribution of this project is a component management system that enables the reuse of software components by enabling implementation groups to publish their components and reuse others'. The expected theoretical contribution is a set of design principles for enterprise software platform vendors who want to enable software component reuse within their ecosystems. As such, they offer utility for vendors who want to build similar systems by guiding their design.

1.6 Thesis structure

The rest of the thesis is structured in the following manner:

- **Chapter 1 – Introduction:** Introduces the topics of the thesis and describes its motivation, research context, research aims, and research question.
- **Chapter 2 – Background:** Provides an overview of DHIS2, HISP, and the DHIS2 Design Lab.
- **Chapter 3 – Related Research:** Presents relevant research on digital platforms, enterprise platforms, boundary resources, localization, and component-based software engineering.
- **Chapter 4 – Research Approach:** Presents the abstract research approach, the research methods used, and the implemented research process.
- **Chapter 5 – Artifact Description:** Presents the component management system's role within the ecosystem, data collected during diagnosis and evaluations, the component management system's architecture, and the initial set of design principles.
- **Chapter 6 – Final Evaluation:** Presents the data collected on the component management system's final evaluation concerning the design principles and evaluation criteria. It also presents the final design principles.
- **Chapter 7 – Discussion:** Discusses the final design principles with the CBSE literature, then discusses the enterprise software platform ecosystem they emerged in, bootstrapping and internal reuse, reflections on team management, and limitations.
- **Chapter 8 – Conclusion and Future Work:** Concludes the conducted design science research project and offers suggestions for future work.

Chapter 2: Background

This chapter provides background related to the research problem and aim of the thesis. The problem is that software components created by implementation groups to aid them during localization are not reused. Therefore, the practical aim is to explore, develop, and evaluate a component management system within the DHIS2 ecosystem to address this problem. For this purpose, this chapter provides history on the HISP project and the DHIS2 platform, the DHIS2 platform's current situation concerning tackling localization issues, and the role of the DHIS2 Design Lab.

2.1 HISP and DHIS2

The Health Information Systems Programme (HISP) is a "sustainable and scalable research project supporting health information systems" coordinated by the University of Oslo (Adu-Gyamfi et al., 2019, p. 1). Its software, DHIS2, is implemented in over 100 developing countries and estimated to cover 2.28 billion people (Adu-Gyamfi et al., 2019). Health Information Systems (HIS) is a specific type of Information System (IS) that is designed to handle different kinds of health-related data and is a foundational part of every health system (Adam & de Savigny, 2009). The DHIS2 software is a "generic open-source software system with data warehousing functionalities and customizable modules for integrated health data management" (Adu-Gyamfi et al., 2019, p. 73). Adu-Gyamfi et al. (2019) explain that the DHIS software was created over two decades ago to support the reconstruction of the health sector in provinces in post-apartheid South Africa in 1995, where the goal was to develop a district-based health information system.

Further, Adu-Gyamfi et al. (2019) explain that after success during pilot testing in districts, it grew to become adopted as the national standard in all districts in the country by the Department of Health in South Africa. Throughout this process, the project had invested significant resources in building capacity in HIS where DHIS2 was implemented, from formal training part of a master's course, which cascaded down to health workers in the provinces by trainees (Adu-Gyamfi et al., 2019). Based on the described approach's continued success in other countries, a growing ecosystem of actors have formed around DHIS2, such as HISP groups, universities, Ministries of Health, non-governmental organizations (NGOs), researchers, students, global donors, and more (Adu-Gyamfi et al., 2019). Twelve formal implementation groups specialize in implementing the DHIS2 platform (Li & Nielsen, 2019). While health is the DHIS2 platform's primary goal, it is also starting to see use within agriculture, education, e-government, and logistics management (Adu-Gyamfi et al., 2019).

The original DHIS2 software faced challenges in its first version due to its architecture not being suited for distributed development (Adu-Gyamfi et al., 2019). In addition, Adu-Gyamfi et al. (2019) explain that the standalone installation of DHIS2 at each health facility required a large maintenance team to travel around to keep each implementation up to date and functional. Because of this, the development of the second version of DHIS (DHIS2) started, which allowed for distributed software development and centralized maintenance (Adu-Gyamfi et al., 2019). Following such a platform architecture strategy have turned DHIS2 from a simple software solution into a digital platform (Msiska et al., 2019) that follows Tiwana's (2013, p. 5) definition of a software platform as a "software-based product or service that serves as a foundation on which outside parties can build complementary products or services."

Adu-Gyamfi et al. (2019) explain that one tension for the continued growth of the DHIS2 platform is the balancing between developing globally relevant generic software and serving the particular needs of one user. Li & Nielsen (2019, p. 2) report end-user challenges in DHIS2 as "unfamiliar terminologies and mismatches between work practices, conceptual understandings, and the software user interfaces." Further, Li & Nielsen (2019) explain that many of these challenges are impossible to solve during generic design due to multiple local variations. Additionally, solving them during implementation is also challenging due to "limitations in localization capabilities of the DHIS2 software, limited awareness and competence around usability-design, lack of appropriate design methods, time and resource constraints, and potential maintenance implications of customizing the software extensively" (Li & Nielsen, 2019, p. 2). Such challenges with localization are also significant in other enterprise software platforms. For example, Wareham et al. (2013) explain that large enterprise software platform providers such as SAP and Oracle face similar challenges in designing for end-use. Therefore, they also use local implementation groups with far greater competence in their local markets.

2.2 Generic software development

The DHIS2 Core Team is responsible for developing the platform and its generic resources. They provide a dashboard and a set of "bundled apps" that allow for generic functionality such as data entry for end-users. To address the challenges with localization in DHIS2, HISP groups leverage resources presented by the DHIS2 Core Team, known as boundary resources (Msiska et al., 2019). In total, Msiska et al. (2019) summarize four types of boundary resources within the DHIS2 ecosystem. The first is DHIS2 Academies or Workshops meant for capacity building through interactions during events. The second is the DHIS2 manuals released with each version of DHIS2: an end-user manual, an implementer manual, and a developer manual, used for capacity building through documentation. The third is the facilitators themselves during the DHIS2 Academies and Workshops, which are the human actors engaged in capacity building. The fourth are the DHIS2 APIs, Libraries, and SDKs which are artifacts meant to facilitate app development through software development. All these resources are used in tandem to build capacity in the ecosystem and to make the DHIS2 platform more configurable to a local context.

React is a state-of-the-art web development framework based on the concept of components. Components are independent and reusable pieces of code that can be implemented based on their interfaces (Sommerville, 2016). Web applications built with React are, in theory, just one large component, composed of several other components. Therefore, one builds individual components that handle one aspect of the business logic and piece them together to form the whole application. Additionally, these components can be reused multiple times in the same application or across multiple applications. One can easily find components others have made and implement them directly in the application's code. This inherent focus on components has made Node.js one of the largest software ecosystems, hosting more than 230,000 packages (Wittern et al., 2016). The DHIS2 Core Team has adopted React as their primary web development framework, causing many HISP groups to follow. Developing in such a web framework brings the inherent focus on components and reuse with it and is a foundation for the component management system to succeed in this context.

2.3 DHIS2 Design Lab

The DHIS2 Design Lab is a project at the University of Oslo established to strengthen the usability and local relevance of DHIS2 (Li, 2019b). Its members consist primarily of master's students who write their theses through the lab. In addition, some of them act as coordinators in scheduling seminars, workshops, and other sources of knowledge transfer. A Ph.D. candidate heads the DHIS2 Design Lab. An essential element of the DHIS2 Design Lab is transferring knowledge from last-year master's students to first-year master's students. The knowledge transfer helps make sure one need not relearn experiences on, i.e., methodology and practices within the DHIS2 ecosystem. This thesis is conducted as part of the DHIS2 Design Lab, where most practical elements of the project were conducted in collaboration with two other master's students from the lab.

This chapter has introduced the HISP project and history on how the DHIS2 platform went from being a simple software solution requiring standalone installation to following a platform architecture strategy. Additionally, balancing generic and local design has been presented as an ongoing tension in the DHIS2 ecosystem, and the importance of local design in making generic software usable for end-users is specified. HISP groups are specialized implementation groups that use app development as one approach to performing this localization. Last, the DHIS2 Design Lab has been described as an essential entity in guiding and assisting this thesis.

Chapter 3: Related Research

This chapter addresses research related to the research problem and aim of the thesis. The problem is that software components created by implementation groups to aid them during localization are not reused. Therefore, the practical aim is to explore, develop, and evaluate a component management system within the DHIS2 ecosystem to address this problem.

The chapter first discusses the literature on digital platforms, focusing on enterprise software platforms and implementation groups' approaches to localization by developing third-party apps. This literature positions the problem within existing streams of research, motivates the existence of a solution, and helps position the component management system within the DHIS2 ecosystem. Second, the literature on component-based software engineering (CBSE) adds a relevant vocabulary to discuss the component management system and add justificatory knowledge used during its construction. Furthermore, it adds context to analyze implementation groups' existing software component reuse processes and potential barriers to successful reuse. Finally, this chapter presents a gap in existing literature, which provides no prescriptive knowledge on how to enable the reuse of software components created by implementation groups. In summary, literature on platforms, localization, and boundary resources help situate and motivate the problem, while the literature on CBSE brings essential concepts to inform artifact construction.

3.1 Digital platforms

This section discusses the literature on digital platforms. It first provides an overview of the three primary types of digital platforms to illustrate their differences and similarities. It secondly describes the ecosystems that form around software platforms and emphasizes how multiple actors are involved in the co-value-creation process. Third, this chapter discusses the enterprise software platforms used in business-to-business markets before it motivates how supporting localization makes these platforms locally relevant and useful. This chapter finally presents literature on boundary resources and how they support the localization process before summarizing the section.

In recent years, digital platforms have been high on the agenda in the IS field (Constantinides et al., 2018). Platforms are used to explain how both relatively short-lived and new platform businesses like Amazon, Facebook, and Google, have become some of the world's most valued companies and how older companies like SAP have made significant investments to go through the so-called process of platformization (Constantinides et al., 2018). Constantinides et al. (2018, p. 1) further define digital platforms as "a set of digital resources—including services and content—that enable value-creating interactions between external producers and consumers." Roughly three different digital platforms have been identified through classification by Gawer (2014) and Evans & Gawer (2016): transaction platforms, innovation platforms, and integration platforms. Koskinen et al. (2019) elaborate on these types and their differences:

- *Transaction platforms.* Often referred to as multi-sided markets or exchange platforms. Their primary purpose is to facilitate transactions between organizations, entities, and individuals. Benefits of such an approach include reduced transaction costs by allowing different actors to find each other and reduce friction in the transaction process. In addition, these platforms benefit highly from network effects to create a bigger market for transactions. Some examples include Airbnb, which connects individuals who wish to rent with individuals who have available rooms, and Uber, which connects drivers and passengers.

- *Innovation platforms.* Formed of technological building blocks, they provide a basis for developing services and products. In addition, these platforms provide implementation groups with tools and resources to build new solutions on top of the platform. As innovation platforms focus on the technical architecture that enables innovation, they are often called "software platforms" (Tiwana, 2013). Tiwana (2013, p. 5) defines these software platforms as a "software-based product or service that serves as a foundation on which outside parties can build complementary products or services''. The term "software platforms" will be used for the rest of this thesis. Some examples include Android and DHIS2, which allow third-party application development on the platforms.
- *Integration platforms.* Integration platforms combine aspects from the two principal platform types, transaction platforms and innovation platforms (Evans & Gawer, 2016). Evans & Gawer (2016) explain that this category includes Apple, which has matching platforms such as the App Store and an ecosystem for innovation.

To summarize, these different types of platforms support different purposes. While the transaction platform's primary role is to facilitate transactions between organizations, entities, and individuals, a software platform's primary role is to form the technological building blocks from which new products or services can be built. Integration platforms combine aspects from the other two principal platform types. The software platform has been extensively researched based on its popularity. Tiwana (2013, p. 5) builds on Baldwin & Woodard (2008) and defines a software platform as "an extensible software-based system that provides the core functionality shared by "apps" that interoperate with it, and the interfaces through which they interoperate." Gawer (2009) elaborates the perspective from Baldwin & Woodard (2008) and builds upon Tushman & Murmann (1998) to argue that the fundamental architecture behind all platforms essentially is the same: A set of 'core' components with low variety and a complementary set of 'peripheral' components with greater variety. Gawer (2009) states that the low-variety components are what constitutes the platform. 'Peripheral' components, or apps, are defined by Tiwana (2013, p. 6) as a "software service that connects to the platform to extend its functionality." Tiwana (2013) explains that apps work as complementary goods for the platform and that a platform's functionality is enhanced through many apps.

This section brings along the concept of software platforms as a "software-based product or service that serves as a foundation on which outside parties can build complementary products or services'' (Tiwana, 2013, p. 5). Software platforms consist of a platform core and interfaces that allow for complementary products and services by third parties to extend the core functionality, referred to as apps. The following section discusses the ecosystem surrounding a software platform and its actors to understand how and why third-party apps are built on top of software platforms. Additionally, this section brings along the concept of transaction platforms, which facilitate transactions between organizations, entities, and individuals by reducing friction in the transaction process.

3.1.1 Software platform ecosystems

Entire ecosystems emerge around software platforms. Software platforms ecosystems follow Jacobides et al. (2018) definition of a platform ecosystem that emphasizes interactions between consumers, producers, and third-party actors. Tiwana (2013) argues that a platform-based ecosystem consists of two major elements: the platform core and apps. To support app developers in creating apps, the vendor must provide resources, referred to as boundary resources (Ghazawneh & Henfridsson, 2013). They are software tools and regulations, such as software development kits (SDKs), application programming interfaces (APIs), and other tools which facilitate an arms' length relationship between the vendor and external actors (Ghazawneh & Henfridsson, 2013). Section "3.1.3 Boundary Resources" further discusses the concept of boundary resources. Conclusively, a software platform ecosystem consists of the platform core, its boundary resources, third-party apps, and the interactions between the vendor, third-party actors, and consumers.

Software platform ecosystems must balance coordination and autonomy for the ecosystem to flourish (Iansiti & Levien, 2004). Coordination is vital as the platform, and its third-party apps within the ecosystem must integrate into one coherent solution for consumers. On the other hand, autonomy is vital for continuous innovation within the ecosystem and the feeling that third-party actors own what they create. Following Tiwana (2013), this is reflected in the architecture of the software platform ecosystem. Tiwana (2013) explains that the most significant potential strength, or weakness, is the software platform ecosystem's diversity and ability to address unknown challenges. Moreover, architectural choices by the vendor that limit the autonomy of third-party actors may kill innovation, while architectural choices that fail to leverage this diversity into a cohesive platform end up in chaos (Tiwana, 2013).

To summarize, software platform ecosystems emerge around software platforms and consist of the platform core, boundary resources, third-party apps, and the interactions between the vendor, third-party actors, and consumers. This section brings the concept of software platform ecosystems and their containing elements. This thesis focuses on software platforms, specifically enterprise software platforms, and the ecosystems that surround them.

3.1.2 Enterprise software platforms

The software platform literature discusses two broad types of platforms: consumer software platforms and enterprise software platforms. Consumer software platforms such as iOS and Android focus on consumer markets, while enterprise software platforms such as DHIS2 and SAP focus on business-to-business markets. Enterprise software platforms are a particular type of software platform, referred to as generic software (Li & Nielsen, 2019), enterprise software (Foerderer et al., 2019; Rickmann et al., 2014), or enterprise software platform (Foerderer et al., 2019). Companies are increasingly adopting digital platform strategies in business-to-business markets (Kauschinger et al., 2021). Kauschinger et al. (2021) explain that traditional enterprise resource planning (ERP) software vendors such as Oracle and SAP have had to transition from on-premises systems to cloud-based solutions within the enterprise software market, based on competition from companies such as Salesforce and ServiceNow, which has followed a platform business model since their inception. These enterprise software platforms implement app stores like those known from mobile operating systems such as the App Store (Kauschinger et al., 2021). These marketplaces act as an intermediary between implementation groups and users.

This transition from on-premises systems to enterprise software platforms has turned vendors into curators of ecosystems where businesses collaboratively develop and commercialize a shared technology (Foerderer et al., 2019). Foerderer et al. (2019) explain that a substantial risk of platform strategies arises from moving product development from within the firm to implementation groups. If the ecosystem is curated correctly, this can bring significant benefits, while if done incorrectly can severely damage the firm's competitive standing. As such, boundary resources play a significant role for implementation groups to develop and innovate on top of the platform. Therefore, the vendor faces the challenge of furnishing the implementation groups with the required knowledge and resources to develop, integrate and maintain third-party apps (Foerderer et al., 2019). The firm responsible for the platform is referred to as the vendor (Foerderer et al., 2019; Wareham et al., 2013), platform owner (Tiwana, 2013), sometimes also called platform sponsor (Rickmann et al., 2014), or the ecosystem's keystone firm (Iansiti & Levien, 2004). This thesis uses the word vendor and defines it as the firm responsible for the platform's development and ecosystem's orchestration. The developers of complementary products or services are referred to as app developers (Tiwana, 2013), complementors (Eisenmann et al., 2008; Rickmann et al., 2014), third-party complementors (Wareham et al., 2013), partners (Foerderer et al., 2019; Wareham et al., 2013) or implementation groups (Li & Nielsen, 2019). This thesis uses the word implementation group to describe an actor involved in extending the functionality of an enterprise software platform through app development.

The vendor also needs to orchestrate the software ecosystem. While curating is the sum of actions taken to grow and nurture the ecosystem, orchestration means aligning its processes with those of the implementation groups in providing a constant stream of offerings for end-users (Iansiti & Levien, 2004; Rickmann et al., 2014). Such orchestration is required based on the number of implementation groups and demands a high level of standardization and automation in managing the co-value-creation process (Rickmann et al., 2014). One sector that has successfully developed a vibrant ecosystem of implementation groups is ERP software (Wareham et al., 2013). Wareham et al. (2013, p. 1196) explain that providers such as SAP and Oracle have benefited dramatically from local or regional implementation groups that make "country-specific modifications, sector specific add-ons, and company-specific [customizations] that meet the local, distinct needs of their clients." Wareham et al. (2013) state that the reason to use such implementation groups is the extreme heterogeneity that characterizes this market. Therefore, using one standard software suite to meet the needs of multiple sectors is a considerable challenge, especially considering national differences. As such, these ERP vendors consign localization tasks to "regional [implementation groups] with far greater expertise in their native markets" (Wareham et al., 2013, p. 1196). Continuing, they explain that the very high cost of localization is generally the highest single cost in the ecosystem collectively, as stated by a software vendor: "[Localization] affects three main areas: translation, legal requirements, and local business practices," "It's not easy to manage the [localizations] at the global level, continent level, country level," "[Localization] costs remain one of our single largest concerns" (Wareham et al., 2013, p. 1207). Even though localization is a considerable challenge, the sharing of resources such as country-level modifications between distinct implementation groups is not always the case as there is no incentive to do so (Wareham et al., 2013). One reason for this is their direct competition, where each implementation group claims that their application is better than their competitors.

Li & Nielsen (2019) conceptualizes the usability-related design of the generic enterprise software platform DHIS2 as concerning two levels and types of design. First, generic-level design concerns the development of generic software across different organizational settings to be used directly. Second, the generic-level design also involves designing resources to be used during software implementation, referred to as design for implementation-level design. These features and resources are to be used by implementation groups to shape the software towards local needs and represent components of a design infrastructure (Li & Nielsen, 2019). As such, implementation groups leverage the capabilities of the design infrastructure during localization.

One can view design infrastructures as open, shared, and heterogeneous information infrastructure that evolves within an ecosystem (Li & Nielsen, 2019). Design infrastructures consist of three resources: the adaptable software, supporting artefacts, and soft resources (Li & Nielsen, 2019). Li & Nielsen (2019) describe adaptable software as entailing three types of localization capabilities: configuration, customization, and extension. Configuration represents the design of functionality that can be switched on or off as part of the software. Customization represents changes where those offered by configuration options are not sufficient, often done by modifying the source code. One can describe extension as a form of customization. However, Li & Nielsen (2019) argue that extension through apps within a platform context is significantly different from changing the software's source code. These apps are created based on stable APIs and then integrated, resulting in far fewer problems during updates of the source code. Li & Nielsen (2019) argue that extension within a platform context seems ideal for localization support as it offers implementation-level designers far more design flexibility than configuration. However, Li & Nielsen (2019) share the same view as Wareham et al. (2013) in that creating such apps during localization is resource-intensive.

Further, Li & Nielsen (2019) describe supporting artefacts as representing all material resources that aid implementation-level designers during the localization of the adaptable software, such as functional component libraries and SDKs. Soft resources provide competence and methodical support during localization, such as online tutorials and forums where sharing of competence and best practices are prevalent. The vendor is often the lead provider of these supporting artefacts and soft resources while still eliciting contributions from the community. Within this, two leading roles of generic-level designers are presented by Li & Nielsen (2019). The first is developing resources that facilitate desired processes, such as adaption features in the software, supporting artefacts, or documentation. The second is to create mechanisms that support and enable implementation-level designers to extend the design infrastructure. For example, to enable component libraries to be reused by other implementers.

To summarize, the transition to enterprise software platforms has transitioned vendors of enterprise software into curators of ecosystems, where businesses collaboratively develop and commercialize a shared technology (Foerderer et al., 2019). Localization tasks, adapting the software to the local context, are consigned to “regional [implementation groups] with far greater expertise in their native markets” (Wareham et al., 2013, p. 1196). One approach to localization providing a high level of design flexibility is creating third-party apps (Li & Nielsen, 2019). This approach seems ideal but is resource-intensive (Li & Nielsen, 2019). To support their app development processes, implementation groups use resources in the design infrastructure, such as supporting artefacts. This section has introduced an enterprise software platform to refer to a software platform used in a business-to-business market. Additionally, the term localization refers to adapting the enterprise software platform to the local context. The design infrastructure is conceptualized as the collection of resources that aid implementation-level designers in designing for end-use.

3.1.3 Boundary resources

Discussions on boundary resources are prominent within the broader platform literature (Ghazawneh & Henfridsson, 2013; Tiwana, 2013). When discussed in comparison to design infrastructures presented by Li & Nielsen (2019), there is an overlap between supporting artefacts and soft resources, and boundary resources. This overlap is further elaborated after discussing the role of boundary resources.

Ghazawneh & Henfridsson (2013) argue that to successfully build platform ecosystems, the vendor's focus must be to provide resources that support implementation groups in their work. These resources are referred to as platform boundary resources. Boundary resources are software tools and regulations, such as SDKs, APIs, and other tools that facilitate an arms' length relationship between vendors and external actors (Ghazawneh & Henfridsson, 2013). These are the interfaces with which third-party apps interact. Continuing, Ghazawneh & Henfridsson (2013) explain that these boundary resources have a two-fold role. First, they are imperative to 'transfer design capability to users' (von Hippel & Katz, 2002, p. 824). Secondly, they are designed to control the platform and the ecosystem that evolves around it (Ghazawneh & Henfridsson, 2010).

Ghazawneh & Henfridsson (2013) define the two processes, resourcing and securing, as two drivers of boundary resource design. Resourcing is where the scope and diversity of a platform are enhanced, while securing is the process where the platform's control is increased. Both resourcing and securing are vital parts of curating an ecosystem, as the capabilities of the boundary resources often decide what implementation groups can do. In general, there are two aspects of resourcing: resourcing and self-resourcing (Ghazawneh & Henfridsson, 2013). While resourcing is the act of the vendor creating additional resources that transfer design capability to the ecosystem, self-resourcing is the act of the implementation groups creating new resources based on a perceived limitation of those from the vendor, or when they fall short of what is necessary.

Interactions through the arms' length relationship are, following Wenger-Trayner (2000), facilitated by three bridges: boundary objects, boundary interactions, and brokers.

- *Boundary objects.* Artifacts, such as tools, documents, or models that are shared by communities across a boundary.
- *Boundary interactions.* Encounters such as visits or discussions that provide direct exposure to members of another community.
- *Brokers.* Individuals who act as brokers between communities.

This distinction of boundary resources is also reflected within enterprise software platforms (Foerderer et al., 2019). Msiska et al. (2019) expand on this notion by enabling the demarcation between capacity building boundary resources and software development boundary resources. Msiska et al. (2019) explain that software development boundary resources are boundary objects that correspond to those introduced earlier by Ghazawneh & Henfridsson (2013). On the other hand, capacity building boundary resources comprise boundary objects, boundary interactions, and brokers deployed within a software ecosystem to facilitate the propagation of generative capacity between vendors and users. Generative capacity here refers to an individual's ability to produce something ingenious or new in a particular context (Avital & Te'eni, 2009; Msiska et al., 2019). Msiska et al. (2019, p. 8) continue that the implicit shift of capacity from vendors to implementation groups "necessitates the existence of both software development boundary resources and capacity building boundary resources."

Supporting artefacts and soft resources discussed in the previous chapter can collectively be compared to software development boundary resources, although with some differences. Supporting artefacts and soft resources are part of a design infrastructure used during implementation-level design consisting of several options for local design. Software development boundary resources are used during app development, which is one approach to localization. In that sense, software development boundary resources are a subset of the resources made available through a design infrastructure to strengthen app development. Thus, software development boundary resources are a subset of supporting artefacts and soft resources that aid implementation groups in extending the software platform's capabilities during localization.

To summarize, boundary resources are essential to successfully build platform ecosystems (Ghazawneh & Henfridsson, 2013) and enterprise software platform ecosystems (Foerderer et al., 2019). For the problem being explored in this thesis, the most critical aspect of boundary resources is software development boundary resources and how they transfer design capabilities to implementation groups. This section brings the concept of software development boundary resources by Msiska et al. (2019), which in this thesis when viewed in tandem with the design infrastructure, is defined as the subset of supporting artefacts and soft that during localization transfer design capabilities to implementation groups in extending the capabilities of an enterprise software platform.

This subchapter has discussed research on digital platforms related to the problem that software components created by implementation groups to aid them during localization are not reused. By reviewing the literature on digital platforms, the problem has been positioned within enterprise software platforms and narrowly positioned within the localization of enterprise software platforms through app development. This subchapter brings the concepts described in Table 1 below.

Concept	Definition	Type
Enterprise software platform	A software platform in a business-to-business market (Foerderer et al., 2019).	Entity
Enterprise software platform ecosystem	The ecosystem that emerges around an enterprise software platform. It consists of the platform core, boundary resources, apps, and the interactions between the vendor, implementation groups, and consumers (Foerderer et al., 2019; Jacobides et al., 2018; Tiwana, 2013; Wareham et al., 2013).	Collection of entities, actors, and processes
Localization	The process of adapting the enterprise software platform to a local context (Li & Nielsen, 2019).	Process
Software development boundary resources	The subset of the design infrastructure that during localization transfer design capabilities to implementation groups in extending the capabilities of an enterprise software platform (Li & Nielsen, 2019; Msiska et al., 2019).	Entity
Vendor	The firm responsible for the platform's development and ecosystem's orchestration (Foerderer et al., 2019; Rickmann et al., 2014).	Actor

Implementation group	An actor involved in localizing an enterprise software platform (Li & Nielsen, 2019).	Actor

Table 1. Concept definitions from enterprise software platforms.

3.2 Component-based software engineering

During localization of DHIS2, it has been documented that implementation groups create self-resourced software development boundary resources to aid them in extending the software through third-party apps (Li & Nielsen, 2019; Msiska et al., 2019). One such resource is component libraries (Li & Nielsen, 2019), a collection of individual components used to build third-party apps. This subchapter discusses the literature on CBSE concerning the problem that software components created by implementation groups to aid them during localization are not reused.

The subchapter first discusses an overview of CBSE literature, focusing on the benefits and barriers to successful component reuse. This overview gives context as to why some implementation groups rely on component-centric development practices. This subchapter secondly describes the two main processes within CBSE: development for reuse and development with reuse. These processes provide context on the implementation group's reuse practices. Finally, this subchapter draws on the discussed literature to conceptualize a component management system to aid implementation groups in reusing components. By doing so, CBSE adds the required vocabulary to discuss the component management system and add justificatory knowledge used during its construction.

Components are the basic unit of reusability, the building blocks of component-based software, and the core element of CBSE itself (Kumar & Tiwari, 2020). Szyperski (2002, p. 41) defines a component as a "unit of composition with contractually specified interfaces and context dependencies only" that can be deployed separately and is subject to composition by others. Sommerville (2016) describes components as abstractions defined by their interfaces, where all implementation details are hidden from other components. Further, Sommerville (2016) explains that the underlying design principles refer to components as independent entities that do not interfere with each other's operation and communicate through well-defined interfaces. Thus, a component is a black box responsible for some implemented functionality, and multiple components are composed together to form apps.

Sommerville (2016) explains that CBSE involves two types of components: software development using embedded components and software development using external services. During development with embedded components, components are directly integrated into the system using a copy of that component. During service-oriented development, the external service is referenced without including a copy of that service in the system. This thesis focuses on embedded components, rather than components as external services, as these embedded components are packaged together and provided in supporting artefacts as component libraries. Therefore, this thesis refers to these embedded components as "software components."

CBSE "is the process of defining, implementing, and integrating or composing" components into systems (Sommerville, 2016, p. 465) and is a proven way of efficiently reusing software (Kumar & Tiwari, 2020). There are three fundamental properties in CBSE (Heineman & Councill, 2001): the development of systems using existing software components, the capability to reuse these existing software components and newly developed components in other apps, and the preservation, fabrication, and maintenance of these components. Kumar & Tiwari (2020) list some of the primary characteristics of CBSE as the following:

- *Reusability.* Software reusability is defined as creating software from existing software rather than building it from scratch (Krueger, 1992) and is the focal property of CBSE.
- *Composability.* Composability refers to apps being composed of components, which subsequentially are composed of other components.
- *Shorter development cycle.* A shorter development cycle is in CBSE gained by following the divide, solve and conquer approach, where large apps are divided into manageable modules, being the components.
- *Maintainability.* One gains better maintainability by the independence of the software components, where composable components are easier to maintain than monolithic software.
- *Improved quality and reliability.* One gains improved quality and reliability by using pre-tested and qualified components.
- *Flexibility and extendibility.* One gains flexibility and extendibility by allowing developers to compose apps from components that are easy to add, update, modify, and remove without modifying other components.

Even though CBSE promises faster time-to-market and reduced costs, there are also barriers to potentially successful reuse (Kumar & Tiwari, 2020). For example, B et al. (2010) present some barriers as:

- *Lack of components.* Many domain areas might lack useful components. While stable domain areas can have many well-tested components, newer domain areas still need to adapt to rapidly changing requirements.
- *Poor cataloging, distribution, and access methods.* The burden of finding reusable components is often up to the individual user(s). Additionally, Kumar & Tiwari (2020) describes the maintenance of mechanisms for component cataloging as a major issue within CBSE.
- *Component certification.* Certification methods that guarantee well-tested and certified components can make users comfortable reusing them. However, Kumar & Tiwari (2020) agrees and describes the lack of specifically established certification procedures for the validity and usability of components as one of the major issues within CBSE.
- *Incentive and management support.* Adopting strong reuse programs can prove beneficial in the long term but might influence short-term competitiveness. Therefore, the focus is often on applying the reuse process after finished development instead of integrating it during active development.

Following Kumar & Tiwari's (2020, p.23) brief description of the evolution of CBSE, the idea of components was first introduced in 1968 by Douglas McIlroy at a NATO conference on software engineering. Kumar & Tiwari (2020) further explain that in the 1980s and 1990s, a “component” acquired the identity of a building block and architectural unit. Now, 40 years later, CBSE has become a standard software engineering paradigm and one of the most promising for developing large, complex, and high-quality systems (Capretz et al., 2001; Kumar & Tiwari, 2020). Its importance has slowly become apparent as software systems become larger and more complex. Moreover, with customers demanding dependable software delivered quickly, the focus has shifted from reimplementing software components from scratch to reusing them. As such, CBSE is an effective way to develop custom software systems more effectively in a reuse-oriented way (Sommerville, 2016).

To summarize, this thesis combines Szyperski's (2002) and Sommerville's (2016) views. It defines a component as a unit of composition that is defined by its interfaces, which can be deployed separately and is subject to composition. CBSE as a development paradigm is described as one of the most promising for the development of large, complex, and high-quality systems. However, there are also barriers to successful reuse. Most notably is the lack of components in domain areas, poor cataloging, distribution, access methods, and component certification methods that guarantee well-tested and certified components.

3.2.1 CBSE Processes

To analyze implementation groups' component reuse practices, one must understand the processes of CBSE and their actors. CBSE at the highest level consists of two separate processes, CBSE for reuse and CBSE with reuse (Kotonya et al., 2003; Kumar & Tiwari, 2020). Kotonya et al. (2003) describe development for reuse as concerned with analyzing app domains and developing domain-related components. During this process, the primary objective is to produce one or more reusable components and making them available through a component management system (Sommerville, 2016). Kotonya et al. (2003) describe development with reuse as concerned with assembling software systems from existing components. Sommerville (2016) adds that one might not know what components are available during this process, so these components need to be discovered and the system designed to make use of them most effectively. Kotonya et al. (2003) further explain that similarities in related systems must be discovered and represented in a form that can be exploited for successful software reuse. Moreover, Kotonya et al. (2003, p. 2-3) explain that for organizations to achieve the benefits of component reuse, a successful strategy must include the following:

1. "A reuse method that is consistently applied by component and application developers."
2. "The establishment of component review mechanisms whose function is to identify common components by identifying commonalities of related systems."
3. "Establishment of component design reviews of all ongoing projects."
4. "Establishment of a repository for maintaining the information about available reusable components."
5. "Establishment of effective component 'harvesting' mechanisms."
6. "Adopting effective component management methodologies."

One aspect of CBSE is the necessary mediation between component producers and component users (Szyperski, 2002). Component producers are involved in developing, providing, and distributing components during the development for reuse process. In contrast, component users search for and identify relevant components before leveraging these during the design and development of software during the development with reuse process. These are the two user groups that interact with the component management system by producing new components and the extraction and use of those components.

Within the two overarching processes exist other supporting processes. Sommerville (2016) describes them as component acquisition, component management, component certification, and component identification.

- *Component acquisition.* The process of acquiring components for reuse (Sommerville, 2016). Sommerville (2016) describes this process as acquiring external components to reuse or develop components from locally accessible code. Developing new components can include removing app-specific methods, changing names to make the components more general, or adding methods to provide complete functional coverage.
- *Component management.* The process of managing a company's reusable components, ensuring that they are correctly cataloged and made available for further reuse in a component management system (Sommerville, 2016). The process includes deciding how to classify components to be discovered and then including the components and their information in a component repository.
- *Component certification.* The process of certifying if a component meets its specification and if it has reached an acceptable quality standard before it is made available for reuse (Sommerville, 2016). The process of component certification is described as necessary by Sommerville (2016, p. 477), where an entity "apart from the developer checks the quality of the component" before it is made available for reuse. However, Sommerville (2016) describes this process as expensive; therefore, it is often left to the component producer to certify their components. Multiple researchers discuss leaving the certification up to third-party certification authorities, also called certifiers (Kotonya et al., 2003; Sommerville, 2016; Szyperski, 2002). Heineman & Councill (2001) additionally discusses third-party certification as a requirement for trusted components. Kumar & Tiwari (2020) illuminate qualitative models with pre-defined assumptions and guidelines to address issues with quality and certification.
- *Component identification.* The process of identifying suitable and useful components to reuse in apps (Sommerville, 2016). Sommerville explains that to identify components, component users first search for components before selecting the appropriate, on which they validate correct functionality. Sommerville (2016) further explains that since there are few component vendors, component users first look at the company's reusable components in the component repository. They alternatively look in the open-source world, such as GitHub and Sourceforge, to see if the source code for the component needed is available. After searching, the components most fitting the requirement are selected before validated to see if they behave as advertised.

This section has discussed the primary processes within CBSE as development for reuse and development with reuse, conducted by component producers and component users. Additionally, this section discussed component acquisition, component management, component certification, and component identification as underlying processes within CBSE. Lastly, this section introduces a component repository for storing information on reusable components as a critical element of component reuse.

3.2.2 Central concepts in CBSE

Component repositories are a place to store components for future reuse after acquisition (Kumar & Tiwari, 2020). Component repositories are well known within the CBSE paradigm to catalog, manage, and maintain an organization's components (Kotonya et al., 2003; Kumar & Tiwari, 2020; Sommerville, 2016; Szyperski, 2002). Kumar & Tiwari (2020) describe it as a component database that contains the component's source code, metadata, interfaces, and additional information. Kumar & Tiwari (2020) further explain that these repositories need to be managed regularly to ensure their consistent efficiency. Component repositories are accessed both during development for reuse, relating to the insertion of new components, and development with reuse, relating to reusing components in apps (Sommerville, 2016). Component repositories form the foundation by which other processes rely upon, therefore having a well-managed component repository makes it easier to follow CBSE practices. Maintenance of component repositories is a major issue within CBSE (Kumar & Tiwari, 2020). One needs to identify both useful and outdated components, update the list of components, update their metadata, and maintain different versions of each component without unnecessary duplication and ambiguity. Nonetheless, during the development for reuse process, components are made available in a component repository through a component management system.

One can package software components in many ways, such as function libraries and frameworks (Kotonya et al., 2003). These function libraries are comparable to those discussed by Li & Nielsen (2019) in the previous section, "3.1.2 Enterprise Software Platforms". Sommerville (2016) further mentions how components should be packaged for deployment as independent, executable routines. Additionally, Szyperski (2002) explains that within the development language “Java,” individual components are packaged into a single file for distribution consisting of the source code, dependencies, and meta-information. This packaging allows components to be distributed in a way that new instances of the component are effectively copied from the original. Furthermore, such packages are used to generalize the collection of components and provide the overarching information of where the components should be used.

This subchapter has discussed research on CBSE related to the problem that software components created by implementation groups to aid them during localization are not reused. By reviewing the literature on CBSE, this subchapter presents benefits such as shorter development cycles and better maintainability. In addition, this subchapter describes known barriers to successful reuse as the lack of components, poor cataloging, distribution, access methods, and poor certification methods. The process of component management is described by Sommerville (2016) as managing a company’s reusable components, ensuring that they are correctly cataloged and made available for further reuse. Additionally, Sommerville (2016) describes that such reusable components are made available through a component management system after their development.

Subscribing to this conceptualization, a component management system is in this thesis used to describe a system through which reusable components are made available for reuse in a component repository. Such a system is based on the foundation of a component repository to catalog, manage, and maintain components, component certification procedures to quality assure components, and mechanisms to support component identification. From this subchapter, this thesis brings the concepts presented in Table 2 below.

Concept	Definition	Type
App	Complimentary service developed by implementation groups during localization (Li & Nielsen, 2019; Tiwana, 2013) and composed of components (Heineman & Councill, 2001; Kumar & Tiwari, 2020).	Entity
Component	A unit of composition that is defined by its interfaces and is subject to composition (Kumar & Tiwari, 2020; Sommerville,	Entity

	2016; Szyperski, 2002). It is developed and reused by implementation groups (Li & Nielsen, 2019).	
Package	A unit of distribution that encapsulates and contains one or more reusable components (Kotonya et al., 2003; Sommerville, 2016). It is created by implementation groups (Li & Nielsen, 2019).	Entity
Component management system	A system through which reusable components are made available for reuse. Based on the foundation of a component repository to catalog, manage, and maintain components, component certification procedures to quality assure components, and mechanisms to support component identification (Sommerville, 2016). It is accessed by component producers and component users.	Entity
Component reuse	The processes of development for reuse and development with reuse, conducted by component producers and component users, respectively (Kotonya et al., 2003; Szyperski, 2002).	Process
Component producer	A member of an implementation group involved in creating and publishing components in the development for reuse process (Szyperski, 2002).	Actor
Component user	A member of an implementation group involved in using components others have made in the development with reuse process (Szyperski, 2002).	Actor

Table 2. Concept definitions from CBSE.

3.3 Summary

This chapter has discussed research on digital platforms related to the problem that software components created by implementation groups to aid them during localization are not reused. By reviewing the literature on digital platforms, the problem has been positioned within enterprise software platforms and narrowly positioned within the localization of enterprise software platforms through app development. The literature on enterprise software platforms, localization, and boundary resources helps situate and motivate the problem. The concepts in Table 3 below are used to position the work within this context.

Concept	Definition	Type
Enterprise software platform	A software platform in a business-to-business market (Foerderer et al., 2019).	Entity
Enterprise software platform ecosystem	The ecosystem that emerges around an enterprise software platform. It consists of the platform core, boundary resources, apps, and the interactions between the vendor, implementation groups, and consumers (Foerderer et al., 2019; Jacobides et al., 2018; Tiwana, 2013; Wareham et al., 2013).	Collection of entities, actors, and processes

Localization	The process of adapting the enterprise software platform to the local context (Li & Nielsen, 2019).	Process
Software development boundary resources	The subset of the design infrastructure that during localization transfer design capabilities to implementation groups in extending the capabilities of an enterprise software platform (Li & Nielsen, 2019; Msiska et al., 2019).	Entity

Table 3. Concepts to position the platform context.

Next, the concepts in Table 4 below refer to the relevant actors within this platform context. Again, the primary focus is on the implementation groups, as they are the actors leveraging the software development boundary resources presented by the vendor to localize the enterprise software platform.

Concept	Definition	Type
Vendor	The firm responsible for the platform's development and ecosystem's orchestration (Foerderer et al., 2019; Rickmann et al., 2014).	Actor
Implementation group	An actor involved in localizing an enterprise software platform (Li & Nielsen, 2019).	Actor

Table 4. Concepts of actors involved in the platform context.

Since many implementation groups face the same challenges during localization, reusing software components can be valuable. However, existing literature provides no prescriptive knowledge on how to accomplish it. For this purpose, CBSE describes potential barriers to implementation groups' successful component reuse, such as poor distribution of components, poor cataloging, and lack of certification metrics (B et al., 2010). Therefore, this chapter has introduced concepts from the CBSE literature to address such barriers and the problem that software components created by implementation groups to aid them during localization are not reused. Table 5 below presents the essential concepts from the CBSE literature used to analyze implementation groups' component reuse processes, provide the vocabulary to discuss the component management system, and inform its creation.

Concept	Definition	Type
App	Complimentary service developed by implementation groups during localization (Li & Nielsen, 2019; Tiwana, 2013) and composed of components (Heineman & Councill, 2001; Kumar & Tiwari, 2020).	Entity
Component	A unit of composition that is defined by its interfaces and is subject to composition (Kumar & Tiwari, 2020; Sommerville, 2016; Szyperski, 2002). It is developed and reused by implementation groups (Li & Nielsen, 2019).	Entity
Package	A unit of distribution that encapsulates and contains one or more reusable components (Kotonya et al., 2003; Sommerville, 2016). It is created by implementation groups (Li & Nielsen, 2019).	Entity

Component management system	A system through which reusable components are made available for reuse. Based on the foundation of a *component repository* to catalog, manage, and maintain components, *component certification* procedures to quality assure components, and mechanisms to support *component identification* (Sommerville, 2016). It is accessed by component producers and component users.	Entity
Component reuse	The processes of development for reuse and development with reuse, conducted by component producers and component users, respectively (Kotonya et al., 2003; Szyperski, 2002).	Process
Component producer	A member of an implementation group involved in creating and publishing components in the development for reuse process (Szyperski, 2002).	Actor
Component user	A member of an implementation group involved in using components others have made in the development with reuse process (Szyperski, 2002).	Actor

Table 5. Concepts from CBSE to address the research problem.

This thesis explores creating a component management system discussed in the CBSE literature to address potential barriers in the component reuse process experienced by implementation groups in an enterprise software platform ecosystem. Addressing these barriers is essential to address the research problem and thus enable software component reuse.

Chapter 4: Research Approach

This chapter describes the research approach used to address the research problem that software components created by implementation groups to aid them during localization are not reused. Therefore, the practical aim is to explore, develop and evaluate a component management system within the DHIS2 ecosystem, while the theoretical aim is to extract design principles from this artifact.

This chapter first presents the overarching methodology which has guided this research project: Design Science Research (DSR). From DSR, an abstract research approach is built using central concepts, a process model, kernel theories for justification knowledge, and design principles as the type of contribution. Additionally, this chapter presents the categories of inquiry for data collection, the methods conducted, the approaches to data analysis, and concludes the abstract research approach. Afterward, this chapter describes the implemented research process. To do so, it uses the abstract research approach and follows the activities from the process model. Here the essential aspects of the process are presented regarding the problem, the objectives of the artifact, artifact construction during design and development, the demonstrations and evaluations before describing how the contribution is communicated back to literature. Last, this chapter describes the team management, ethical considerations, limitations with the applied method, and a summary.

4.1 Methodology

The research question that was asked: "*What are design principles for component management systems within enterprise software platform ecosystems?".* This research question attempts to address the practical problem that software components created by implementation groups to aid them during localization are not reused. This project has followed the Design Science Research methodology, as it through engaged research allows for the generation of prescriptive knowledge on how to solve problems. IS as a field is about solving problems at the intersection between organizations and information technology (IT) (Peffers et al., 2007). DSR is a problem-solving paradigm that continuously shifts its focus between the world as acted upon (processes) and the world as sensed (artifacts) (Hevner et al., 2004). Hevner et al. (2004) explain creating artifacts within DSR as cycles of building and evaluation. The researcher attempts to solve a problem by creating an artifact and subsequent evaluation to further inform artifact construction. Subscribing to such a methodology provides the project with two essential aspects. First is the space for prototyping, designing, and developing an artifact to provide utility in solving a problem. Second is its defined processes of building on prior research to inform artifact construction, and then the subsequent communication of findings back to theory. As such, it allows this project's contribution to be twofold. First, it allows for a practical contribution of an artifact that attempts to solve the practically relevant problem while secondly generalizing and extracting design principles that can be contributed back to theory. These design principles should be prescriptive, meaning they should inform action by providing instructions for other researchers or designers to follow.

The problem is explored through the interpretive paradigm. Throughout the process, the aim has been to understand the perspectives of HISP groups regarding their current and future app development practices, available resources, and challenges met during the app development process. Additionally, the aim has been to understand the perspectives of the DHIS2 Core Team concerning the DHIS2 ecosystem and the artifact's construction and evaluation. For both aims, the focus has been on collecting rich, qualitative data on their thoughts, ideas, and experiences through focus groups, discussions, interviews, evaluations, and informal communication channels. As with interpretive case studies, this thesis's nature should be viewed as findings from one context. Therefore, the data emerging from this context and the resulting design principles can not be thought of as universal truth. Furthermore, the set of design principles that form this thesis' contribution is from the context of one enterprise software platform ecosystem and needs to be applied and tested in multiple ecosystems. Thus, this thesis's goal is not to provide universal truth but to provide utility in addressing a research problem.

As an overarching framework for conducting DSR, this project has followed the Design Science Research Methodology (DSRM) (Peffers et al., 2007). It builds on previous research within the DSR field and provides a coherent perspective on the DSR process and its activities. Subscribing to such a methodology provides a conceptual perspective of the process and its essential steps. It also provides specific actions that can be taken during each of these steps. Moreover, following a well-known methodology also provides the thesis with a greater level of validity. Figure 1 below displays the DSRM process model.

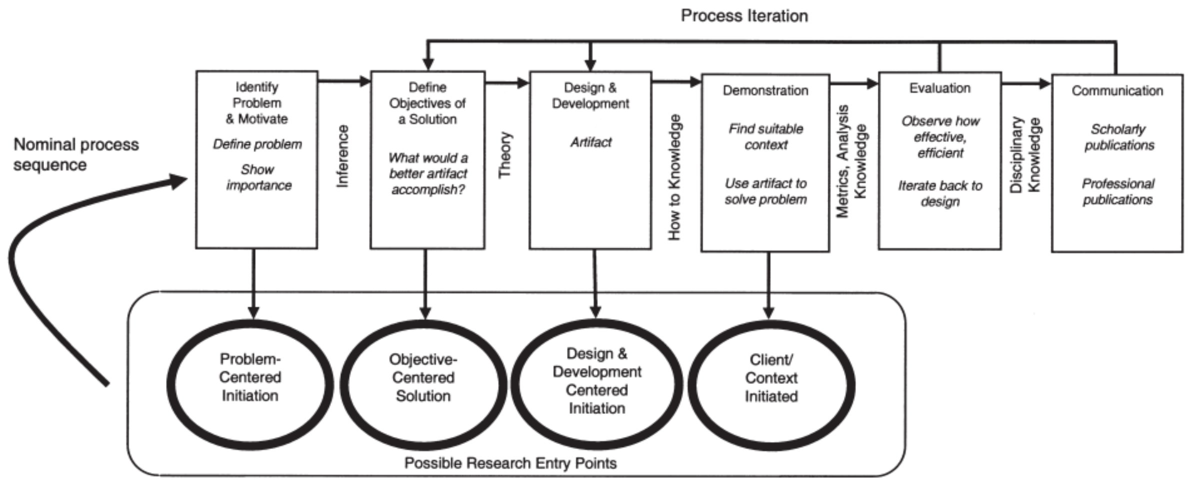

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 1. Design Science Research Methodology (DSRM) Process Model, reprinted from Peffers et al. (2007, p. 54).

The DSRM Process Model consists of six activities. The process can be entered through one of its four first process steps depending on what spurred the research. Then it progresses through the process sequence, going through each of the different activities. Following the explanation by Peffers et al. (2007), the steps are as follows. First *problem identification and motivation*, then *defining the objectives of a solution*, before *designing and developing* the artifact, *demonstrating* the artifact, before *evaluating* its performance. Last, *communicating* the findings to both practitioners and researchers. Even though these steps are sequentially ordered, a researcher can start at any step and move outwards. This research project was initiated in the problem identification and motivation activity. This process model is used later in this chapter to present the implemented research process in "4.4 Implemented Research Process".

4.1.1 Central concepts in DSR

Following March & Smith (1995), IT artifacts can generally be put into four categories. These are constructs, models, methods, and instantiations. First, constructs constitute the vocabulary of a domain and form a language to describe its problems and solutions. Second, models are a set of propositions expressing relationships among constructs. These are abstractions and representations seeking to describe how things are. Third, methods are sets of steps used to perform tasks. Methods are based on underlying constructs for its language and models for its representations to form a guideline. Fourth, instantiations are realizations of artifacts in their environment. Instantiations operationalize the underlying constructs, models, and methods. The practical contribution of this study is that of an instantiation, which is a specific implementation of the underlying constructs, models, and methods that attempts to solve the practical problem. The following section further explains the thesis' contribution.

The creation of artifacts within DSR relies on existing kernel theories "that are applied, tested, modified, and extended through the experience, creativity, intuition, and problem solving capabilities of the researcher" (Hevner et al., 2004, p. 76). Gregor & Hevner (2013) refer to a kernel theory as any descriptive theory that informs artifact construction. Gregor & Hevner (2013) employ the term justificatory knowledge to be nearly synonymous with kernel theory, although with a slightly broader meaning. Justificatory knowledge includes any knowledge that informs design research, including experience from practitioners and knowledge from the field. Moreover, when attempting to make contributions, the research should desirably be formally grounded in kernel theories (Gregor & Hevner, 2013). Doing so brings additional justification to the contribution, as they are built on prior research. This thesis relies on CBSE as its kernel theory for artifact construction based on its relevance to the problem discussed in the previous chapter. Additionally, justification knowledge includes both empirical data and literature that informs artifact construction.

4.1.2 Research contribution

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

Table 6. DSR Contribution types adapted from Gregor & Hevner (2013, p. 342)

Gregor & Hevner (2013, p. 342) presents a table for DSR contribution types consisting of three different levels (Table 6 above). On level one, the knowledge is specific, limited, and less mature, while on level three, the knowledge is more abstract, complete, and mature. As such, the knowledge ranges from less abstract to more abstract, from limited to more complete. Level one consists of a situated implementation of an artifact, level two of nascent design theory – knowledge as operational principles or architecture, while level three consists of well-developed design theory about embedded phenomena. A specific DSR project can contribute on one or more of these levels. To put it into contrast, a contribution on level one could be the instantiation itself, without any design theory. In contrast, level three would be either mid-range or grand design theories. The contribution of this thesis is positioned on levels one and two, being an instantiation (level 1) and a set of design principles describing nascent design theory through a set of design principles (level 2). This nascent design theory is the main contribution, as the knowledge is more mature and complete. Nonetheless, the instantiation is a valuable contribution as well.

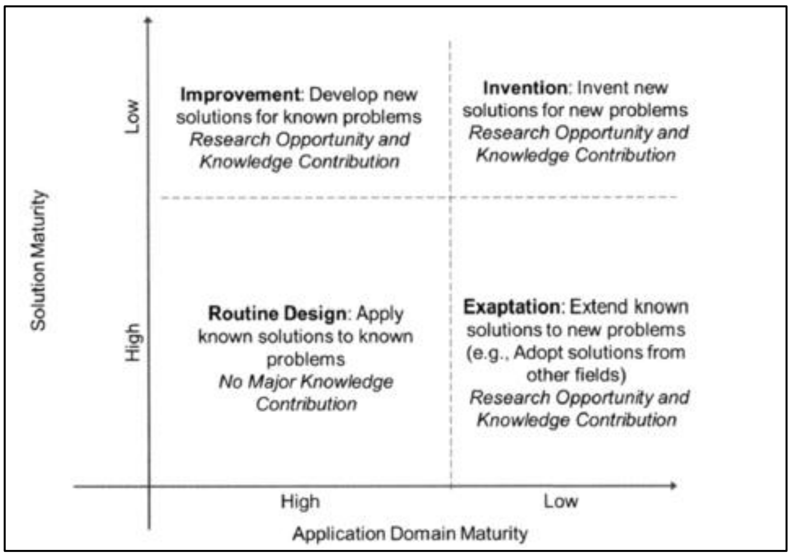

Solution Maturity
Low
High
Improvement: Develop new solutions for known problems *Research Opportunity and Knowledge Contribution*
Invention: Invent new solutions for new problems *Research Opportunity and Knowledge Contribution*
Routine Design: Apply known solutions to known problems *No Major Knowledge Contribution*
Exaptation: Extend known solutions to new problems (e.g., Adopt solutions from other fields) *Research Opportunity and Knowledge Contribution*
High
Low
Application Domain Maturity

Figure 2. DSR Knowledge Contribution Framework reprinted from Gregor & Hevner (2013, p.345)

Additionally, Gregor & Hevner (2013, p.345) present a DSR Knowledge Contribution Framework, a matrix consisting of four quadrants relating to the solution and application domain maturity regarding the contribution of the research (Figure 2 above). The first quadrant on low solution maturity and high application domain maturity is “Improvement,” which is about developing new solutions to known problems. Then, on low solution maturity and low application domain maturity is “Invention,” which is about inventing new solutions to new problems. Next, on high solution maturity and high application domain maturity is “Routine Design,” which is about applying known solutions to known problems, and is by Gregor & Hevner (2013) not regarded as research at all, but instead consultancy work. Finally, on high solution maturity and low application domain maturity is “Exaptation,” which is about extending known solutions to new problems. This thesis is positioned within this last quadrant. Through adopting solutions from CBSE, it applies them to the problem that software components created by implementation groups to aid them during localization are not reused. By doing this, known solutions from CBSE in the software engineering field are applied to enterprise software platform ecosystems in the IS field.

The generation of design principles in this thesis has been a linear process. The design principles were developed based upon different sources and types of knowledge. Table 7 below is based upon four interrogative dimensions for defining design principles from Purao et al. (2020). The four dimensions are described as such: Influences – What are the key influences? Temporality – When are they generated? Actors – Who identified them? Content – How are they documented? Table 7 below summarizes what factors influenced the design principles in this thesis; when they were generated, who generated them, and how their content is written. The initial design principles are described in section "5.6 Initial design principles", while the final design principles are presented in section "6.7 Final design principles".

Dimension	Application
Influences	Initial design principles emerged from the design, development, and evaluation process. Final design principles are elaborated and justified based on an evaluation and CBSE theory.
Temporality	Initial design principles were formulated during the design, development, and evaluation process. Final design principles are elaborated and justified after the process was complete.
Actors	After design, development, and ongoing, formative evaluations, I formulated the initial design principles—the final set based on an evaluation with a member of the DHIS2 Core Team.
Content	Initial design principles were formulated as a set of simple assertions. The elaborated and justified design principles follow the structure defined by Gregor et al. (2020). This structure splits the design principles into four sections. The first, “Aim, Implementer, and User,” describes who the design principle is for, the overarching aim, and who should implement it. The second, “Context,” describes for what context the design principle is created. The third, “Mechanism,” describes what must be accomplished to serve the aim. Finally, “Rationale” is the justification for why the mechanism will serve to accomplish the aim. This structure is used to present the final design principles in section "6.7 Final design principles".

Table 7. Four dimensions for defining design principles.

Master’s students within HISP at UiO have a long tradition of performing action research during field trips to collaborating HISP groups. Action research is defined as using recognized research techniques to produce a description of changes to practice within the research cycle (Tripp, 2005). Following this definition, action research lends itself well to the planned process of developing an artifact and then introducing it in an organization to study its effect on practice through interventions. Action research was the preferred methodology before the outbreak of the COVID-19 pandemic, which imposed strict travel restrictions by official decrees—as such, performing relevant interventions and observing changes to practice was impossible. Therefore, DSR was found to be a suitable methodology as it allowed for creating and evaluating an artifact without the requirement to observe organizational changes. Looking back at the choice of methodology, I would now argue that following the DSR methodology is more suitable for this type of project than action research. The primary reason is DSR’s inherent focus on designing an artifact to address a problem situation while still building on prior research. Additionally, is the space DSR allows for conducting rigorous evaluations that measure the capabilities of the artifact.

This subchapter has discussed the DSR methodology and its essential elements in this project, which lends itself to engaged research projects to solve problems. It has introduced the DSRM Process Model presented by Peffers et al. (2007) as an overarching framework for conducting DSR projects, consisting of six activities: identify and motivate the problem, define objectives of a solution, design and development of artifact, demonstration, evaluation, and communication. CBSE is presented as a kernel theory based on its relevance to the problem and its potential to justify a solution. Lastly, the contribution of this thesis has been positioned and discussed concerning contribution types within DSR, where design principles as nascent design theory have been discussed as this thesis's main contribution.

4.2 Data collection

Collecting empirical data is essential to inform the problem, artifact creation, and evaluation. In addition, empirical data is essential to justifying the design principles contributed from a DSR project with evidence. Throughout this project, three main categories of inquiry have been used for collecting data.

The first category is diagnostic data collected through interviews with HISP groups. It was necessary to get an overview of their app development and component reuse process to better understand the context and problem. This data created an understanding of their current development process, component reuse process, and resources involved in these processes. This data is the foundation to create an understanding of the research problem and subsequently inform artifact construction.

The second category is the data collected through the design and development of the artifact. Using the design and development activity as an exploratory process to address the research problem makes the artifact's implemented functionality a direct reflection of both what worked and how it worked. Additionally, remnants embedded in the artifact reflect ideas that did not work. These experiences from the design and development process and the data embedded in the artifact are an essential source of data and have helped construct the design principles. Examples of such data are the artifact's direct implementation of a website with a component gallery for component discovery. This data provides evidence of how it is feasible to implement this functionality, documented in the artifact's source code.

The third category of data is the evaluations performed on the artifact. These evaluations provide evidence that the created artifact can achieve its purpose (Venable et al., 2012). Formative evaluations have been performed with HISP groups, the project supervisors, and the DHIS2 Core Team during the development process to inform artifact construction. One final evaluation was performed after the finished artifact construction process to inform the final design principles. Table 8 below summarizes the categories of inquiry.

Category of inquiry	Methods	Participants
Diagnostic	Interviews	HISP groups
Design and development	Design and development	The research team
Evaluations	Focus groups, expert evaluation	The DHIS2 Core Team, the project supervisors, HISP groups

Table 8. Categories of inquiry for data collection.

4.2.1 Participants

The participants in this study involved actors within the DHIS2 ecosystem representing different user groups. Members of the DHIS2 Core Team assisted in defining the objectives of a solution and the evaluation of the artifact as experts in developing resources for the DHIS2 ecosystem. In addition, members of HISP groups in Sub-Saharan Africa were involved in interviews regarding their current app development practices and component reuse processes. They also assisted in one short formative evaluation. Lastly, the project supervisors were available to evaluate the artifact to determine the objectives of a solution.

4.2.2 Methods

Specific methods were applied to conduct data collection. As the focus was on collecting qualitative data, the methods applied focused on direct engagement. The methods conducted are interviews, focus groups, design and development, and expert evaluation. The interviews were used for diagnostic data collection. The focus groups have been used to explore new topics and ideas as a group and evaluate and suggest improvements to the artifact. Design and development were used to construct the actual artifact, the experiences in doing so, and the embedded functionality in the artifact. Finally, expert evaluations were used to evaluate the artifact with domain experts in developing tools for the DHIS2 ecosystem. Because of the restrictions imposed by official decrees due to the COVID-19 pandemic, both physical and digital data collection methods have been used. Physical meetings were used when possible, while digital tools such as Zoom were used to perform digital meetings. Zoom is a digital application that handles meetings and conference rooms (*Video Conferencing, Web Conferencing, Webinars, Screen Sharing*, n.d.).

4.2.2.1 Interviews

Interviewing is a way to collect information from conversations and range from highly structured, semi-structured, to relatively unstructured (Crang & Cook, 2007). The interviews to collect data for this thesis have been semi-structured, where a general outline of questions and topics are prepared to guide the discussion. This interview guide was developed based upon a set of learning goals that informed what knowledge was to be gained from the interviews. These interviews were conducted with members of HISP groups regarding their app development processes and resources used during these processes. The topics included their organizational setup, the participant's role in the HISP group, their app development processes, component reuse processes, and plans. In retrospect, it is evident that I and my colleagues who conducted the interviews did not know exactly what we were doing, which is reflected in the general outline of questions and the follow-ups to those questions. Unnecessary amounts of time were spent discussing their organizational setup and broader topics compared to their component reuse process. Because of this, most of the data collected through the interviews are more on broader topics instead of investigating the details of the component reuse process, which is the most important to investigate. The interviews were performed through Zoom, where each participant joined a collective meeting room from their device.

4.2.2.2 Focus groups

Focus groups are a way to explore topics in a setting where groups of people discuss their thoughts and experiences in a social context and with the researcher (Crang & Cook, 2007). In this project, focus groups were used to explore topics related to artifact construction and evaluation, such as ideas for features during construction and improvements during evaluation. These focus groups have been set up so that the participants are in a setting where it is natural for them to discuss the topics of relevance, for instance, a focus group where members of the DHIS2 Core Team discuss the objectives of the component management system. Within DSR, Tremblay et al. (2010) say that focus groups must be adapted to meet two specific goals of design research. The first is exploratory focus groups to study the artifact to propose improvements and new ideas for artifact refinement. The second is confirmatory focus groups to field test the artifact to tests its utility. This project made use of the exploratory focus groups, although they did not follow a set structure. For example, some of the focus groups were only discussions, while some of them included a presentation by the research team, while some others included a demonstration of the artifact.

4.2.2.3 Design and development

The design and development of the artifact are viewed as a method of data collection by itself. Hevner et al. (2004, p. 88) describe design as a "search process to discover an effective solution to a problem." Design and development involved exploring ideas, prototyping, designing, and developing features embedded into the artifact. Throughout this process, the experiences gained, and the discussions around problems are among the primary influences of artifact construction. The outcome of these discussions is then embedded in the artifact as implemented functionality, which directly reflects what did and did not work. The data collected through design and development is therefore twofold. First, it is the experiences gained by the research team by being involved in this process. Second is the embedded functionality of the artifact itself, which is a clear representation of what worked. Both are invaluable to the generation of this thesis's contributed design principles.

4.2.2.4 Expert evaluation

Expert evaluation is the assessment of the artifact by one or more experts (Peffers et al., 2012). While "experts" can be a loose term, it is referred to as someone with expertise in relevant subject areas. In this thesis, the collaborating members of the DHIS2 Core Team are defined as experts in developing tools for the DHIS2 ecosystem and are used in the expert evaluations. Experts have been used to evaluate the artifact for the sake of discussing potential improvements and shortcomings

This section has presented four methods for data collection. First, interviews are used to collect diagnostic data with HISP groups. Second, focus groups are used to explore ideas and evaluate the artifact. Third, design and development are used to construct the artifact and its embedded experiences and functionality. Finally, expert evaluation is used to evaluate the outcomes of the artifact.

4.2.3 Evaluation

The evaluation activity of DSR projects is essential. Hevner et al. (2004, p. 85) explain that the "utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods." Through evaluations, one can provide evidence that a new technology developed in DSR projects achieves its purpose (Venable et al., 2012). For evaluations, the collected data aims to provide such evidence. Depending on the type of artifact, the phase, and the project context, different evaluations are available. The framework presented by Venable et al. (2012) provides a comprehensive, step-by-step guide on how to conduct evaluations, what properties to evaluate, what contextual factors (i.e., where the project is in its lifespan), and other factors to consider when conducting evaluations for a designed artifact. This framework is used as Venable et al. (2012) present it: as a guide to help researchers determine the best type of evaluation at any specific point in the project. This section presents the overarching evaluation framework used to construct the evaluation conducted in this project.

The framework contains a four-step method for evaluation research design, which this section broadly presents (Venable et al., 2012). Venable et al. (2012) explain the first step as analyzing the context of the evaluation and deciding the evaluation requirements. This step includes identifying, analyzing, and prioritizing all goals for the evaluation. The second step is matching the factors from the first step to determine the evaluation’s strategy. The third step is selecting appropriate evaluation methods based on the evaluation’s strategy. Finally, the fourth step uses the evaluation strategy and methods and designs the evaluation itself in detail.

Venable et al. (2012) explain that a project should have more than one evaluation, where an ex ante evaluation should precede an ex post evaluation. Ex ante evaluations align themselves to what is called formative evaluations, while ex post evaluations align themselves to summative evaluations. The goal of formative evaluations is to measure the artifact’s ability to solve the problem to gather data that can further inform artifact construction (Venable et al., 2012). One performs these throughout the early stages of the project, focusing on prototyping, design, and development. The goal of summative evaluations is to use the artifact in a real user context and collect data on its ability to solve the problem. One performs these in the later stages of the project, focusing on the finished artifact.

Venable et al. (2012) explain the other dimension in the framework as artificial and naturalistic. Artificial means a focus on the efficacy of the artifact. Efficacy here means the degree to which the artifact performs in reaching its goals considered narrowly, without addressing situational concerns (Prat et al., 2015). In another way, this evaluates if the artifact can solve the problem or has the necessary capabilities to solve the problem. Naturalistic means a focus on the effectiveness of the artifact. Effectiveness here means the degree to which the artifact actually achieves its goal in a real context addressing real problems with real users (Prat et al., 2015). This evaluation would involve evaluating the artifact in a naturalistic research site, where it is possible to test if the artifact is solving a real problem with real users. Artificial evaluations are conducted over less time than naturalistic ones that need more prolonged periods. Both ex ante and ex post, and artificial and naturalistic, can be considered on a scale. Usually, the artifact in a project is not entirely within one dimension but can be placed depending on the goals of the evaluation.

To summarize, evaluations are essential to rigorously demonstrate the utility, quality, and efficacy of a designed artifact (Hevner et al., 2004). Furthermore, through evaluations, one can provide evidence that a new technology developed in DSR projects achieves its purpose (Venable et al., 2012). The framework presented by Venable et al. (2012) has been introduced to inform the final evaluation design in this project. This step-by-step guide is used in section "4.4.4 Demonstration and evaluation" to construct the evaluation in this thesis.

This subchapter on data collection has presented the main categories of inquiry: diagnostic, design and development, and evaluations. Within each of these categories, methods such as focus groups, interviews, design and development, and expert evaluation are performed in collaboration with the DHIS2 Core Team, the project supervisors, and HISP groups.

4.3 Data analysis

Data analysis is essential in research to understand and make sense of collected data and provide evidence of the performed process's validity. This section presents transcriptions, a research diary, and design and development activities as applied approaches to data documentation and subsequent analysis. The project has followed an inductive approach to analysis where text is analyzed thematically (Braun & Clarke, 2006), and sections of text are coded to identify themes. It was essential to identify themes that emerged during the empirical data collection before taking a deductive approach by relating the analyzed data to existing CBSE literature. The analysis approach is further described later in this section.

4.3.1 Data documentation

Data has mainly been documented in three ways. The first are transcriptions of the performed interviews, focus groups, evaluations. The second is through an informal research diary, where notes, thoughts, and reflections from the research process have been noted. The third is through development, where data is embedded into the source code of the artifact. These are different forms of documented data, and as such, need to be treated differently during their analysis.

4.3.2 Data analysis

Data analysis is how one makes sense of the documented data. The purposes for data analysis have been for three purposes. First to understand the data collected, second to inform artifact construction, and third to reason about the validity of the design principles. For example, diagnostic data is gathered and analyzed to inform the problem. In contrast, development and the subsequent evaluations are analyzed to inform artifact construction and reason about the project’s contribution. The analysis methods are presented following how the data was documented. Additionally, the research diary was not directly analyzed but instead served as a complementary source of information to add context to the analysis.

Transcriptions from interviews and evaluations have been analyzed thematically (Braun & Clarke, 2006), where sections of text are coded as a means to identify themes. Additionally, the level of generalization was kept low, to be specific. First, these are created by going through the text to categorize relevant aspects. Second, the codes are given relation or structured hierarchically. This way, relevant data is combined to inform more prominent themes. Thus, a more precise overview of the essential themes is presented. For example, discoverability of components, app development issues during the development process, and “component library” were some of the themes developed. The data was afterward deductively analyzed concerning related research (Fischer & Gregor, 2011). For example, CBSE discusses barriers to successful component reuse. The data collected show the same barriers being experienced by HISP groups, i.e., discoverability of components. Therefore, these barriers are limiting HISP groups' successful component reuse.

The analysis of the artifact is ongoing reasoning about its performance and why it performs as it does. The artifact created were primarily analyzed through discussions with my colleagues or during evaluations. Direct lines of code were not analyzed. Instead, the artifact's functionality and the feasibility to create such an artifact. The artifact is analyzed by using it to solve problems and subsequent discussion related to implemented functionality. These discussions are essential to make sure my colleagues and I understand the artifact and reason about its benefits and shortcomings. The design principles – the contributions of this type of DSR project – are the outcome of such an analysis process.

This project has primarily followed an inductive analysis method as it was essential to analyze the empirical data to infer themes that emerged during data collection. Furthermore, the creation of design principles was based mainly upon empirical data and the artifact's analysis. Additionally, the project used deductive analysis to connect the empirical data to the literature on CBSE. Table 9 below summarizes the approaches to data documentation and the analysis applied in this project.

Documented data	Applied analysis
Transcriptions from interviews, focus groups, and evaluations	Thematic analysis. Coding sections of text into themes
Research diary	Not directly analyzed. Served as context for the other sources of information
The artifact's source code	Through discussions with my colleagues and evaluations to extract design principles

Table 9. Conducted data documentation and analysis.

So far, this chapter has constructed the abstract research approach. First, Design Science Research (DSR) was introduced as the overarching methodology, with concepts such as justification knowledge to support artifact construction and design principles to refer to this project's contribution. Additionally, the DSRM Process Model by Peffers et al. (2007, p. 54) was presented as the overarching framework that has guided the steps of my research.

Next, data collection and categories of inquiry were presented to collect diagnostic data, design and develop the artifact, and lastly, conduct evaluations that further inform artifact construction. Thus, methods such as focus groups, interviews, design and development, and expert evaluation were presented to collect the data.

Finally, approaches to inductive data analysis were presented to understand and make sense of collected data and provide evidence of the performed process's validity. The transcribed interviews, focus groups, and evaluations were analyzed thematically, where sections of text are coded to infer themes. Further, the emerging themes are deductively connected to existing CBSE literature. Additionally, the artifact's source code was presented as documented data analyzed through discussions and evaluations.

To conclude, this section has presented the abstract research approach created by building on the DSR methodology. These aspects are summarized in Table 10 below. The following section presents the implemented research process describing how these aspects were implemented.

Aspect	Description
Methodology	Design Science Research (DSR).
Methodology process model	DSRM Process Model consisting of six activities: 1. Problem identification and motivation 2. Defining the objectives of a solution 3. Design and development 4. Demonstration 5. Evaluation 6. Communication
Kernel theory	Component-based software engineering (CBSE) to inform artifact construction, provide a vocabulary to discuss the artifact, and analyze implementation groups reuse processes.
Research contribution	Design principles with prescriptive knowledge extracted from the artifact.
Data collection	Diagnostic, design and development, and evaluations as categories of inquiry. Interviews, focus groups, design and development, and expert evaluations as methods performed.
Data analysis	Thematic analysis of the transcribed text and discussion and evaluation of the artifact.
Participants	The vendor, implementation groups, and the project supervisors.

Table 10. Summary of the abstract research approach.

4.4 Implemented research process

This subchapter presents the activities and steps performed throughout the implemented research process. The early activities of this project (ca. 10.2019 – 06.2020) were performed before a choice of methodology was made. Therefore, the activities performed throughout this period have been retroactively categorized into the DSRM Process Model by Peffers et al. (2007), described in section "4.1 Methodology". Furthermore, performing this retroactive connection has helped me extract the most important goals and activities performed through this period into a format that is easier to communicate.

Even though the DSR methodology by Peffers et al. (2007) describes six different activities conducted throughout a DSR project, they are presented as iterative. Compared to a linear process where one conducts one activity after another until the process is finished, iterative means one might often return to an earlier activity as one may always come across new findings that impact earlier activities. I find this highly relatable, where, i.e., discoveries made during the design and development activity sheds new light on the problem or the objectives of the solution.

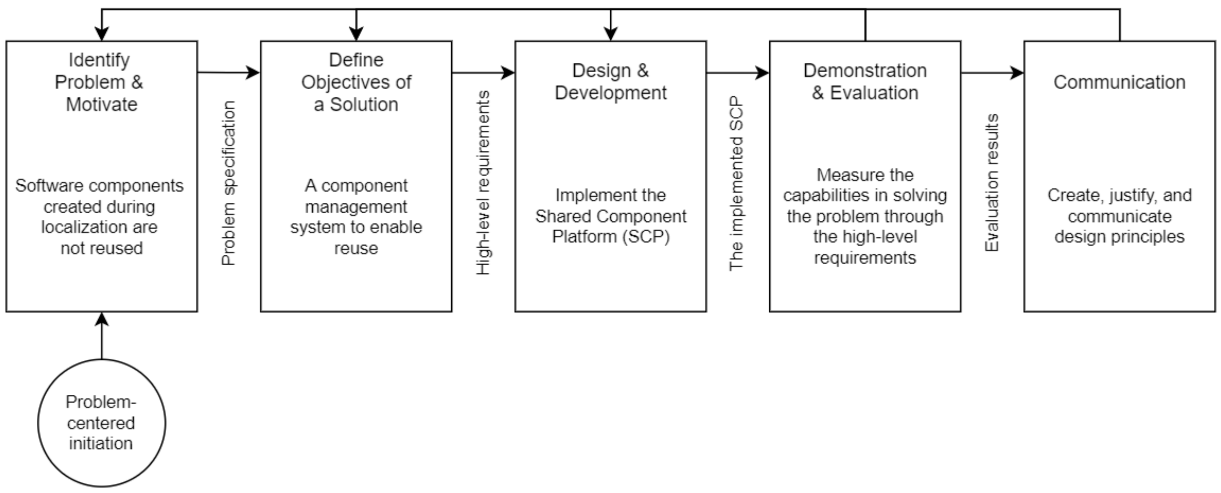

Identify Problem & Motivate
Software components created during localization are not reused
Problem specification
Define Objectives of a Solution
A component management system to enable reuse
High-level requirements
Design & Development
Implement the Shared Component Platform (SCP)
The implemented SCP
Demonstration & Evaluation
Measure the capabilities in solving the problem through the high-level requirements
Evaluation results
Communication
Create, justify, and communicate design principles
Problem-centered initiation

Figure 3. Implemented research process, adapted from Peffers et al. (2007, p. 54).

Figure 3 above presents the implemented research process, adapted from the DSRM Process Model presented by Peffers et al. (2007, p. 54). It shows how the implemented research process had a problem-centered initiation based on the problem that software components created by HISP groups to aid them during localization are not reused. Afterward, defining the objectives of a solution consisted of creating the Shared Component Platform's (SCP) high-level requirements based on the problem specification from the previous activity. Next, the design and development activity consisted of implementing these high-level requirements into the SCP. Then, the demonstration and evaluation activity consisted of using the implemented SCP to measure its capabilities in solving the problem and the high-level requirements. Methods conducted in this activity were focus groups and expert evaluations with the DHIS2 Core Team, HISP groups, and project supervisors. Finally, the communication activity consisted of extracting more generalized design principles from the SCP using the evaluation results as justification.

As shown in Figure 3 above, the demonstration and evaluation activity and the communication activity feed back into the three first activities. Thus, the learnings collected in the two last activities further inform the problem specification, the high-level requirements, and the implemented artifact itself. There are two primary differences between the implemented research process and the DSRM Process Model by Peffers et al. (2007). The first difference is the merging of the demonstration and evaluation activities into one activity. Instead of being separate, these were always combined in the implemented research process. The second difference is the iteration back to the problem identification and motivation activity. Instead of being set in stone, the problem was revisited frequently when relevant data was collected. Both differences are further explained in their corresponding sections later in this chapter.

It is essential to understand that even though the main activities of the research process are presented chronologically, aspects of the process happened simultaneously. Throughout this process, getting access to people proved to be a time-consuming task with much uncertainty. Because of this, my colleagues and I had to be flexible regarding when certain data collection instances occurred and adapt our process to when stakeholders had the opportunity to meet with us. An overview of the data collection methods is shown in Table 11 below.

Data collection method	Description
Interviews	In total 2 interviews on app development practices and resources with two different HISP groups.
Focus groups	In total 4 focus groups with the DHIS2 Core Team, HISP groups, and the project supervisors.
Expert evaluations	In total 1 expert evaluation with a member of the DHIS2 Core Team split over two sessions.
Design and development	The implemented functionality that embeds decisions and experiences by the research team.

Table 11. Data collection methods.

The rest of this subchapter follows the structure of the five activities presented in Figure 3 above by building on the DSRM Process Model: problem identification and motivation, defining the objectives of a solution, design and development, demonstration and evaluation, and communication. Each of these activities presents the events that happened within them.

4.4.1 Problem identification and motivation

The first activity of the project was problem identification and motivation. Two sources of data collection concerning the problem identification and motivation were performed during this activity, as shown in Table 12 below. The participants, group A and group B, are used later in section "5.3 HISP groups' app development processes" to refer to the data collected during these interviews.

#	Month	Method	Participants	Description
1	July 2020	Interview	HISP group A	Interview with members of HISP group A on their app development practices and component reuse processes.
2	October 2020	Interview	HISP group B	Interview with members of HISP group B on their app development practices and component reuse processes

Table 12. Data collection on problem identification and motivation.

The initial problem was identified during earlier projects conducted within the DHIS2 ecosystem, and the DHIS2 Design Lab responded to this challenge (Li, 2019a). (Li, 2019a, para. 1) presents the challenge as “designing, developing and evaluating a platform that facilitates the sharing of reusable web-components for use in web-based app development within a generic software platform.” Using this challenge as the foundation, this activity involved investigating literature on platform theory to position the problem and understand what had been done previously.

Additionally, interviews were conducted with HISP groups to understand their app development processes to inform the research problem. By revisiting the problem specification throughout the project, it was defined through a three-leveled list, as shown in Table 13 below. The problem on level 1 is very open and generic, while the problem on level 2 is narrower, while the problem on level 3 is the most specific being explored in this thesis. The idea is that by contributing to the problem at level 3, level 2 also gains a contribution, which can offer a small contribution to the overall problem described in level 1.

Level	Problem
1	Generic enterprise software does not work optimally if it is not localized (Li & Nielsen, 2019; Wareham et al., 2013).
2	Developing apps as an approach to localization is resource-intensive for implementation groups (Li & Nielsen, 2019; Wareham et al., 2013).
3	Software components created by implementation groups to aid them during localization are not reused.

Table 13. The three-leveled problem specification.

4.4.2 Defining the objectives of a solution

The second activity of the process was defining the objectives of a solution. This definition can come in qualitative and quantitative forms, such as terms in which a desirable solution is better than current ones, as a description of how the solution solves a problem, or as specific requirements the solution needs to implement. The SCP's objectives are the high-level requirements that guide the research project based on the problem specification fed from the previous activity.

This activity started with a discussion and brainstorming with my colleagues. We tried to imagine what a solution could look like based on the problem and what was technically feasible. After brainstorming, there was a focus group with the DHIS2 Core Team members and a visiting member of a HISP group (event 1 in Table 14 below). This event discussed the initial objectives and goals and how they could be implemented. The outcome was taken and developed into a more coherent set of objectives by sending this set of objectives to these members of the DHIS2 Core Team for feedback, getting their new thoughts and ideas, and altering it accordingly. This set of objectives served as the initial objectives to use during the design and development activity.

After some time spent on design and development, a focus group with the project supervisors consisted of demonstration and discussion (event 2 in Table 14 below). During this focus group, the utility for developers was taken into consideration. Questions such as "why are you building this" and "why is it useful?" were brought up for discussion by the project supervisors, which resulted in exploration in how the SCP can be most useful.

Later, there was a new focus group with the DHIS2 Core Team (event 3 in Table 14 below), where new objectives to enhance its utility for developers were discussed. The primary example is the distinction between *packages* and *components*, mentioned in the previous chapter in section "3.2.2. Central concepts in CBSE". It will be discussed in detail later in the thesis in section "5.1 Node Package Manager (NPM)". The central premise is that packages encapsulate one or more components. This distinction is vital for the SCP's functionality.

Afterward, there was a focus group with members of a HISP group (event 4 in Table 14 below), where it was confirmed that the set objectives so far were suitable. After some time developing, there was a need to define a more coherent set of certification requirements, which spurred informal communication with a member of the DHIS2 Core Team (event 5 in Table 14 below), where a more coherent consensus around certification guidelines was reached. Finally, after the design and development of the SCP were finished, there was a need to provide a final evaluation. The evaluation was conducted as an expert evaluation with a member of the DHIS2 Core Team (event 6 in Table 14 below).

Event	Description	Outcome	Month
1	Initial focus group with three members of the DHIS2 Core Team, a member of a HISP group, and project supervisors. Follow-ups on mail afterward.	Decided on initial objectives, architecture, modules, and goals. A website to display components, ideas on how to store them, and ideas on certification.	February 2020
2	Focus group with project supervisors.	Explore how the SCP can be more useful for developers.	September 2020
3	Focus group with members of the DHIS2 Core Team.	Improvements for the SCP: present components instead of packages, whitelist for certification	October 2020
4	Focus group with members of a HISP group.	Confirmation that the set objectives are suitable.	October 2020
5	Informal talk with a member of the DHIS2 Core Team	Better constructed certification guidelines.	October 2020
6	Expert evaluation with a member of the DHIS2 Core Team split over two sessions (after the development period is over.)	Improvements for the SCP: whitelist certification handling, how components are presented on the website, and more. Presented in detail in "Chapter 6: Final Evaluation."	December 2020 and February 20201

Table 14. DSR process iterations.

One type of event that is not mentioned here is the subsequent need to change the objectives based on the activities performed in the "design and development" activity. For example, we would often try to implement some functionality during design and development but needed to change the SCP's objectives to make it work based on shortcomings either in the SCP or third-party services. Such small changes happened continuously throughout the "design and development" activity, so it is not presented as specific events. To conclude, the events presented in Table 14 above form an overview of the events that impacted the SCP's design apart from those taken by the research team.

4.4.3 Design and development

The third activity of the process was design and development. During this event, design and development occurred interchangeably. This activity aimed to create the artifact, in our case, to create the SCP. As mentioned in the previous activity, defining the objectives of a solution, the objectives were changed based on interviews, discussions, and evaluations and based on design and development. Thus, the design and development activity is fed the high-level requirements from the previous activity.

One can split this activity into two distinct steps, *design* and *development*. Design involves using the objectives defined in the previous activity to determine the SCP's required functionality and architecture. Development involves creating and programming the actual SCP. The initial design of the SCP started after the initial objectives were set in event 1. The initial design involved choosing development languages, third-party services to integrate with, and designing the system's architecture. Then my colleagues and I would separate tasks in between ourselves and work on those tasks. These tasks could take the form of direct functionality that needed to be implemented or a problem needing a solution.

For each subsequent change in the objectives, design decisions for the SCP would also need to be made if appropriate. One such example is when the decision, based on evaluations, of creating the distinction between *packages* and *components* was made to increase the utility. This decision influenced the objectives, which influenced the design, which required a time-consuming refactoring of the codebase during development. Finally, after the last iteration of design and development, a more presentable and cohesive system was made.

4.4.4 Demonstration and evaluation

The fourth activity is demonstration and evaluation combined. First, demonstrations are conducted to demonstrate how well the SCP can solve the problem and then perform subsequent evaluations to measure its ability. During the research process, these happened simultaneously and are therefore merged into the same activity. Finally, demonstration and evaluation are fed the implemented SCP from the previous activity to be used in the activity.

This project has used two different methods for evaluations: focus groups and expert evaluations. Throughout the early stages of the project, these evaluations were in the shape of focus groups. During these focus groups, concepts of importance and their implementation in the SCP were evaluated. These evaluations were conducted in collaboration with the DHIS2 Core Team, the project supervisors, and HISP groups.

After more research into DSR and understanding the importance of rigorous evaluations, more focus was put on making a conscious decision on criteria to evaluate and the methods to evaluate them. The evaluations can, in general, be summed up as occurring in three different phases. The first was before the development of the artifact. The second was during the active development of the artifact. The third was after the development of the artifact was finished.

Additionally, throughout this project, the evaluations can be classified as formative. The focus has been on collecting data that further informs the SCP's design and development. The categorized evaluations are shown in Table 15 below. The events are from Table 14 above, categorized as evaluations based on their purpose to inform further artifact construction.

Phase	Description	Methods
Before development	Decide on initial requirements, event 1 in Table 14 above.	Focus groups
During development	Use SCP to inform further artifact construction, events 2-5 inclusively in Table 14 above.	Focus groups, informal communication

After development	Use SCP to inform further artifact construction and finalize the project. An expert evaluation was split over two sessions with a member of the DHIS2 Core Team, event 6 in Table 14 above.	Expert evaluation

Table 15. Evaluations in the research process.

After the development of the SCP was ending because of the project's deadline, it was an active thought to perform more rigorous, formative evaluations on the outcomes of the development process. The evaluations performed in the "before development" and "during development" phases in Table 15 above did not follow a rigorous evaluation design. In contrast, the expert evaluation conducted in the "after development" phase has followed the previously described "Four-Step Method for DSR Evaluation Research Design" by Venable et al. (2012, p. 434-435). The four steps are broadly described to recap the framework: (1) Analyzing the context and the evaluation requirements, (2) match the contextual factors with the presented framework, (3) select appropriate evaluation methods, and (4) designing the evaluation in detail. Table 16 below presents the steps, their description, and the implemented step's outcome fed into the next step.

Step	Description	Outcome
1	Analyzed the evaluation's goals, constraints in the research environment, required rigor, and evaluation criteria.	Results in prioritized factors: goals, constraints, rigor, and evaluation criteria. Finally, these are fed to the next step.
2	The factors from the previous step are matched with a 2x2 matrix to determine the evaluation strategy.	Determined the formative, artificial evaluation strategy with elements from naturalistic based on the SCP's socio-technic nature.
3	Uses the evaluation strategy from the previous step to select appropriate evaluation methods.	The expert evaluation method is selected and fed to the next step.
4	The evaluation method selected in the previous step is used to design the evaluation in detail.	A designed expert evaluation over two sessions with a member of the DHIS2 Core Team.

Table 16. The implemented four-step method for evaluation design from Venable et al. (2012, p. 434-435).

This section further describes the implementation of the framework and the four-step method presented by Venable et al. (2012). These steps are used to provide guidance in selecting and designing an appropriate evaluation method (Venable et al., 2012). This section presents what is implemented in each step and what it feeds to the next step. Additionally, this section provides motivation and justification for why the expert evaluation method was selected. Finally, this section describes how the evaluation is designed and performed with a member of the DHIS2 Core Team.

4.4.4.1 Analyze the context and the evaluation requirements

The first step is to analyze the context and the evaluation requirements to determine the goal of the evaluation (Venable et al., 2012). Analyzing the context and the evaluation requirements aids in prioritizing these factors in determining the most suitable evaluation method (Venable et al., 2012). Based on what was written earlier in section "4.2.3 Evaluation", it was a question on how the SCP should be viewed. The SCP is viewed as a socio-technical artifact implementing functionality to support actors in social processes. Therefore, to evaluate the SCP, the goal is to evaluate how its functionality supports actors in their processes. These actors are the component producers, component users, and certifiers. Additionally, since the SCP is working but not complete, the evaluation is performed with a 'somewhat real system.'

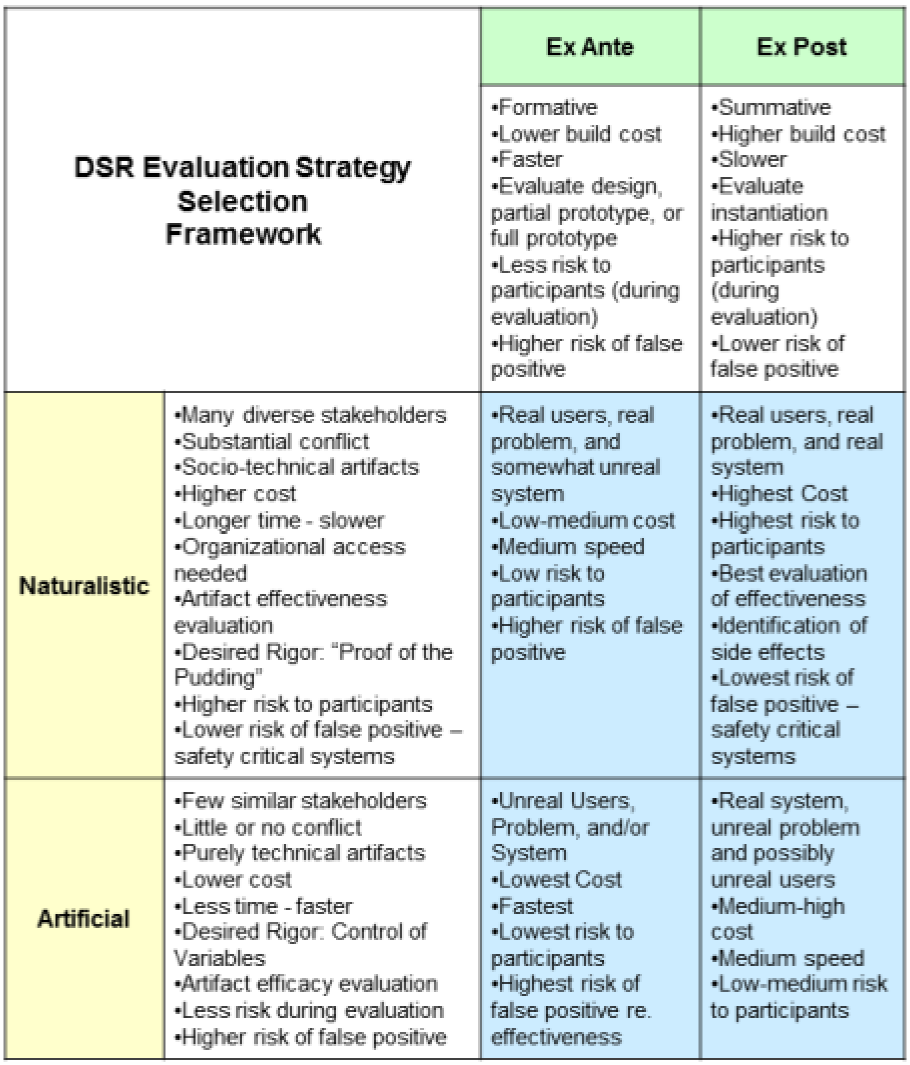
Regarding constraints in the research environment (Venable et al., 2012), time is limited because of the project deadline, and getting access to 'real users' has proven time-consuming. Additionally, a naturalistic research site with 'real users' is not available because of restrictions imposed by official decrees due to the COVID-19 pandemic. 'Real users' primarily refer to the target user group of the SCP, the HISP groups, as component producers and component users. However, 'real users' can also include certifiers currently represented by the DHIS2 Core Team. However, since they are not considered the primary user group, they will be referred to as 'somewhat real users.' Considering these factors, the evaluation should take little time, getting access to 'real users' should be avoided due to lack of time, and a naturalistic research site is not available. Thus, the evaluation should be performed with 'unreal users' or 'somewhat real users' and an 'unreal problem' since there is no naturalistic research site where real problems can be addressed.

Regarding the required rigor of the evaluation and the project's status (Venable et al., 2012), the evaluation can be formative. In general, summative evaluations require more rigor as they are meant to conclude a project and provide more definite answers. In contrast, formative evaluations require less rigor as they are meant to inform further artifact construction. Venable et al. (2012) explain that some parts of the evaluation can be done after the project's completion. More functionality must be implemented and tested before the SCP can be implemented to support actors in an organization based on its status. Therefore, the evaluation can be performed as a somewhat rigorous formative evaluation to inform artifact construction. Doing this opens up the possibility for new master's students in the DHIS2 Design Lab to take over the project, develop further, and conclude it with summative evaluations when it is ready to be implemented in an organization.

Based on what has been explained in the previous paragraphs, I argue that suitable properties to evaluate are the SCP's efficacy, technical feasibility, and utility (Prat et al., 2015). Using these evaluation criteria, the goals of this evaluation are split into three categories. First, efficacy measures if the created artifact has the ability to reach its goals without considering situational concerns (Prat et al., 2015; Venable et al., 2012). Efficacy answers the question: can the SCP address the research problem? Second, technical feasibility measures the ease at which the artifact can be built and operated from a technical point of view (Prat et al., 2015). Technical feasibility answers the question: is the SCP a good representation of its underlying abstract concepts and models? Third, utility measures the value of achieving the artifact's goals. Utility answers the question: can the SCP be useful if it reaches its goals? These evaluation criteria collectively measure the capabilities of the artifact.

To summarize, analyzing the context and evaluation requirements aids in prioritizing factors and determining the most suitable evaluation method (Venable et al., 2012). First, due to the SCP's socio-technical nature, the evaluation aims to measure how well the SCP's functionality supports component producers, component users, and certifiers in their processes. Additionally, due to constraints in the research environment, getting access to 'real users' and a naturalistic research site should be avoided due to lack of time, but 'somewhat real users' can be accessed. Further, the evaluation can be formative and does not require the rigor of a summative evaluation. Last, the criteria to measure is the SCP's efficacy, technical feasibility, and utility. The context and evaluation requirements are brought to the next step, where they are matched with the framework to select evaluation strategy (Venable et al., 2012).

4.4.4.2 Match the contextual factors

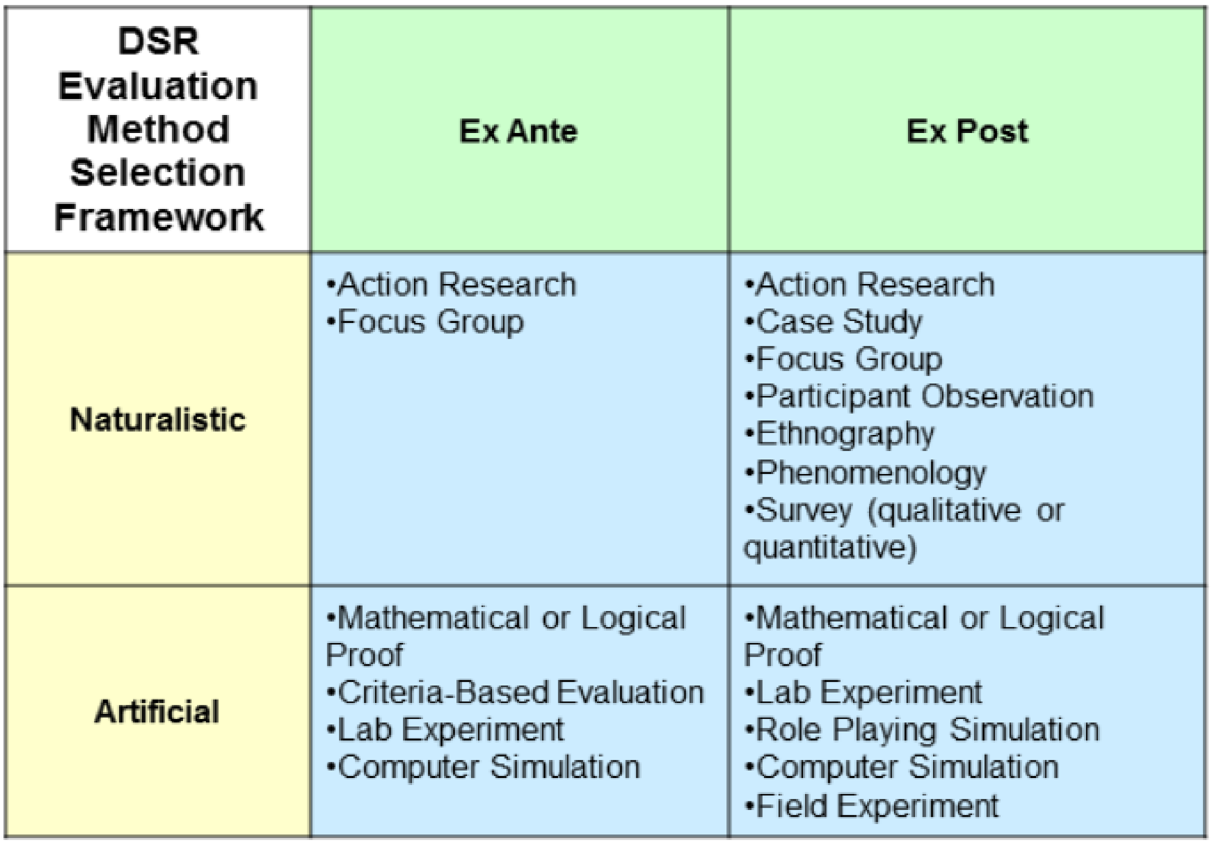


		Ex Ante	Ex Post
DSR Evaluation Strategy Selection Framework		•Formative •Lower build cost •Faster •Evaluate design, partial prototype, or full prototype •Less risk to participants (during evaluation) •Higher risk of false positive	•Summative •Higher build cost •Slower •Evaluate instantiation •Higher risk to participants (during evaluation) •Lower risk of false positive
Naturalistic	•Many diverse stakeholders •Substantial conflict •Socio-technical artifacts •Higher cost •Longer time - slower •Organizational access needed •Artifact effectiveness evaluation •Desired Rigor: "Proof of the Pudding" •Higher risk to participants •Lower risk of false positive – safety critical systems	•Real users, real problem, and somewhat unreal system •Low-medium cost •Medium speed •Low risk to participants •Higher risk of false positive	•Real users, real problem, and real system •Highest Cost •Highest risk to participants •Best evaluation of effectiveness •Identification of side effects •Lowest risk of false positive – safety critical systems
Artificial	•Few similar stakeholders •Little or no conflict •Purely technical artifacts •Lower cost •Less time - faster •Desired Rigor: Control of Variables •Artifact efficacy evaluation •Less risk during evaluation •Higher risk of false positive	•Unreal Users, Problem, and/or System •Lowest Cost •Fastest •Lowest risk to participants •Highest risk of false positive re. effectiveness	•Real system, unreal problem and possibly unreal users •Medium-high cost •Medium speed •Low-medium risk to participants

Figure 4. Matching the factors and requirements to the DSR Evaluation Strategy Selection Framework reprinted from Venable et al. (2012, p. 432)

This step matches the context and evaluation requirements presented in the previous step to those presented in the framework in Figure 4 above. One looks at the four white dimensions and the four blue quadrants to determine in which blue quadrant the evaluation should be placed. As Venable et al. (2012) explain, it may well be that multiple blue quadrants apply, indicating the need for a hybrid methods evaluation design. Thus, the blue quadrant that applies the most is the bottom left, where the artificial and ex ante (formative) axes meet. However, due to the SCP's socio-technical nature, aspects from the naturalistic dimension are also vital. Therefore, the evaluation follows the artificial and formative strategy and should incorporate as many naturalistic elements as possible. This strategy is brought to the next step to select appropriate evaluation methods.

4.4.4.3 Select the appropriate evaluation methods



DSR Evaluation Method Selection Framework	Ex Ante	Ex Post
Naturalistic	•Action Research •Focus Group	•Action Research •Case Study •Focus Group •Participant Observation •Ethnography •Phenomenology •Survey (qualitative or quantitative)
Artificial	•Mathematical or Logical Proof •Criteria-Based Evaluation •Lab Experiment •Computer Simulation	•Mathematical or Logical Proof •Lab Experiment •Role Playing Simulation •Computer Simulation •Field Experiment

Figure 5. Matching the contextual factors to the DSR Evaluation Method Selection Framework reprinted from Venable et al. (2012, p. 433)

This step uses the evaluation strategy determined in the previous step and determines appropriate evaluation methods. Figure 5 above presents a selected few to choose from within each blue quadrant, reflecting those from Figure 4 in the previous section. Additionally, one can explore extant literature to find methods that fit the high-level evaluation strategy. For instance, the expert evaluation method was suitable and generally considered artificial since it uses experts from the field (Peffers et al., 2012; Venable et al., 2012). The expert evaluation method fits well, considering the factors in the first step. Therefore, the next step will use the expert evaluation method to design the evaluation.

4.4.4.4 Design the evaluation in detail

This step uses the high-level evaluation strategy from the previous step to design the evaluation in detail. The expert evaluation would be performed with a member of the DHIS2 Core Team, an expert in developing tools and resources for the DHIS2 ecosystem. While the expert evaluation method is often considered artificial, it is also naturalistic in this context. As mentioned earlier, one of the SCP's supported processes is the certifiers', who are responsible for manually certifying components. Thus, the evaluation is also naturalistic, as the DHIS2 Core Team members have been discussed as the role of certifiers. Therefore, the DHIS2 Core Team member is not an expert unrelated to the problem but rather an invested actor in the SCP. To summarize using the same categorization as Venable et al. (2012), the evaluation is designed with a 'somewhat real system,' 'somewhat real users,' and 'unreal problem.' The problem is unreal as the SCP is not yet implemented in an organization where it can address 'real problems'; instead, it will address a simulated real problem.

The expert evaluation was split into two separate sessions, which focused on distinct parts of the SCP. In the first session, the DHIS2 Core Team member adopts the role of a HISP group member and tests the SCP's functionality in supporting its processes while solving a set of simulated problems. The second session is discussion-oriented, where each part of the SCP is discussed and scrutinized to look for improvements. So, while the first session measures efficacy and utility by directly using the SCP's implemented functionality to solve a problem, the second session attempts to measure efficacy, technical feasibility, and utility by building on the first session and opening for discussion.

The first session of the expert evaluation was performed with a member of the DHIS2 Core Team, further referred to as the 'participant.' To start, the participant went through a set of tasks on their computer during a Zoom call. The tasks make the participant interact with each of the modules of the SCP. These tasks took the participant through the process of creating a package with components, publishing this package to NPM, certifying the package, and then reusing the package:

1. As a component producer, create an NPM package with components and running pre-certification.
2. As a component producer, publish the package to NPM.
3. As a component producer, submit the package for certification.
4. As a certifier, decide if the package should be certified or not, then accepting or declining it.
5. As a component user, discover the components through the website.
6. As a component user, navigate to NPM to reuse the component.

Throughout this process, both the participant and I could ask questions about topics that emerged. Questions were used frequently and provided data around issues with existing functionality and ideas for more functionality. Towards the end, there was some room for further discussion around each module and how they interact.

The second session is a follow-up to expand on the data gathered from the first session. The original plan was to structure this as an expert focus group, with more members of the DHIS2 Core Team, but scheduling issues meant this instead needed to be changed to be performed with a single member on the fly. Thus, the evaluation is a follow-up with the same person as the first session, focusing on discussing the person's experience with the SCP. The evaluation consisted of questions related to the primary modules of the SCP, if the representation in the artifact is suitable, what improvements can be made, and if other options could be more suitable. Table 17 below presents the final evaluation in the "after development" phase.

Evaluation type	Participants	Properties measured
Expert evaluation session 1	DHIS2 Core Team member	Efficacy, utility
Expert evaluation session 2	DHIS2 Core Team member	Efficacy, technical feasibility, utility

Table 17. Conducted evaluation methods.

To summarize, this project has used two different evaluation methods: focus groups and expert evaluations. These were separated into the phases of "before development," "during development," and "after development." The evaluations conducted in the "before development" and "during development" phases followed the focus group evaluation method and involved members of the DHIS2 Core Team, HISP groups, and the project supervisors. The evaluation conducted in the "after development" phase was designed using the more rigorous and formal evaluation design framework by Venable et al. (2012). The expert evaluation method was conducted with a DHIS2 Core Team member to measure the SCP's efficacy, technical feasibility, and utility. Based on the SCP's socio-technic nature, the goal was to measure how well the implemented functionality can support component producers, component users, and certifiers in their processes. The results of the data collected in the "before development" and "during development" phases are presented in section "5.4 Artifact construction process". Finally, the results of the data collected in the "after development" phase are presented in "Chapter 6: Final Evaluation".

4.4.5 Communication

The fifth and last activity of the project was communication. During this activity, the problem and its importance, the utility and novelty of the artifact, and the rigor of its design are communicated to the relevant audiences (Peffers et al., 2007). This thesis is how the research project is communicated to other researchers through the created design principles. Both the thesis and the written artifact documentation are how the project is communicated to practicing professionals. The communication activity is fed the evaluation results from the previous activity.

This thesis uses the generation of design principles as its primary avenue for communication. These design principles are extracted from the implemented SCP and collected justification knowledge. As was mentioned in section "4.1.2 Research contribution", design principles were generated in two separate instances. The initial design principles were generated after the development of the SCP was ended, but before the expert evaluation discussed in the previous section was performed. These design principles are presented in section "5.6 Initial design principles". Afterward, those initial design principles are elaborated and justified based on data collected from the expert evaluation and presented in "6.7 Final design principles". Finally, the design principles are discussed with literature on CBSE in the Discussion chapter.

To summarize, the research process has involved five distinct activities: problem identification and motivation, defining objectives of a solution, design and development, demonstration and evaluation, and communication. The events presented in section "4.3.1 Problem identification and motivation" and section "4.3.2 Defining the objectives of a solution" collectively form an overview of the events that occurred during the research process. Many of these activities happened simultaneously, and Table 18 below presents the period they were part of the main focus. In addition, based on the spontaneous approach required to get access to stakeholders for data collection, many of these activities were revisited when the opportunity presented itself.

Activity	Period
Problem identification and motivation	10.2019 – 08.2021
Defining objectives of a solution	12.2019 – 02.2021
Design and development	06.2020 – 12.2020
Demonstration and evaluation	02.2020 – 02.2021
Communication	12.2020 – 05.2021

Table 18. Process overview.

4.5 Team management

Most of the practical parts of this project were done in collaboration with two other master's students within the DHIS2 Design Lab at the UiO. They have been referred to as my 'colleagues,' and we are collectively the 'research team.' This practical work includes the design and development of the SCP and most data collection. To consider separate angles and research questions, my colleagues and I conducted the final evaluation separately.

Because of the restrictions imposed by official decrees due to the COVID-19 pandemic, we have had to adapt most collaboration aspects to be digital. As such, our primary channels of communication were the digital, collaborative tools Slack, Mattermost, and Zoom. In addition, collaboration on the source code of the SCP occurred over the version control tool Git. Finally, we had physical meetings both for discussion and other types of group work when possible. However, the meetings did not follow a formalized structure throughout the majority of the research process. Instead, we brought up matters of importance for discussion concerning, i.e., the SCP's development, issues, and data collection.

The research team followed an Agile development approach, where we iteratively developed features. Most significant decisions relating to implementing the SCP were reached during discussions. Additionally, minor decisions regarding specific implementation details were left to the person working on it not to slow down the process unnecessarily. The reached decisions were written down and converted into functionality that was to be implemented by one or more group members. The team used a Kanban board in Trello to track this functionality. The board was split into three sections to visualize the development process: "To do," "Doing," and "Done." This board was used to delegate work between team members with set deadlines to incentivize each member to finish their tasks in time.

Based on a meeting to assess team performance further into the process, we attempted to create a formal meeting structure and schedule. The schedule included at least one meeting each week, structured as a scrum meeting. Each team member was asked three questions: What have I done since the last meeting? What am I currently working on? Do I need help with anything? These questions were used to get all team members up to speed regarding each part of the project and assist if problems occurred. During the group assessment meeting, the team also decided that the Kanban board should receive more attention. Therefore, we focused on better delegation of tasks during the scrum meetings and stricter deadlines.

All team members were involved in data collection activities, such as their planning and execution. The planning activity consisted of creating presentations, learnings goals, and interview guides. During the execution of data collection activities, each team member had designated sections where they were responsible.

The main contributions to creating the SCP are categorized into three aspects: Who came up with the ideas? Who found the solution to the ideas? Who implemented the functionality? Through this abstraction, it is easier to present my contribution. In general, we worked on three separate modules:

- Shared Component Platform (SCP) Website
- Shared Component Platform (SCP) Command-Line Interface (CLI)
- Shared Component Platform (SCP) Whitelist

My main contribution comes from the work done on the SCP Website, where I have been responsible for much work related to coming up with ideas, finding solutions to these ideas, and implementing them. I created the SCP Website's initial setup, meaning its bootstrapping, initial file structure, and initial design and layout. Afterward, all team members focused on the SCP Website for some time. After a while, I got the primary responsibility of the SCP Website, while another team member got the primary responsibility of the SCP CLI and the SCP Whitelist. The third team member did a bit of work on all three modules of the system. My main contribution comes from coming up with ideas, finding a solution, and implementing the solution in the SCP Website.

For the two other modules, the SCP CLI and the SCP Whitelist, my contribution does not include any direct implementation but rather through discussions around the ideas and their solution. Thus, my understanding of the SCP Website is much more in-depth than the SCP CLI and SCP Whitelist. Therefore, the thesis also reflects this by putting a heavier focus on describing the SCP Website. The "Appendix" includes a "Detailed work distribution" table for each module, presenting implementation details and by whom it was conducted.

4.6 Ethical considerations

Throughout the research process, some guidelines were put in place to consider participants' privacy and their right to confidentiality in the study. In research projects, it is vital to consider participants' privacy and not misuse their trust. As a researcher, one must understand that, i.e., citations and quotes from participants can put them in a bad light. Additionally, one has the responsibility to be critical of one's use of data and not to misuse or misrepresent what participants say.

A consent form was created detailing the participants' involvement in the project to consider the guidelines. First, the consent form explained the project's purpose, the people responsible for conducting the project, why the participant is asked to participate, and what that participation involves. Next, it describes that participation is voluntary and that participation can be withdrawn at any point without consequences. Last, it describes how the participant's data is stored and how the project uses their data. Care is taken to anonymize data such that information from data collection activities cannot be used to identify the participant. This consent form was given to each participant in the study to be signed.

4.7 Limitations of the applied method

This section describes the limitations of the applied method when compared to the presented DSR methodology. First, this section presents the limitations involved in understanding the research problem through conducting digital data collection. Afterward, this section describes the limitations of the evaluations performed in the "before development" and "during development" to visualize an enhanced process.

4.7.1 Digital data collection

Because of the restrictions imposed by official decrees due to the COVID-19 pandemic, most data collection throughout the process has been performed through digital tools. Regarding the diagnostic data collected with HISP groups in Sub-Saharan Africa to better understand the context and problems, I have relied solely on interviews over Zoom. Performing these interviews remotely means I cannot interact with actors directly, where I could have experienced more than what is explicitly said.

Additionally, the participants during these interviews did not use a web camera, meaning I had to rely purely on what was being said and how it was said. There can sometimes be a mismatch between what is said and what is done that is hard to gauge when I cannot experience and observe the practices myself. Suppose I had the opportunity to experience the problem directly and interview members of the HISP groups in a physical location. In that case, I think I would better understand the research problem on a deeper level as I can observe how it is addressed daily.

4.7.2 Evaluations

Even though DSR and especially the topic of artifact evaluation has not yet reached maturity in IS (Prat et al., 2015), there is still a consensus that rigorous evaluations are needed to provide evidence that the artifact fulfills its purpose (Hevner et al., 2004; Peffers et al., 2012; Venable et al., 2012). One limitation of the applied method is the evaluations performed in the "before development" and "during development" phases. Compared to the evaluation performed in the "after development" phase, which followed the framework for evaluation design by Venable et al. (2012), the evaluations in the earlier phases did not follow a framework for their design. Therefore, they were designed to the best of our ability without proper guidance from the DSR literature concerning what an 'evaluation' contains. Thus, it was challenging to determine the evaluation's goals and analyze its results precisely.

I think this was because the DSR literature had not been explored well enough during the early phases of the project. If I were to repeat the process, I would emphasize early creating a plan for envisioned evaluations. This plan is then the foundation for basing evaluations designed by following an accepted framework, such as the one by Venable et al. (2012). Thus, one gets guidance along the way in designing appropriate evaluations based on the context, the evaluation's goals, and its requirements.

The result is evaluations with clear goals and precise evaluation methods to measure those goals, as were described in "4.4.4 Demonstration and evaluation", where the framework by Venable et al. (2012) was used to create and justify the evaluation performed in the "after development" phase. Nonetheless, while the evaluations conducted in the "before development" and "during development" phases have not followed such a framework, they have been crucial in informing the artifact construction process.

I am generally satisfied with the applied method. However, it has some limitations regarding understanding the research problem through digital data collection and some evaluations designed without following commonly accepted evaluation design frameworks. Nevertheless, the applied method has resulted in a constructed component management system based on empirical data regarding the research problem and justified with empirical data through evaluations.

4.8 Summary

This chapter has presented the abstract research approach and the implemented research process. The implemented research process has involved five activities: problem identification and motivation, defining objectives of a solution, design and development, demonstration and evaluation, and communication. Throughout this process, three main categories of inquiry are applied: diagnostic, design and development, and evaluation, to collect data in collaboration with members of the DHIS2 Core Team, HISP groups, and the project supervisors. First, Table 12 in "4.4.1 Problem identification and motivation" presents the interviews conducted to inform the research problem. Next, Table 14, with events presented in "4.3.2 Defining the objectives of a solution", forms an overview of the events that impacted the SCP's design choices apart from those taken by the research team. Finally, combined with Figure 3 in "4.4 Implemented Research Process" and the evaluation phases from "4.4.4 Demonstration and evaluation", Figure 6 below presents a high-level view of the entire implemented research process. It uses the five activities and maps them out on a timeline. Additionally, the data collection activities are presented corresponding to when they happened on the timeline.

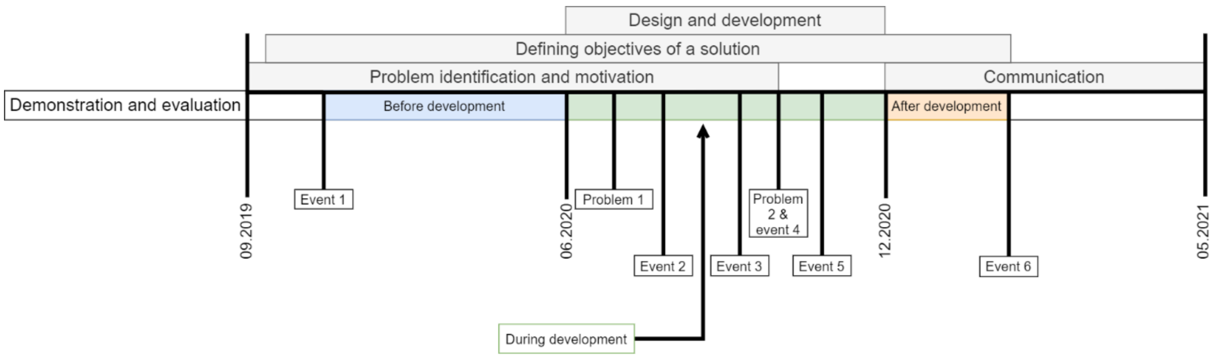


Figure 6. Implemented research process summary.

Evaluations have been presented as a vital part of measuring how well the artifact performs in solving the problem. The evaluations have been conducted in three separate phases, as seen in Figure 6 above: "before development," "during development," and "after development" of the SCP. The results of the data collected in the "before development" and "during development" phases are presented in the next chapter and have directly informed artifact construction. The results of the data collected in the "after development" phase are presented in "Chapter 6: Evaluation" and forms the basis when discussing the SCP's required improvements and final design principles. In conclusion, the five activities presented form the overall implemented research process, while the evaluations conducted in the three separate phases concern artifact construction and its capabilities.

Chapter 5: Artifact Description

This chapter describes the component management system resulting from applying the research approach. The component management system attempts to address the problem that software components created by HISP groups to aid them during localization are not reused. This chapter first presents the role of the Node Package Manager (NPM). Secondly, it describes the SCP's role within the DHIS2 ecosystem to position it by other available software development boundary resources. Third, this chapter presents collected data from the initial diagnosis of HISP groups' app development processes. Fourth, this chapter presents the SCP's construction process and the design considerations which emerged through it. Fifth, the architecture of the SCP is presented in detail. Last, initial design principles that emerged from this process are presented. The component management system constructed in this project is called the Shared Component Platform (SCP).

5.1 Node Package Manager (NPM)

Before going into detail about the SCP, it is vital first to understand the role of NPM in this context. NPM is the world's largest software registry, where developers share and use packages (About Npm | Npm Docs, n.d.). It has become the default package manager for Node.js, a JavaScript runtime environment (Node.js, n.d.). NPM handles code and metadata in the form of *packages*. These packages are directories described by a "package.json" file that includes metadata information such as the package's name, version, description, author, keywords, and dependencies. Developers within the Node.js ecosystem can publish packages to NPM and use packages other developers have published. As a result, software development processes in the Node.js ecosystem often heavily rely on using these packages. The types of packages come in all shapes and forms, from that of React, a well-known framework for creating user interfaces, to that of "atob," which simply turns base64-encoded ASCII data into binary data. The SCP relies on NPM for its infrastructure concerning these *packages*.

Relying on NPM means the unit of distribution is an NPM package with its inherent requirements. Thus, the distinction between a *package* and a *component* is presented.

- *Package*. The unit of distribution in NPM and the unit that developers include in their projects. The package acts as a wrapper for components and can include one or more components.
- *Component*. A reusable unit of software code that developers import from the package. Components within a package can be reliant on each other or completely independent.

This distinction has already been briefly discussed in "3.2.2 Central concepts in CBSE" and is a well-known approach to encapsulate components and turn them into a distributional form. With this distinction presented, packages are the unit distributed in the SCP, and they include one or more reusable components. Thus, while the SCP discusses components as the construct of most importance, these components are in reality encapsulated, handled, and distributed within packages. For the SCP, this means component producers publish a package to NPM containing one or more components. Component users discover components through SCP's interfaces, then reuse components through the package they are distributed within from NPM.

5.2 SCP's role within the DHIS2 ecosystem

The SCP's primary role within the DHIS2 ecosystem is to enable software component reuse by facilitating software components between its HISP groups. The DHIS2 ecosystem already consists of a wide array of available boundary resources that provide functionality. The DHIS2 Core Team is responsible for developing the platform, its stable interfaces and core, and developing a set of bundled apps that come with the platform. These bundled apps allow the end-users to perform certain activities on the platform, such as data entry. The HISP groups are professional implementation groups that specialize in localizing the DHIS2 platform. When extending the platform with apps, they rely on software development boundary resources discussed in "3.1.3 Boundary Resources" to aid them. Most of these are provided by the DHIS2 Core Team. One example is the DHIS2 APIs used for extending the platform. Another example is the DHIS2-ui component library, which provides components for reuse. These make the app development process less resource-intensive by automating parts of the process or allowing the reuse of resources. Additionally, the HISP groups rely on self-resourced initiatives, such as component libraries, to aid them in their development process. Table 19 below presents an overview of boundary resources in the DHIS2 ecosystem, as discussed in Msiska et al. (2019, p. 7-8).

Category	Resources
Software development boundary resources	DHIS2 APIs, Libraries, SDKs, and the SCP
Capacity building boundary resources	DHIS2 Academies, DHIS2 Manuals, Facilitators

Table 19. DHIS2 boundary resources adapted from Msiska et al. (2019, p. 7-8).

The Shared Component Platform (SCP) is a software development boundary resource for enabling the reuse of software components created through self-resourced initiatives with other HISP groups in the DHIS2 ecosystem. As with other software development boundary resources provided by the DHIS2 Core Team, it has the role of resourcing and securing the DHIS2 ecosystem. It resources the ecosystem by enabling software component reuse and secures the ecosystem by defining standards for use, such as development frameworks. While it contains aspects that secure the ecosystem, its primary goal is to resource the ecosystem. The SCP's goal is to enable HISP groups to share their components through a standardized system so that other HISP groups can effectively reuse them. In addition, the SCP's purpose is to coordinate with other already existing tools and boundary resources during the app development processes.

As described in section "3.1 Digital platforms", transaction platforms have the central role of facilitating transactions between organizations, entities, and individuals by reducing friction in the transaction process. Therefore, the SCP can also be conceptualized as a transaction platform that facilitates the exchange of software components by the two user groups presented in section "3.2 Component-based software engineering": components producers and component users. Conceptualizing the SCP as a transaction platform allows the usage of the concept of network effects (Tiwana, 2013) to assist in describing what such a component management system needs to succeed. In this case, it inhibits positive, indirect network effects, referring to how the value for component users increases for each component producer and vice-versa. Consequently, if the platform fails to attract component producers, there is no value for component users.

One notable difference between the SCP and other transaction platforms is that there are no direct ways for component producers to earn a profit by publishing their components. DHIS2 is an open-source project where all code is publicly available, and most of the components created by HISP groups are already available for free in their repositories. Additionally, component producers and component users are not necessarily distinct user groups. One can argue that component producers will also use other component producers' components, making them component producers and component users in different scenarios. As such, the goal of supporting transactions is for the benefit of all collaborating HISP groups.

One can see similar types of transaction platforms on top of software platforms, like the DHIS2 AppHub, responsible for managing apps' distribution. In a similar sense, the SCP can be viewed as a transaction platform residing on top of the DHIS2 platform for managing component’s distribution. The main difference between the two is their purpose. The DHIS2 AppHub is used to acquire DHIS2 apps for end-use. In contrast, the SCP is used to acquire software components for creating apps for end-use. Their main similarity is that they rely on the underlying software platform, DHIS2, to provide any utility to their users.

Most transaction platforms use the possibility of earning money to attract providers and fill that side of the transaction, thereby providing more content and potentially filling the consumer side of the transaction. However, because components on the SCP are meant to be free, there is a question on if there are enough incentives for component producers to provide enough components. These incentives are discussed further in section "7.2 Enterprise software platform ecosystems".

This section has presented the SCP’s role within the DHIS2 ecosystem concerning the two conceptualizations that help position it: a software development boundary resource and a transaction platform. Its conceptualization as a software development boundary resource helps discuss the SCP with other resources in the DHIS2 ecosystem. Additionally, its conceptualization as a transaction platform provides value when discussing the transaction of components and incentives for participants to partake in that transaction. Nonetheless, since the thesis focuses on designing the artifact, it is primarily conceptualized as a component management system if not otherwise stated. Thus, it makes it easier to reason about its architecture and supporting processes with the CBSE literature.

5.3 HISP groups’ app development processes

This subchapter discusses how HISP groups do app development before describing their practices of component reuse and its related challenges. This section presents the diagnostic data collected from the two interviews conducted with HISP groups presented in section "4.4.1 Problem identification and motivation". This data forms the foundation for an understanding of how HISP groups do app development. In addition, this data has been essential in informing artifact construction and making it relevant for its intended users. The two HISP groups are presented as “group A” and “group B” for their confidentiality and are both parts of the DHIS2 ecosystem in Sub-Saharan Africa.

5.3.1 Initial configuration and requirements

Group A initiates a new development project in two ways. The first is when their partners, such as Ministries of Health or other organizations, contact them about a problem, challenge, or need they are experiencing. This problem, challenge, or need can, for example, be the need for mechanisms to handle data visualization or problems with an existing apps' presentation of a table. The partners do not specifically ask for new apps but often ask how something can be done on DHIS2, where an app is often the solution. The second way is when they find a problem where they can build a solution or make an app applicable to a problem the partner is facing and then present it to the partner.

The HISP group consists of two prominent roles, that of implementers and developers. Implementers are responsible for doing the initial planning and are the first to talk with the partners, then communicating with the developers. The initial planning includes looking to DHIS2 to identify what is available that can solve the problem. They check the "generic apps, and most times [they] also check the apps that are available, developed by other developers to see if there is something that can solve [their] problem." If the conclusion is that the problem cannot be solved using other readily available resources, the implementers try to understand precisely the partner's needs before including the developers. If the solution is a new app, they create the product backlog through brainstorming sessions and talks with the partner. During this phase, they set standards for the app and identify common components that the development team can use.

Group B also mentions requirements gathering as an essential step. They involve a business analyst in the early phases of the project to analyze the domain and assist in creating the requirements. Additionally, they say that most requirements after a project's initiation are often ad-hoc, in the sense that they need implementation as soon as possible. After the initial set of requirements, if the conclusion is to create an app, they create user stories and scenarios that are part of their behavioral-driven development approach, which allows them to connect the user stories to the app's domain.

5.3.2 Development

After acquiring the requirements for the app, group B has what they call a "seed app." This seed app provides the basic architecture and components used across most apps. They use it as a starting point for further development as it is preconfigured for integration with DHIS2. Group A also has a "skeleton app," similar to that of group B's. For group B, this app relies on their component library, which contains all their components. This component library is also available for other developers on NPM. Additionally, the app has the setup to support both development and production environments. For group A, their app implements the component's source code explicitly.

Group A planned to create their internal component library but did not go ahead with the plan as they "are not sure it is really needed for [them] as an organization." Additionally, they explain that they "quickly can reuse the components using the source code" as it works well for them. Moreover, they have not met any significant barriers regarding the maintenance of these components so far. They are happy with the level of configuration it gives, as the source code of the component in each project can be individually updated. Both groups then build on top of this skeleton app with the requirements found during the previous phase during cycles of agile development based on the product backlog.

5.3.3 Testing and handover

Both groups mention testing as an essential process towards the end of development. Group A explains that the developers perform verification tests "to guarantee that everything was supposed to be done was done" before handing it over to the implementers to do evaluation tests "to make sure that was done, was done properly." Group B also uses automated tests such as integration tests to make sure everything is running as intended.

5.3.4 Software component reuse

For group B, updating the component library happens ad-hoc. They do not have any specific schedules but rather add or update components when requirements change. Depending on their activities, components can be added or updated, and deployed multiple times a week. During this deployment process, they have created automated pipelines for testing. Furthermore, they have also planned to add automatic deployment, but this is currently done manually. Additionally, to ensure that the quality is checked and the tests passed, they use pre-commit checkers before pushing the code to source control. On NPM, the packages with these components are all stored under a singular "scope." An NPM scope is a way to organize packages under a namespace. There, members from the development team have been given the necessary access right to that scope so that different members can add and update components. They do not require permission from one specific person, as it is "not convenient waiting for someone to just publish a package."

Group B mentions lack of diversity as a significant barrier to their component reuse, as their development team, for the most part, has been using a development framework incompatible with the one that the DHIS2 Core Team is standardizing. Since more of the ecosystem follows the DHIS2 Core Team, there are fewer components in other development frameworks. One member of group B mentioned: "In fact, we have not really found many components from other HISP groups," indicating that not only the mismatch of development frameworks is a barrier, but also the component's discoverability.

Group A also developed in a different framework but decided to start transitioning all their apps to the one standardized. Their reason is to take part in the benefits of the software development boundary resources the DHIS2 Core Team provides—for example, a tool for bootstrapping apps and a component library. The decision was also based on bugs and quick version changes in the other framework, making apps hard to maintain. For group B, some developers on the team have taken the initiative to learn the standardized framework independently, but they do not have any specific plans to transition yet. They explain that they already have many resources and components, and to serve customers quickly, they need to rely on those resources. Therefore, there will be downtime in trying to transition, which makes it harder to justify. As a workaround, they often manually port components created by the DHSI2 Core Team over to their development framework.

Both groups engage in finding components other developers have produced during the development process, although it is not always easy. When asked what could be improved in the component reuse process, group B mentioned two things. The first was "the discoverability of the components, in a sense that it is easier to search for the components," and the second "but also you may discover the components, but probably it may not be useful as such, as it probably may lack appropriate documentation on how best to use the component." Group A have good experiences with some components from the DHIS2 Core Team, while the experience has been worse with others based on lack of documentation. Group B also mentions the lack of documentation as a barrier for component reuse. Because of this, they attempt to include a "Readme" file for each component they create for their component library, which describes how the component should be used. In addition, group A mentions how it is crucial to stay in touch with the DHIS2 Core Team when using their components to know if the components will have updated documentation or versions. Indicating that good communication between the component producer and the component user is essential for trust in component reuse.

Both HISP groups are experienced with the component reuse process from this initial diagnosis but meet multiple challenges. Group B mentions the “discoverability of the components” and the subsequent fact that they “may lack appropriate documentation” on their use as challenges. Since they are developing in another development framework, they add: “In fact, we have not really found many components from other HISP groups.” Indicating that discoverability and documentation are two of the significant challenges met by group B. Group A mentions how they have to check the documentation and the number of downloads to find suitable and stable components. Additionally, the well-documented components from the DHIS2 Core Team accelerated their app development process. In summary, it indicates that documentation, the number of downloads, and other similar metrics for quality assurance are essential for reuse. Table 20 below summarizes the challenges found in the component reuse process and their description. It is comparable to the barriers in the component reuse process presented by B et al. (2010) in section "3.2 Component-based software engineering", like poor cataloging and distribution and lack of certification metrics to determine quality.

Challenges	Description
Discoverability of components	The challenge of searching for and finding components.
Quality assurance of components	The challenge of determining if a component is stable and suitable through documentation and other metrics.
Diversity in the ecosystem	The challenge of finding components in a matching development framework.

Table 20. Challenges in the component reuse process.

5.4 SCP’s construction process

This subchapter discusses decisions that were important throughout the development process of the SCP. The data presented in this section is from the events presented in section "4.4.2 Defining the objectives of a solution" with the DHIS2 Core Team, HISP groups, and the project supervisors, except for the expert evaluation. This section discusses the results from these events structured concerning central parts of the SCP. Table 21 below reiterates these events and presents their phase concerning the development process, as discussed in section "4.4.4 Demonstration and evaluation".

Event	Description	Phase
1	Initial focus group with three members of the DHIS2 Core Team, a member of a HISP group, and project supervisors. Follow-ups on mail afterward.	Before development
2	Focus group with project supervisors.	During development

3	Focus group with members of the DHIS2 Core Team.	During development
4	Focus group with members of a HISP group.	During development
5	Informal talk with a member of the DHIS2 Core Team	During development

Table 21. The events from section "4.4.2 Defining the objectives of a solution."

5.4.1 Before development

The initial focus group (event 1) lay the foundation of both the goals of the SCP and its envisioned architecture. This section presents the foundation laid from this event and the decisions reached by the research team.

During the initial brainstorming in event 1 there was a consensus that component users needed a "main page" to discover the components in the SCP. The proposals that arose were a simple list with package names or a more extensive website with a gallery. It was decided that this would come in the form of a website, as it is more user-friendly and allows further extension. It was also agreed that there should be mechanisms to promote components that had passed some certification so that there would be incentives to publish well-crafted components. To not unnecessarily gatekeep contributions, we reached the distinction between uncertified and certified components, which would be handled differently by the system. No direct criteria for certification were created during this event, but a DHIS2 Core Team member proposed the idea of a whitelist. After performing certification on a component, its identifier would be entered in a whitelist, indicating its certification. Although a whitelist was discussed, it was not decided at this point, as other options such as collecting components under a shared namespace were also attractive. The gallery in the website would then somehow promote the components that had passed certification.

It was during this event decided that the SCP would build on top of the Node Package Manager (NPM) for two reasons: it provides extensive infrastructure for hosting and distributing packages, and all developers in the Node.js ecosystem are already using it. However, the distinction between packages and components discussed in section "5.1 Node Package Manager (NPM)" was not reached yet. Therefore, at this point, the decision was to follow a "one component, one package" philosophy, where component producers would have to encapsulate one individual component in one package and then publish it. This philosophy would allow the SCP to catalog and display components easily. From the research team's understanding at this point in the research process, this seemed like a good idea, as we did not see any other way around it. However, this early philosophy brought some confusion regarding the definition of a component and a package as they were used interchangeably.

Then, the question was how the SCP would find the packages published to NPM, which spurred three main ideas from a DHIS2 Core Team member. First, component producers use the name "dhis2-ui-contrib-<package name>" for consistency. Second, they add a keyword such as "dhis2-ui" to their packages. Third, the DHIS2 Core Team set up a package under "@dhis2/ui-contrib," where they add components to a centralized repository. The same DHIS2 Core Team member describes that the first and second "shifts control to the community" as they can freely decide on their process without interference from the DHIS2 Core Team. In contrast, the third means the DHIS2 Core Team "would gatekeep and quality control the contributions" in an officially stamped package. Which of these options would be the final one was not explicitly decided before development. Lastly, a command-line interface (CLI) was discussed so that component producers can bootstrap new packages with all integration requirements already inputted.

To summarize, the “before development” phase created the initial objectives of the solution. The architecture would consist of a website for discovering components and a whitelist for components that had gone through a certification process. Additionally, the architecture would be built on top of the infrastructure provided by NPM. This meant component producers would publish their components in packages to NPM following a standard that allows the SCP to find them afterward. Lastly, a CLI for component producers to bootstrap new packages were discussed.

5.4.2 During development

After the previous phase had laid the foundation for the goals of the SCP, the development of the artifact started. This section presents the design decisions reached in the “during development” phase. It uses the events presented in Table 21 above and relates them to the then two significant parts of the process, the website and certification.

5.4.2.1 Website

After the foundation laid in the previous phase, design and development started on the website as a gallery to display components. Since we had decided to follow the “one package, one component” philosophy, this was a simple list with the package name and description from NPM. Thus, component users could input a package name, and the website would search the NPM registry for matches. After a demonstration to the project supervisors in event 2, it received feedback on its utility for HISP groups if it only inhibited the same functionality as NPM. After this, the search was enhanced by adding NPM-specific filters such as keywords and namespace. Additionally, the functionality to filter on certified and uncertified packages was added, even though tools to handle the certification were not yet created.

Event 3 discussed much concerning certification and the “one package, one component” philosophy. NPM packages use a “keyword” attribute in the ‘package.json’ file for its inherent sorting and filtering, so it was agreed during this event that the SCP would do the same. Therefore, the first requirement for packages was to include a specified keyword for the SCP to find them. Additionally, a DHIS2 Core Team member mentioned that it is essential that everyone agrees on the value of this keyword as it is not desirable to change after the SCP is finished, as this would require all packages to change. However, the keyword value itself did not get specified during this event. Lastly, three different options for handling certification were presented to the members of the DHIS2 Core Team. The decision for certification was on a whitelist stored in GitHub. The options are discussed further in the whitelist section below.

The project had so far followed the “one package, one component” philosophy. During event 3, a DHIS2 Core Team member brought up that often multiple components are collected in one package – a component library – and forcing component producers to adapt to the philosophy mentioned above from the SCP might hurt its adoption. Therefore, for the SCP to be most helpful, it would be good to find “some ways to expand the library and expose the individual components.” Another DHIS2 Core Team member suggests basing this on a file that lists a package’s available components. How to construct this list was not brought up, but they thought it could be performed somewhat automatically.

After this event, the research team explored this option and figured that the ‘package.json’ file, which NPM requires for all packages, is extensible with new properties. For component producers to publish components, they need to fill out an attribute called “dhis2ComponentSearch” with the components in their package. It contains information on the package, such as its language and the components with their name, export, description, and DHIS2 version compatibility. The research team reached this specification by going through all the information relevant to gather and display about the package and the components. Section "5.5.1 Package Requirements" further describes this specification. Both the requirement for keyword and “dhis2ComponentSearch” attribute specification was presented to a HISP group in event 4. When asked about defining the components manually, the response from a HISP group member was, “Yeah, I think that is pretty easy to do.”

During event 3, the DHIS2 Core Team had proposed many improvements for the SCP. Afterward, the research team had to explore options and find a way to implement these improvements. First, the current options for search – all of NPM, within a keyword, within namespace – were changed to search within the “dhis2-component-search” keyword. Additionally, the website would fetch these on initialization. The research team decided on the keyword because we wanted a keyword with no preexisting packages for testing. Using such a keyword, component producers could publish their package to NPM and have it appear without directly interacting with the SCP, as long as the “dhis2ComponentSearch” attribute was correctly structured. Next, functionality for the website to fetch the whitelist containing package identifiers and versions from GitHub was implemented.

The website had so far handled components and packages interchangeably. The new distinction between packages and components required an extensive rework of the functionality and structure of the website, including the search result, the search itself, and the filters. Previously, the search term and filters were sent to the NPM registry, which performed the search and returned a set of packages, which could directly be displayed in the gallery. A new layer was added to handle the new package, and component distinction: after packages incorporating the ‘dhis2-component-search’ keyword are fetched during initialization, it fetches the ‘package.json’ file for each package. Thus, it allows the extraction of components from the “dhis2ComponentSearch” property. This collection of the components is then subsequentially used by the website to display, search, and filter components. After this collection was made, all other previously made functionality for packages was refactored to work for this component collection.

To finalize the project sometime later, the research team took a critical view of the website and created the last set of requirements. The requirements mainly consisted of minor functionality such as filters and design requirements; rearranging items to make the interface more intuitive and understandable. Up until this point, the previous packages and now components were displayed in a simple list. Finally, a gallery-type setup of cards displayed in a grid was implemented, where each component had its card describing all its information.

To summarize, the website has been created and improved based on feedback during the development and evaluation process. The initial version displayed a list of packages, where the finalized version extracts the components from the packages and displays them in a gallery. This section has presented the design decisions that have guided the design and development process. The finalized version of the website is described in section "5.5.2 SCP Website".

5.4.2.2 Certification

The distinction between uncertified and certified packages was set during event 1. Uncertified packages could be published to be picked up by the SCP without interaction or oversight by anyone. Certified packages would go through a certification process handled by the DHIS2 Core Team before being marked as certified. A CLI was created to handle the certification process. It would check if the package had correctly defined the keyword and incorporated no errors when run on a package.

During event 3, three options to handle the outcome of certified packages were presented to the DHIS2 Core Team. The first option was to duplicate it to a centralized contribution package created and maintained by the DHIS2 Core Team. One centralized package requires component users to download the entire package, as it bundles components together. The second option was to duplicate the packages to a centralized repository created and maintained by the DHIS2 Core Team, then publish each in their respective package. The third option was a whitelist, where all HISP groups would continue to host packages in their repositories, and then a list with their identifiers and versions is created. The DHIS2 Core Team found the first two options to be undesirable. It would require “a lot of work and maintenance from a management team or a core team,” as they would be responsible for handling the duplicated packages and their republishing under the official DHIS2 brand. They liked the idea of the third option and referred to it as “much more community-based,” as the packages would continue to be hosted and published separately from any interaction by the DHIS2 Core Team. The DHIS2 Core Team would then only be responsible for cultivating this list of packages to promote packages with a certain level of quality, maturity, and testing. Therefore, they conclude that the whitelist is “a list of the third-party NPM packages that are verified or have been checked out by the core team and seem to have a good level of quality.” The plans for a whitelist were brought up and presented to a HISP group in event 4. The whitelist itself did not get many comments, but the requirements for certification seemed to be a topic of importance. However, the certification requirements were not yet specified.

After these events, the concepts of hard certification and soft certification emerged during internal discussions with my colleagues as ways to view the certification requirements. Hard certification was defined as all requirements vital for the package to be accepted by the SCP. These are all checks that can be done entirely automatically, i.e., checking the ‘dhisComponentSearch’ specification. On the other hand, soft certification was defined as those hard to automate and instead need to be checked manually. Soft certification is, for example, testing the package in a project or reading its documentation. After informal talks with the DHIS2 Core Team on Slack and an informal meeting in event 5, both types of certification were concluded as essential in their own right. However, hard certification should be the focus since those are vital for the SCP’s functionality.

The SCP CLI was improved to handle as many of these requirements as possible. The SCP CLI’s primary functionality included certifying the hard certification requirements, such as the keyword and the “dhis2ComponentSearch”. Additionally, a requirement that the source code of the packages needed to be stored in a GitHub repository was added so that the SCP CLI could clone the source code to apply some of the soft certification requirements. So far, the soft requirements included running an ‘eslint’ test to check for errors in the code and an ‘npm audit’ test to check that the package is a valid NPM package, present vulnerabilities, and possible dependency updates.

The SCP Whitelist was created to test these requirements in a distributed location. It was designed as a GitHub repository with a comma-separated values (CSV) file to handle package identifiers and versions. For component producers to submit their package for certification, they create a ‘pull request’ by proposing a change to the CSV file. The pull request triggers an automatic pipeline that uses the SCP CLI to run the certification requirements. Then the result needs to be checked manually by a certifier to decide if it should be included or not. Therefore, the SCP CLI has a twofold role of performing the certification checks in the SCP Whitelist but can also be used directly by a component producer to perform the same checks locally in their environment before submitting the package.

To summarize, this section has presented the design decisions that have guided the design and development process. The finalized version of the SCP CLI is described in the section "5.5.3 SCP CLI" and the final version of the SCP Whitelist in the section "5.5.4 SCP Whitelist". Table 22 below summarizes the important aspects covered in each of the implemented research process’ activities. The following section presents the finalized SCP and its architecture.

Activity	Summary
Problem identification and motivation	The research problem is defined as software components created by HISP groups to aid them during localization are not reused. Discoverability of components, quality assurance of components, and diversity in the ecosystem are identified as barriers for HISP groups' potentially successful reuse.
Defining the objectives of a solution	A component management system that allows HISP groups to publish and reuse components and the DHIS2 Core Team to certify components. High-level requirements that attempt to address the barriers in the previous activity and the research problem.
Design and development	Creating the SCP containing the SCP Website, SCP Whitelist, and SCP CLI to implement the high-level requirements from the previous activity.
Demonstration and evaluation	The “before development” and “during development” evaluations have fed back into the “problem identification and motivation” and the “defining the objectives of a solution” activities. Results from the “after development” evaluation are presented later in “Chapter 6: Evaluation”.
Communication	Initial design principles are presented later in "5.6 Initial design principles". Final design principles are presented later in "6.7 Final design principles".

Table 22. The implemented research process summarized so far.

5.5 SCP's architecture

Now that the role of NPM, the SCP's role within the DHSI2 ecosystem, and the design processes to build it has been presented, this subchapter presents the technical architecture of the SCP. The SCP is a component management system for collecting and sharing software components created by HISP groups with other HISP groups in the DHIS2 ecosystem, thus enabling their reuse. The architecture is created to support three processes shown in Table 23 below.

Actor	Process description
Component producer	Publishes packages to NPM containing components. Additionally, submits packages containing components for certification.
Component user	Discovers components through SCP's interfaces then reuses components through the package they are distributed within from NPM.
Certifier	Accepts or declines the packages submitted for certification.

Table 23. Supported actors and their processes.

To support the component producers, component users, and certifiers in their processes, SCP implements three primary modules. Table 24 below presents the three modules and their primary function(s).

Module	Primary function(s)
SCP Website	Extracts components from their packages and provides an interface for component users to discover these components and reuse them through the packages that contain them.
SCP Whitelist	Contains a list of certified packages submitted by component producers and accepted by certifiers.
SCP CLI	Allows a component producer to pre-certify a package before they submit it for certification. Additionally, used by the SCP Whitelist.

Table 24. SCP module overview.

This subchapter first presents the package requirements needed for integration with the SCP before presenting each module in detail.

5.5.1 Package requirements

As mentioned in section "5.4.2.1 Website", specific requirements are needed in the package for it to work correctly with the SCP. These requirements need to be integrated into the package before the component producer publishes it to NPM. These need to be defined in the previously mentioned 'package.json' file from section "5.1 Node Package Manager (NPM)", which NPM requires all packages to include. The 'package.json' file is a JSON file describing the package that includes it. It describes the package's name, description, dependencies, authors, repository, and similar properties. First, the 'package.json' file needs to have the 'dhis2-component-search' keyword defined, seen in Figure 7 below. Second, the 'package.json' file needs to have a repository property, defined through key/value pairs for repository type and URL, seen in Figure 8 below. This URL must use the HTTPS protocol and not SSH, which causes errors in the SCP Whitelist pipelines. Currently, only GitHub repositories are supported for certification because of these requirements.

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

Figure 7. Package requirement keyword.

```
"repository": {
    "type": "git",
    "url": "https://github.com/<username>/<repository>.git"
}
```

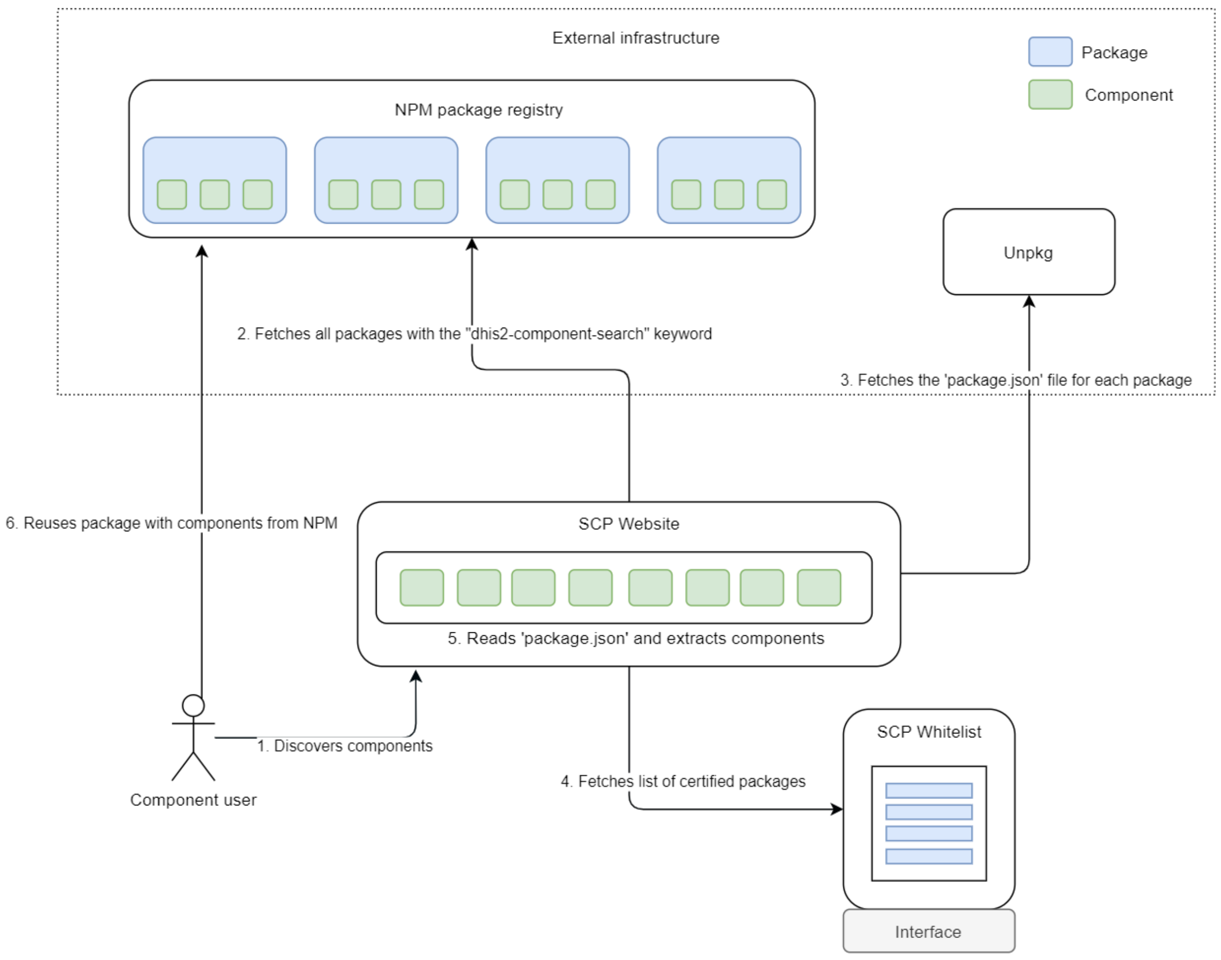
Figure 8. Package requirement repository.

The component abstraction needs to be specified to expose the components within a package. The abstraction is constructed manually by the component producer and placed in the ‘package.json’ file within the ‘dhis2ComponentSearch’ property. On the top level, this is an object containing two properties: ‘language’ and ‘components.’ The language field is a string describing the language of the components, such as ‘angular’ or ‘react.’ The components field is an array of components represented in the form of objects. These objects have three required fields and one optional. The required are ‘name,’ ‘export,’ and ‘description.’ The property ‘name’ is a string and is a descriptive name of the component. The property ‘export’ is the specific variable exported in the code, while ‘description’ is a human-readable description of the component. The optional is ‘dhis2Version’ and is an array containing strings with DHIS2 version numbers, specifying where this component is tested and confirmed to work. The research team specified this standard, as described in section "5.4.2.1 Website". Figure 9 below shows an example of a package containing two components.

```
"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"
     ]
   },
   {
     "name": "Covid-19 Visualizer",
     "export": "Covid19Visualizer",
     "description": "A visualizer for covid-19 data",
     "dhis2Version": [
       "33.0.0"
     ]
   }
 ]
}
```

Figure 9. Package requirements dhis2ComponentSearch property.

5.5.2 SCP Website



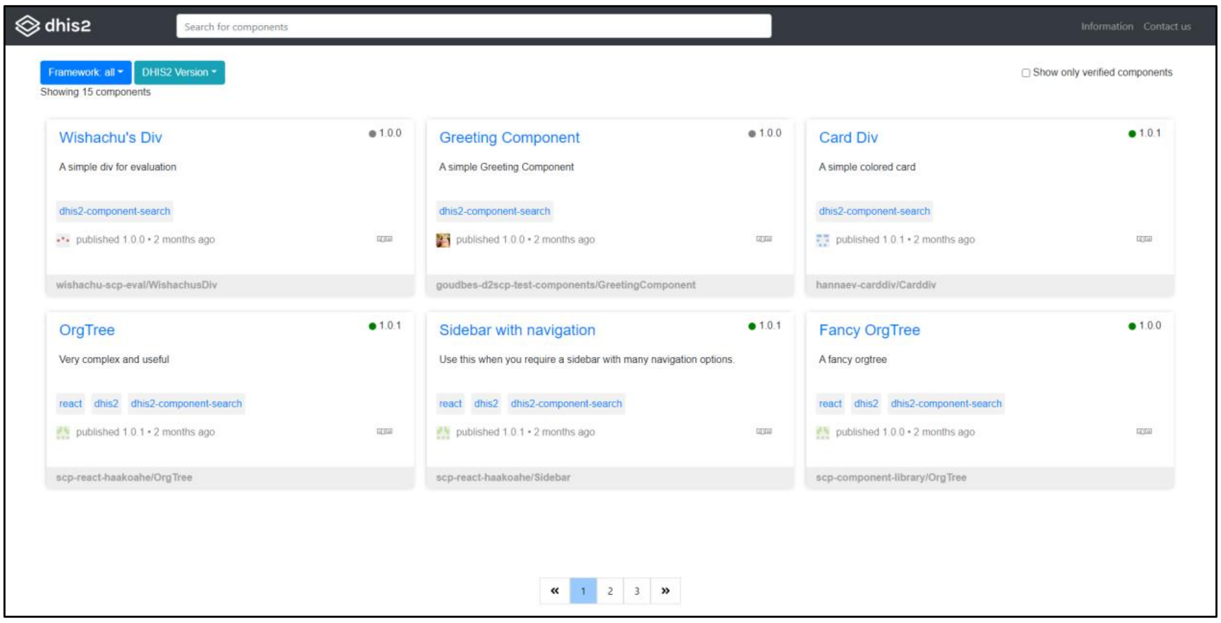
External infrastructure
Package
Component
NPM package registry
Unpkg
2. Fetches all packages with the "dhis2-component-search" keyword
3. Fetches the 'package.json' file for each package
6. Reuses package with components from NPM
SCP Website
5. Reads 'package.json' and extracts components
1. Discovers components
Component user
SCP Whitelist
4. Fetches list of certified packages
Interface

Figure 10. Website architecture and supported process.

The SCP Website is a web application written in React that provides the interface by which component users view all components stored in the NPM package registry by extracting them and applying the whitelist. As such, the website is the module that component users interact with during app development. Figure 10 above shows the process initiated when a component user (1) wants to discover components on the SCP Website. The SCP website then (2-5) fetches all packages, extracts their components, and displays the components to the component user. Finally, the process ends when the component user (6) determines the component they want to reuse and reuses the package with the component from NPM.

The website provides an interface consisting of a navigation bar, search field, information, and a gallery of components cards. Using the search field, filters, and pagination, the component users can navigate through all components cataloged by the system. The navigation bar and its functionality are identical on all the website’s pages. The “dhis2” logo always sends the user to the landing page while searching in the search field will always bring the user to the landing page with the search term conducted.

5.5.2.1 Pages



Showing 15 components

Wishachu's Div ● 1.0.0
A simple div for evaluation
dhis2-component-search
published 1.0.0 • 2 months ago
wishachu-scp-eval/WishachusDiv

Greeting Component ● 1.0.0
A simple Greeting Component
dhis2-component-search
published 1.0.0 • 2 months ago
goudbes-d2scp-test-components/GreetingComponent

Card Div ● 1.0.1
A simple colored card
dhis2-component-search
published 1.0.1 • 2 months ago
hannaev-carddiv/Carddiv

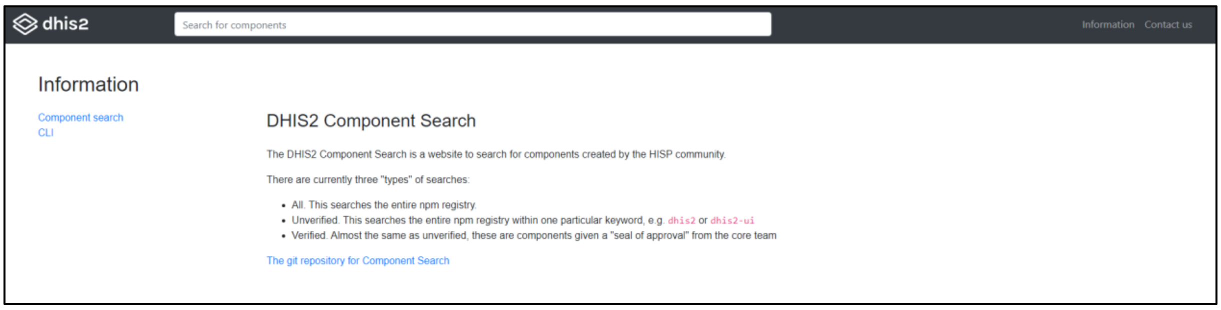
OrgTree ● 1.0.1
Very complex and useful
react dhis2 dhis2-component-search
published 1.0.1 • 2 months ago
scp-react-haakoahe/OrgTree

Sidebar with navigation ● 1.0.1
Use this when you require a sidebar with many navigation options.
react dhis2 dhis2-component-search
published 1.0.1 • 2 months ago
scp-react-haakoahe/Sidebar

Fancy OrgTree ● 1.0.0
A fancy orgtree
react dhis2 dhis2-component-search
published 1.0.0 • 2 months ago
scp-component-library/OrgTree

Figure 11. SCP Website landing page

The website consists of three pages. The first is the landing page, seen in Figure 11 above, which contains all functionality related to the display, search, and filtering of components. The contents of the component cards are presented below in section "5.5.2.2 Component Gallery". Then there is an "Information" page containing information about each of three modules: the SCP Website, the SCP CLI, and the SCP Whitelist, seen in Figure 12 below. Lastly, a "Contact us" page with information on the DHIS2 Design Lab and HISP, seen in Figure 13 below.



Information

DHIS2 Component Search

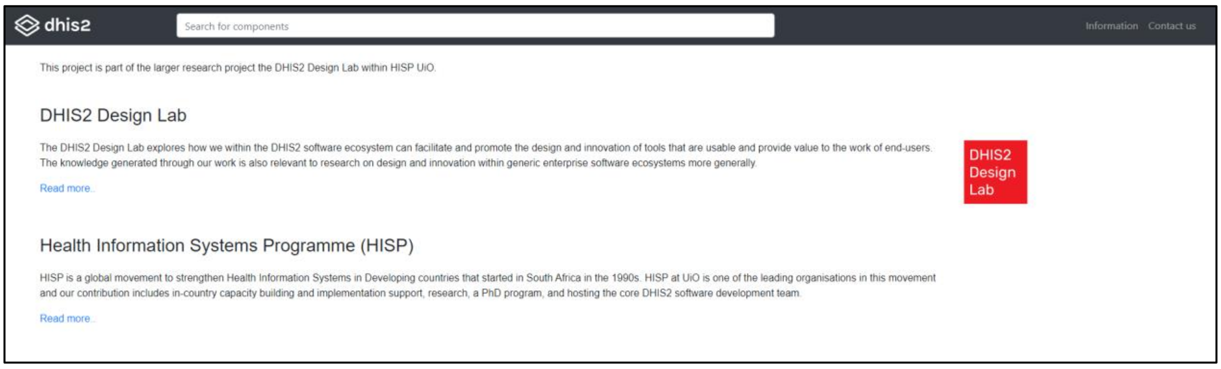
The DHIS2 Component Search is a website to search for components created by the HISP community.

There are currently three "types" of searches:

- All. This searches the entire npm registry.
- Unverified. This searches the entire npm registry within one particular keyword, e.g. `dhis2` or `dhis2-ui`
- Verified. Almost the same as unverified, these are components given a "seal of approval" from the core team

The git repository for Component Search

Figure 12. SCP Website information page



This project is part of the larger research project the DHIS2 Design Lab within HISP UiO.

DHIS2 Design Lab

The DHIS2 Design Lab explores how we within the DHIS2 software ecosystem can facilitate and promote the design and innovation of tools that are usable and provide value to the work of end-users. The knowledge generated through our work is also relevant to research on design and innovation within generic enterprise software ecosystems more generally.

Read more..

Health Information Systems Programme (HISP)

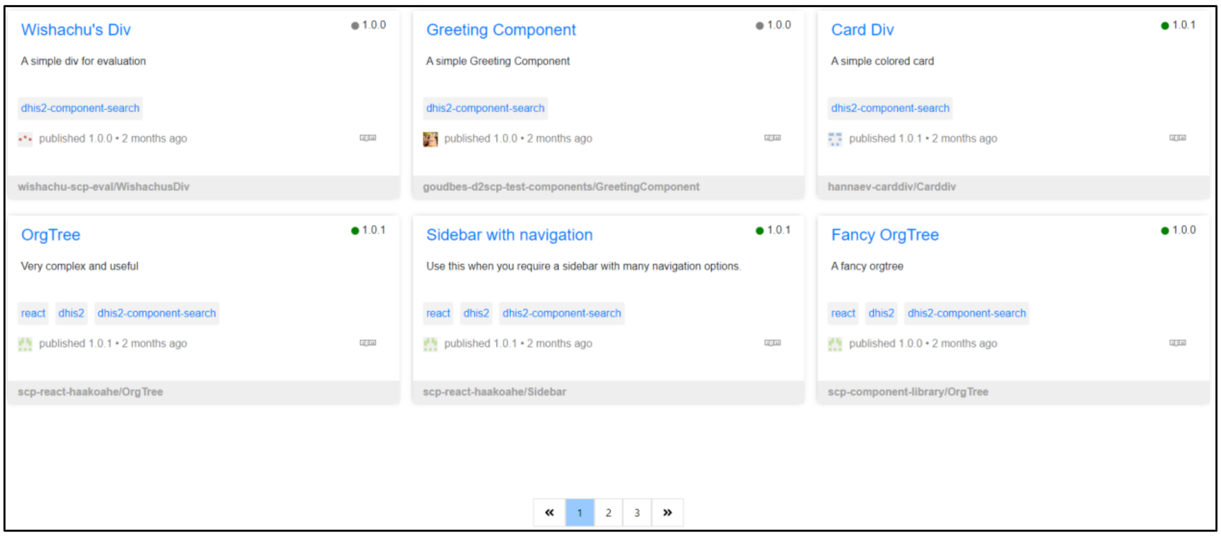
HISP is a global movement to strengthen Health Information Systems in Developing countries that started in South Africa in the 1990s. HISP at UiO is one of the leading organisations in this movement and our contribution includes in-country capacity building and implementation support, research, a PhD program, and hosting the core DHIS2 software development team.

Read more..

Figure 13. SCP Website contact us page

5.5.2.2 Component gallery

The component gallery is where all filtered and searched components are displayed to the component user, as seen in Figure 14 below. It consists of a grid of boxes that can be referred to as “cards,” as such, the component gallery consists of component cards, consisting of the information about the component. For navigation, the component gallery also consists of pagination. The pagination is based on the number of components and provides a way to navigate by going to the next, previous, or a specific page.


Wishachu's Div ● 1.0.0
A simple div for evaluation
dhis2-component-search
published 1.0.0 • 2 months ago
wishachu-scp-eval/WishachusDiv

Greeting Component ● 1.0.0
A simple Greeting Component
dhis2-component-search
published 1.0.0 • 2 months ago
goudbes-d2scp-test-components/GreetingComponent

Card Div ● 1.0.1
A simple colored card
dhis2-component-search
published 1.0.1 • 2 months ago
hannaev-carddiv/Carddiv

OrgTree ● 1.0.1
Very complex and useful
react dhis2 dhis2-component-search
published 1.0.1 • 2 months ago
scp-react-haakoahe/OrgTree

Sidebar with navigation ● 1.0.1
Use this when you require a sidebar with many navigation options.
react dhis2 dhis2-component-search
published 1.0.1 • 2 months ago
scp-react-haakoahe/Sidebar

Fancy OrgTree ● 1.0.0
A fancy orgtree
react dhis2 dhis2-component-search
published 1.0.0 • 2 months ago
scp-component-library/OrgTree

Figure 14. SCP Website component gallery

Each component card consists of the following information, as shown in Figure 15 below.

1. *The component’s name*. Name of the component as described in the component abstraction.
2. *The component’s description.* Description of the component as described in the component abstraction.
3. *The component’s keywords.* These components are published and distributed in the form of NPM *packages*, and these packages contain keywords to describe them. The “dhis2-component-search” keyword must always be included, and others will also be displayed here.
4. *The component’s producer.* The creator and producer of the component together with the version of the package and its publication date.
5. *The package of the component and the component export.* This part is meant for overview purposes, to specify the package and component in combination in this format: <package>/<component export>. This attempts to illustrate how a component producer would use the component during app development.
6. *The component’s certification.* As mentioned above, a component can have different indications based upon what version of the package is certified. A grey circle and a version number represent that this version is not certified, while a green circle and a version number represent that it is certified. They are placed in chronological order, so an earlier version number will always be underneath a newer one. Additionally, if the newest version is certified, only that version is shown with a green circle. If the newest version is not certified, both will show, where the green circle and version number represent the newest certified version.
7. *The component’s NPM link.* A link to the package hosted on NPM. This link is also embedded in the component's name.

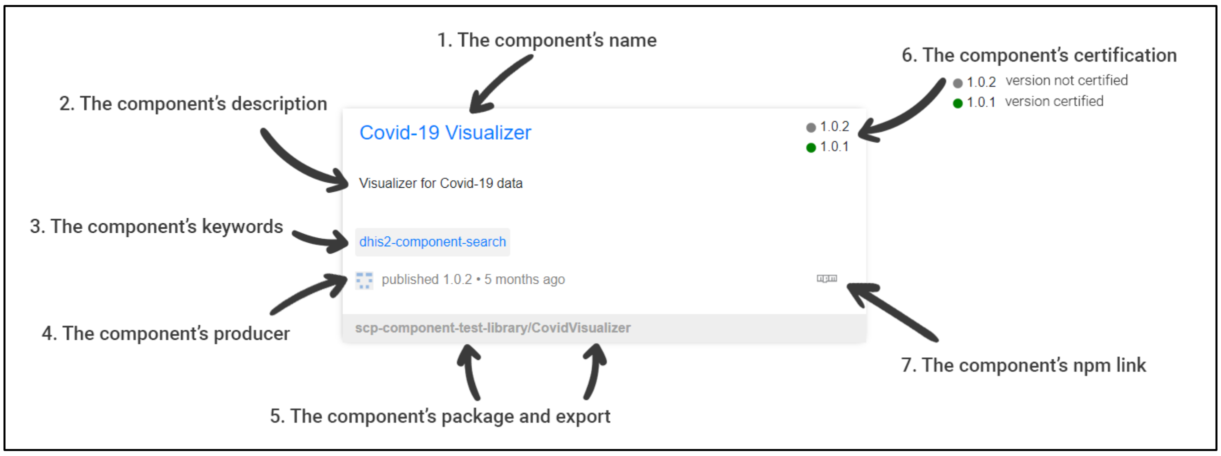

1. The component's name
2. The component's description
3. The component's keywords
4. The component's producer
5. The component's package and export
6. The component's certification
● 1.0.2 version not certified
● 1.0.1 version certified
7. The component's npm link
Covid-19 Visualizer
Visualizer for Covid-19 data
dhis2-component-search
published 1.0.2 • 5 months ago
scp-component-test-library/CovidVisualizer
● 1.0.2
● 1.0.1
npm

Figure 15. Component gallery card

5.5.2.3 Dependencies

This section describes the website's dependencies. Since the website is a React application, it is within the Node.js ecosystem and uses other libraries as dependencies from within the ecosystem. The following list is an extract of its dependencies, focusing on the most important ones:

- *React.* React is a library for building user interfaces and websites (*React – A JavaScript Library for Building User Interfaces*, n.d.). React makes it easier to build interactive user interfaces in a declarative and component-based way. The website is built using React as its overarching framework.
- *Redux.* Redux is a state management library (*Redux - A Predictable State Container for JavaScript Apps. | Redux*, n.d.). Redux provides means to centralize the website's stored state more predictably. Redux is used in combination with React to provide a centralized data store that the React components can access regardless of place in the component hierarchy.
- *React-Bootstrap.* React-Bootstrap is the Bootstrap front-end framework rebuilt for React (*React-Bootstrap*, n.d.). This library provides components that are used to design and develop the website.
- *Pure.css.* Pure.css is a set of small responsive CSS modules (*Pure*, n.d.). The website uses Pure Grids, which provides a set of easy-to-work-with properties for constructing user interfaces.
- *Fuse.js.* Fuse.js is a lightweight fuzzy-search library with no external dependencies (*What Is Fuse.Js? | Fuse.Js*, n.d.). It is a lightweight JavaScript library that provides means for handling effective searching. It is explained that "fuzzy-search" refers to the technique of approximate string matching, which finds strings that are approximately equal to the search term, rather than precisely as written. In addition, Fuse.js provides an extensive toolset for implementing the component search.

5.5.2.4 External dependencies

The website relies on a few external dependencies to provide its functionality. These dependencies are accessed through their APIs and provide the website with its content regarding the components. Strictly how these are used is described in the "5.5.2.7 Initialization" section. The dependencies are as follows:

- *NPM.* The SCP uses NPM as its registry, or database, for accessing the stored components. The website uses the NPM registry's APIs to fetch all packages with the "dhis2-component-search" keyword.
- *UNPKG.* UNPKG is a content delivery network for packages on NPM (*UNPKG*, n.d.). The website uses the UNPKG API to fetch the "package.json" information file for all packages fetched from the NPM registry.
- *The SCP Whitelist.* The website fetches the list of certified packages from the SCP Whitelist repository in GitHub.

5.5.2.5 Filters

The filters section of the website consists of three elements: framework, DHIS2 version, and certification. Selecting any attributes of these filters scans all the components and displays those matching the filters. Filters are essential in such a system as it allows the user to discover components that match their development requirements and environment.

- *Framework*. The framework filter allows the component user to filter between different development frameworks, such as React or Angular, to find components matching their development environment, as seen in Figure 16 below. The filter consists of a dropdown button that opens with three options, All, React, and Angular. Selecting 'All' displays all components, selecting 'React' displays only React components, and selecting 'Angular' displays only Angular components.

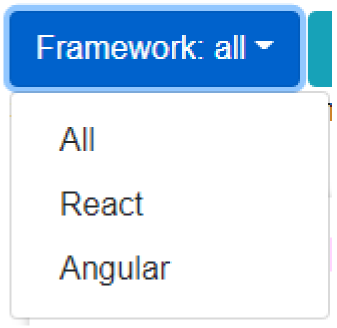


Figure 16. Framework filter

- *DHIS2 Version*. As many components in the SCP can be directly integrated towards the DHIS2 interfaces, the DHIS2 Version filter allows component users to filter what versions of the platform are supported by the component, as seen in Figure 17 below. An example of such an integrated component is the already existing "Headerbar" developed by the DHIS2 Core Team. This filter consists of a search field and a list of versions. The filter is multi-select, meaning the user can select multiple versions, which means components matching one of those versions are displayed. Additionally, the search field allows the component user to filter on the versions. The displayed versions are fetched directly from the components in the NPM registry, so only versions with a component are shown.

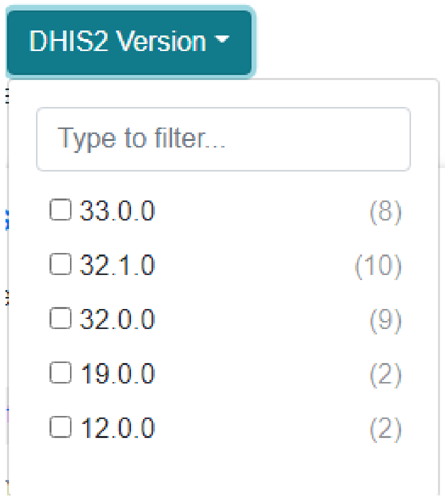


Figure 17. DHIS2 Version filter

- *Certification.* The last filter is on certification, or the verification as it is specified here, as seen in Figure 18 below. When selected, only components that have gone through the process of certification are displayed. These are the components that can be found in the SCP Whitelist repository. When checked, this filter only displays those components with their latest version certified.

Figure 18. Certified filter

5.5.2.6 Search Field

The search field iterates through all components and finds components with a "name" property that matches the search term. The search functionality is built on top of the library Fuse.js, which is already mentioned in dependencies. Additionally, if filters are already checked, the search functionality returns only those already matching the filters.

5.5.2.7 Initialization

When a component user opens the website, it starts the initialization process that displays the components. This process can be separated into the four following steps:

1. The website fetches all packages from the NPM package registry with the ‘dhis2-component-search’ keyword.
2. The data returned for each package contains a unique identifier. This unique identifier is used to fetch the ‘package.json’ file for each package.
3. The website fetches the list of certified packages from the SCP Whitelist GitHub repository. The identifier for each of the certified packages is used as the property of a new object. Then since all versions are certified independently, all certified versions of that package are stored under its property.
4. The website uses the ‘package.json’ file from each package to extract the information in the ‘dhis2ComponentSearch’ property. This information is then used to construct an array containing all components that existed within all the fetched packages. To do this, it creates a new object for each component with the following properties: ‘name,’ ‘export,’ ‘description,’ ‘language,’ ‘dhis2Versions’, and ‘packageIndex.’ The ‘name,’ ‘export,’ ‘description,’ and ‘dhis2Versions’ properties are copied from each component, while the ‘language’ property is copied overarching specification. Finally, the ‘packageIndex’ property is used to reference the package which contained the component.

These objects and arrays collectively form the website’s virtual component collection used for search and filters. After these initialization steps have been performed, the component user is free to navigate the website, enter a search term, or apply filters. Since everything has already been fetched from each respective source of data, all subsequent actions are performed locally on the user’s environment without contacting the server.

5.5.2.8 State management

State management regarding React applications, which this website is, refers to how data such as those presented in the previous section are accessed and handled by the React components. The architecture enforced by React is very hierarchical. The website is practically one large React component, which implements other child components. Each component can pass state to child components in the form of ‘props,’ which is an object containing data. By using this, React applications pass state down the hierarchy to where it is needed. While this type of state management is used in the website, it also uses a library called Redux. Redux allows the website to save its state in a centralized object called a ‘store,’ which can be accessed by all components in the website, regardless of their placement in the hierarchy. In general, this makes it easier to handle the state, as one can circumvent the hierarchy when passing data. In total, the website relies on three types of state:

1. *Internal component state.* State defined within the components itself.
2. *‘Props’ state.* State passed from one component to its child component.
3. *Redux state.* State that circumvents the component hierarchy and can be accessed by all components.

By using what has been described so far, the website uses three rules for state management. First, if the state is always internal to the component itself, it is stored in the internal component state. Second, if the state is always an internal state to a child component, it is always passed down as ‘props’ to that component. Finally, if the state is accessed by one or more components regardless of the component hierarchy, it is stored in the Redux state.

5.5.2.9 Responsiveness

The application is built to support all screen sizes using responsive frameworks such as React-Bootstrap and Pure.css. Through using these frameworks, the application is built using a grid system that automatically scales parts of the application depending on the size of the screen.

To summarize, the website is the interface supporting component users' processes to discover and reuse components. It is a React application whose primary purpose is to display a component gallery with all components published to the NPM registry with the "dhis2-component-search" keyword to enable their reuse. To do so, it fetches the information on all packages. It then creates a component collection based on NPM, UNPKG, and their certification status from the SCP Whitelist.

5.5.3 SCP CLI

The SCP CLI serves to be a tool to certify if components meet their specification and the set requirements. Thus, the CLI serves two different purposes. The first is to be used by component producers to develop reusable components as a pre-certification checker. The second is to be run automatically by the SCP Whitelist during the whitelisting process to certify the components. These are run with the two commands:

- *'dhis2-scp-cli verify'.* Certifies the package specification and the component abstraction.
- *'dhis2-scp-cli pr-verify'*. Used by the SCP Whitelist to first verify security aspects before running the 'dhis2-scp-cli verify' command above.

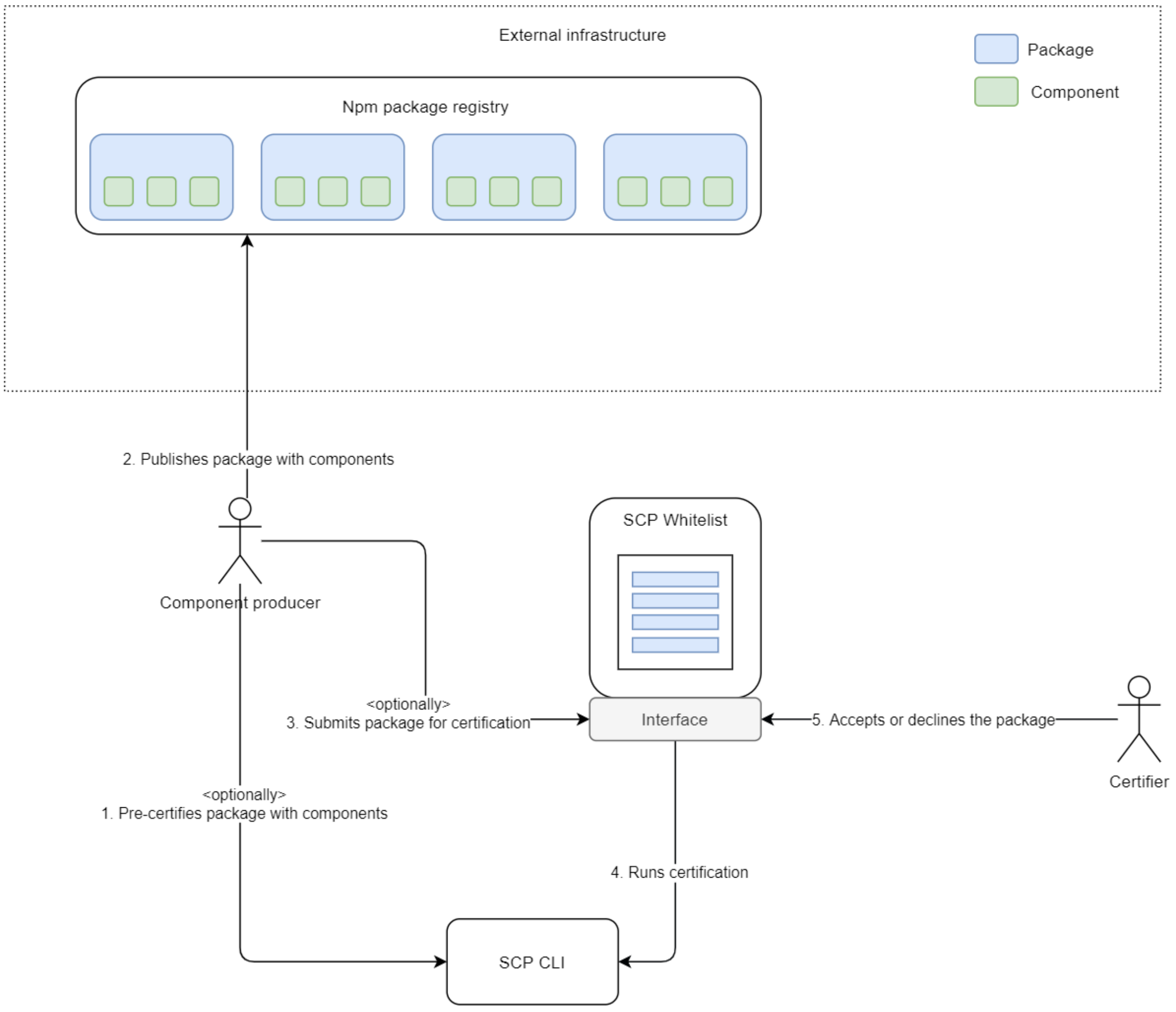


Figure 19. CLI and Whitelist architecture and supported processes.

Figure 19 above displays the SCP CLI and SCP Whitelist architecture and how it supports component producers and certifiers. A component producer initiates the process by (1) optionally pre-certifying their package with the SCP CLI before (2) publishing it to NPM. Then, the component producer (3) optionally submits the package for certification. This triggers (4) the SCP CLI to run the certification process. Finally, a certifier (5) accepts or declines the package. If accepted, the package is added to the SCP Whitelist, and if declined, the component producer receives comments describing why.

5.5.3.1 Dependencies

This section describes an overview of the SCP CLI's dependencies that needs to be present for its commands to run. The list below presents both the CLI's environment and the crucial packages it is dependent upon to run.

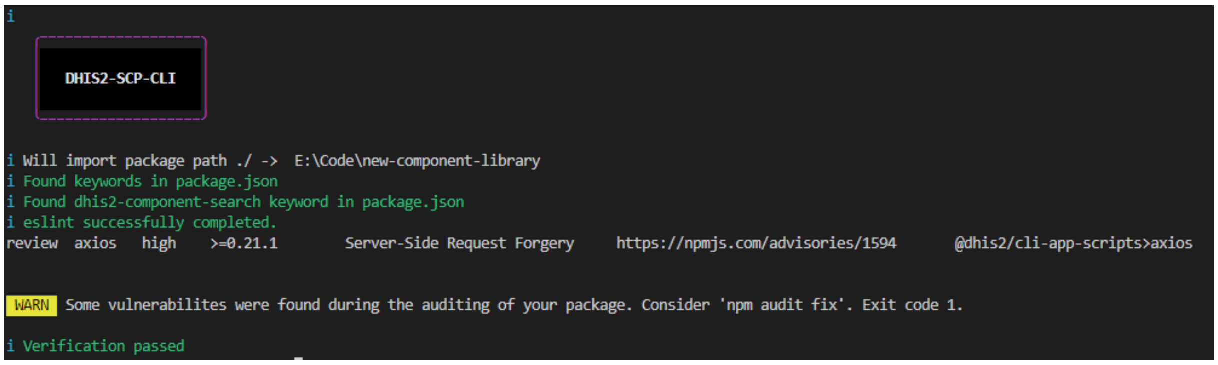
- *Node.js.* Node.js is the runtime and environment the CLI runs on (Node.js, n.d.).
- *Typescript.* Typescript extends JavaScript by adding static type definitions (*Typed JavaScript at Any Scale.*, n.d.) and is a scripting language for writing apps and scripts. It is used in tandem with JavaScript to write the SCP CLI.

- *Yargs.* Yargs is a package for building interactive command-line tools by parsing arguments and generating user interfaces (*Yargs*, n.d.). It is used to handle arguments from the component producer.
- *Jest.* Jest is a JavaScript testing framework that helps with actions such as mocking and provides information about code coverage (*Jest · Delightful JavaScript Testing*, n.d.). It is used to run tests that check if the code does what it is supposed to.
- *ESLint.* ESLint is a linter that helps find and fix problems in JavaScript code (*ESLint - Pluggable JavaScript Linter*, n.d.). It is used to lint the code and fixes problems.

5.5.3.2 Certification

As mentioned in section "5.1 Node Package Manager (NPM)", the components are distributed within units called packages. After the developer has created the package and its components, put in the required keyword, and put information in the 'dhis2ComponentSearch' property, they should run the 'verify' command. This command runs certification on the entire package and its contained components. Therefore, the package receives the certification, which is reflected in its components. A package cannot have some certified and some non-certified components. This command does the following:

1. Verifies that the 'dhis2-component-search' keyword is present in 'package.json.' This needs to be present for the SCP to find the package in the NPM registry.
2. Verifies that the 'dhis2ComponentSearch' property is present in 'package.json' and verifies its structure.
    a. Checks that an array of components exists and that it has the required properties.
    b. Checks if the declared export matches a component exported from the package.
3. Verifies that the package passes an 'npm audit' check. This checks if the package is a valid NPM package, if it has any vulnerabilities, and if any updates to its dependencies are available.
4. Verifies that the package passes an 'eslint' check. The 'eslint' check conducts 'linting' of the code, verifying that the source code in the components does not have any syntax errors that cause errors.



```
i

    DHIS2-SCP-CLI

i Will import package path ./ ->  E:\Code\new-component-library
i Found keywords in package.json
i Found dhis2-component-search keyword in package.json
i eslint successfully completed.
review  axios   high    >=0.21.1        Server-Side Request Forgery     https://npmjs.com/advisories/1594       @dhis2/cli-app-scripts>axios


WARN Some vulnerabilites were found during the auditing of your package. Consider 'npm audit fix'. Exit code 1.

i Verification passed
```

Figure 20. SCP CLI Certification

The certification ran during the 'pr-verify' command runs many of the same that the 'verify' command runs and adds some. The 'pr-verify' command is explained in the "5.5.4 SCP Whitelist" section. Figure 20 above displays the log printed from running the pre-certification on a package. Here we can see that the keywords are found, the property is checked, and then an 'eslint' and 'npm audit' check. In addition, we can see that during the ‘npm audit’ check, the ‘axios’ dependency should be reviewed and potentially updated to a new version number to fix security issues.

5.5.3.3 Unit tests

The unit tests performed using Jest are to check if the functionality implemented by the CLI is working as intended. Table 25 below describes the unit tests performed on the ‘verify’ command.

Test	Function
Correct ‘package.json’ structure	Tests if the complete ‘package.json’ structure is handled correctly.
Missing keyword	Tests if the ‘package.json’ file missing the keyword property is handled correctly.
All fields are empty	Tests if component abstraction fields in the ‘package.json’ all being empty are handled correctly.
Exception in processing	Tests if an error thrown during processing is handled correctly.
Parameterized tests	Tests multiple parameterized cases on the component array to see if they are handled correctly.
Correctly specified DHIS2 versions	Tests if correctly specified DHIS2 versions are handled correctly.
Invalid DHIS2 versions	Tests if invalid specified DHIS2 versions are handled correctly.
ESLint completes successfully	Tests if running ESLint are handled correctly.
ESLint completes with warnings	Tests if running ESLint and encountered warnings are correctly printed to the console.

Table 25. CLI 'verify' unit tests.

Tests performed on the pull request certification, described in Table 26 below:

Test	Function
Missing ‘pull_request’ property	Tests if the input data is missing the ‘pull_request’ property is handled correctly.
‘pull_request’ is undefined	Tests if the ‘pull_request’ property being undefined is handled correctly.
Changed more than one file	Tests if more than the CSV file being changed is handled correctly.
Changed the wrong file	Tests if changing the wrong file is handled correctly.
Changed the correct file	Tests if changing the correct file is handled correctly.
Right output from a correct run	Tests if the output of a correct run has the right output.
Could not fetch diff file	Tests if problems with fetching the diff file are handled correctly.

Failure on multiple packages added	Tests if multiple packages being added to the CSV file give an error.
Correctly extracting package data	Tests if package data is extracted the correct way.
Verify package identifier	Tests if the package identifier is verified correctly.

Table 26. CLI 'pr-verify' unit tests.

To summarize, the SCP CLI is used to pre-certify packages locally by the component producer and is also used by the SCP Whitelist to perform the actual certification covered in the next section.

5.5.4 SCP Whitelist

The SCP Whitelist serves to handle the list of certified packages and the process by which they are certified. The whitelist comes in the form of a GitHub repository containing a comma-separated values (CSV) file where package identifier and version are tracked. This file can at any time be parsed to retrieve a list of all certified packages and their versions. Additionally, the whitelist contains instructions on how new entries are handled and added. These new entries happen through pull requests. A pull request is a feature in GitHub where one can propose changes and set those changes up for review. Then another person with the necessary access rights can review this pull request and apply it, triggering a merge of their proposal onto the source code.

5.5.4.1 Certification Process

A component producer submits their package by opening a pull request and inputting their package identifier and package version to apply for certification. This submitting triggers an automated GitHub actions workflow. The whitelist uses the ‘pr-verify’ command from the SCP CLI presented in the previous section to conduct the certification. The process can be summarized as followed:

1. The event is checked to verify that a pull request triggered it and that only the CSV file was modified. For security reasons, only the CSV file can be modified, and the rest of the workflow will not run if any other files have been modified. Since this is a public repository, users could attempt to change the files describing the GitHub workflow, which cannot be allowed.
2. The ‘package.json’ file for the package is fetched using UNPKG. Afterward, the file is scanned to retrieve its defined Git repository. This repository link is utilized for cloning the package’s Git repository. After cloning, all dependencies are installed.
3. After cloning and building the project in the workflow, the subsequent checks are identical to what is described in "5.5.3.2 Certification" in the previous section.
4. After those steps are performed, the workflow prints out the GitHub workflow results, which form the certification basis.

After a component producer has submitted their package for certification and all the automated checks have been performed, it is a certifier’s job to manually reviewed the submitted pull request. The role of a certifier was defined in section "3.2.1 CBSE Processes" and is someone who performs certification of components. The certifier needs to go through each step of the certification process, read the console prints, potentially test the package, and decide if the package meets the specification and quality needed for certification. After making up their mind, the certifier would either accept the pull request or decline it. If it is accepted, the package is added to the CSV file as a certified package. If it is denied, the certifier should add a comment explaining the reason so that the component producer could fix the issues and resubmit.

The two types of certification are referred to as hard certification and soft certification, as first defined in section "5.4.2.2 Certification". Hard certification was defined as all the requirements vital for the package to be accepted and are all checks performed automatically, i.e., checking properties and keywords. Soft certification was defined as those requirements hard to automate and instead need to be checked manually. Soft certification requirements include, i.e., testing the package in a project and reading documentation. The SCP CLI certifies the hard certification requirements and some of the soft requirements and prints them for review by a certifier.

This chapter has presented the architecture of the SCP and its three modules, the SCP Website, the SCP CLI, and the SCP Whitelist, and how they support actors in their processes. Further, it has described their primary roles in the architecture and how they interact with each other to form the architecture. Understanding the interplay between these modules is essential to make sense of the design principles extracted from them.

5.6 Initial design principles

This section presents and discusses the initial design principles. These were created directly after the design and development process and before the “after development” evaluation phase. These design principles are based upon initial findings from empirical data and the outcomes from the development process and provide present them as simple assertions. I think it is essential to show the design principle’s initial version, justification, and motivation as they influenced the elaborated design principles in section "6.7 Final design principles".

5.6.1 Principle of an interface for component discovery

A component management system provides an interface for component discovery. For component users to interact with a component management system, this interface needs to be present. Looking back at the SCP’s construction process, an interface for component users to discover components was one of the first things the DHIS2 Core Team brought up in event 1. It was continually reinforced throughout design and development and further evaluations as an integral part of a component management system. The SCP implements this design principle through a website, which collects all the components and then provides an interface for component users to discover and reuse them. Based on the SCP’s construction process and the resulting artifact, this principle is so far empirically supported as an integral part of a component management system.

5.6.2 Principle of component certification

A component management system provides a service for component certification. To handle certification of components, component producers need a service to submit their components for certification. This service then provides means for certifiers to complete the certification process and exposes all that is certified. Looking back at the SCP's construction process, the DHIS2 Core Team expressed their interest in quality assurance of components already from event 1. Such quality assurance promotes contributions based on the certification requirements and potentially makes component users more likely to use the components with a certified level of quality. Thus, the DHIS2 Core Team initiated the discussion on certification to provide more utility for component users. How certification was to be implemented was decided later in event 3, as a whitelist, and reinforced in event 5, with certification requirements. The SCP implements this design principle through a whitelist of certified components where component producers publish components for certification and certifiers approve or decline them. Based on the SCP's construction process and the resulting artifact, component certification is so far empirically supported to provide quality assurance on components in a component management system.

5.6.3 Principle of mechanisms for pre-certification

A component management system provides mechanisms to pre-certify components. Building on the *Principle of component certification*, pre-certification is a way for component producers to certify if their component matches the certification requirements locally in their environment before submitting it for certification. Providing such mechanisms makes component producers aware if they need to fix any issues before submitting it for certification, making them more confident in its success. The need for such mechanisms was found through the design and development process by the research team. We noticed it was not easy to understand precisely what the certification requirements needed. The SCP implements this design principle through a CLI that component producers can run to pre-certify their components. Based on the SCP's construction process and the resulting artifact, mechanisms for pre-certification are so far empirically supported to aid component producers in using the service for certification in a component management system.

5.6.4 Principle of infrastructure for hosting and distribution

A component management system requires infrastructure for hosting and distribution of components. First, the components need to be hosted in a database or registry where component producers can publish them. Second, those hosted components need methods for distribution so that component users can reuse them after using the interface to discover components. The topics of hosting and distribution were brought up during event 1 by the DHIS2 Core Team, where it was self-evident that some infrastructure needed to be in place for a component management system to work. In the context of JavaScript components, the SCP relies on NPM to provide this infrastructure. NPM provides the infrastructure to publish, host, distribute, and use packages and is already synonymous with package reuse in the Node.js ecosystem. Based on the SCP's construction process and the resulting artifact, infrastructure for hosting and distribution is empirically supported as a requirement for a component management system.

5.6.5 Principle of the package-to-component abstraction

If a component management system is based upon an ecosystem with widely used package managers such as NPM, it provides functionality to handle the package-to-component abstraction. Thus, for a component management system to be helpful, it shows component users all the available components, not only the packages. Therefore, it needs to extract all components from their packages and present those to the component user. During event 2, the project supervisors asked what utility the component management system had in its current state if it is only built on top of NPM and handles NPM packages in a slightly different way. Additionally, during event 3, a member of the DHIS2 Core Team mirrored this by asking about its utility for HISP groups if it only handles NPM packages. Instead, the DHIS2 Core Team member proposed extracting all components within a package and presenting those. The SCP implements this design principle by requiring packages to describe their components in the JSON format, which are read and parsed after fetching the packages. Based on the SCP's construction process and the resulting artifact, the package-to-component abstraction is empirically supported as an essential approach to increase the component management system's utility for component users.

Design principle title	Implementation in the SCP
Principle of an interface for component discovery	SCP Website
Principle of component certification	SCP Whitelist
Principle of mechanisms for pre-certification	SCP CLI
Principle of infrastructure for hosting and distribution	External: NPM
Principle of the package-to-component abstraction	Requirement for packages

Table 27. Initial design principles.

This section has presented the initial design principles, created directly after the design and development process and before the "after development" evaluation phase. These design principles are based upon initial findings from empirical evidence and the outcomes from the development process and provide present them as simple assertions. The initial design principles are presented in Table 27 above.

Chapter 6: Final Evaluation

Through evaluations, one can provide evidence that a new technology developed in DSR projects achieves its purpose (Venable et al., 2012). Therefore, to provide evidence that the SCP can aid in addressing the problem that software components created by HISP groups to aid them during localization are not reused, it needs to be evaluated. Therefore, in the "4.4.4 Demonstration and evaluation" section, three phases of evaluations were presented: “before development,” “during development,” and “after development.” Furthermore, the results from the evaluations in the “before development” and “during development” phases have previously been presented in the "5.4 SCP’s construction process" section. Thus, the outcomes of these results are the SCP and the initial design principles presented in section "5.6 Initial design principles".

This chapter presents the results of the data collected from the expert evaluation performed in the “after development” phase with a member of the DHIS2 Core Team. The evaluation was designed and performed after we finished SCP’s construction and after the initial design principles were created. The evaluation was designed as a final measure of how well the SCP performs and provides evidence to help justify or refute the design principles. The goal of the evaluation was to collectively measure the SCP’s efficacy, utility, and technical feasibility, which was introduced in "4.4.4 Demonstration and evaluation". The evaluation was conducted following the expert evaluation method described in "4.4.4.4 Design the evaluation in detail" and performed over two sessions. In the first session, the DHIS2 Core Team member reviewed the processes the SCP is designed to support: The component producer in publishing their package to NPM and submitting it for certification, the certifier in certifying the package, and the component user in discovering components through the interface to reuse them. The second session opened for a more in-depth discussion on these processes and how the SCP supports them.

In this section, the DHIS2 Core Team member is referred to as the ‘participant.’ First, the results from both evaluation sessions are presented together with the five previously created design principles to structure the feedback. Then, the results are described concerning the three evaluation criteria: efficacy, utility, and technical feasibility. Last, this chapter presents the final design principles.

6.1 Principle of an interface for component discovery

This section discusses the collected data regarding an interface for component discovery and its implementation in the SCP using a website. The participant referred to the website as “definitely the first thing that should exist” in the SCP. The discovery ability of the website, meaning how easy it is for component users to discover new components through, i.e., filters and search fields, was defined as a “good start,” where some things are missing.

Three improvements related to how the components are displayed on the website and could serve to enhance its current design were suggested. The first improvement suggested was displaying the components more hierarchically to make it easier to identify components within the same package. Second, the participant proposed to add an option to display a screenshot for each component in their implemented visual form. Third, instead of initially displaying a random assortment of components, it would be better to display certified components only. Thus, promoting components that have gone through the quality assurance process. These three suggestions for improvements are related to how the components are displayed on the website and could serve to enhance its current design.

Further, to support more efficient reuse, the participant suggests that more focus is put on facilitating a component’s documentation to component users. Some components might include extensive requirements to be reused in apps. For example, they can be reliant on a specific DHIS2 version or specific metadata configuration of the DHIS2 platform to function. Therefore, for component users to make an educated decision in discovering components and reusing them, documentation describing these requirements and how to implement them must be facilitated.

The participant also brings up the challenge of measuring a component’s active development and displaying it in the interface. For example, how can component users know that the components they use will continue to be updated and maintained? For this purpose, the participant recommends exploring different metrics that can be displayed. So, for component users to continue relying on components, there needs to be a way to measure ongoing maintenance.

The *Principle of an interface for component discovery* can be implemented differently in different component management systems. The SCP implements this design principle as a website. The benefits of a website are that it visually provides options to navigate the components. However, a disadvantage is that component users need to navigate to a website and cannot access the components from their local environment. Therefore, the participant was also asked if other options could be more suitable. For this purpose, the participant responded that a website is “definitely the first thing that should exist.” However, it would also be valuable to access the interface from the environment where developers are doing their active development. The most evident options for this are a CLI tool or a plugin for the code editor that accesses the same interface.

With regards to the evaluation criteria:

- *Efficacy.* The website part of the SCP reaches its goal of acting as an interface for discovering components by providing visual discovery of components.
- *Utility.* In addition to reaching its goal, the website is useful for component users and can enable reuse.
- *Technical feasibility.* The evaluation measured an already instantiated (March & Smith, 1995) website, providing evidence of its technical feasibility to be built. As such, it is a feasible way to implement component discovery to enable component reuse.

To summarize, an interface to discover components is an essential element of a component management system. The SCP’s implementation of this design principle using a website is a “good start,” where there are some notable improvements for component discovery and facilitating a component’s documentation. Additionally, other options to implement this design principle can take the shape of a CLI or a code editor plugin. Conclusively, an interface for component discovery is so far empirically supported. The question is rather how it should be implemented and designed to be most helpful.

6.2 Principle of component certification

This section discusses the collected data regarding the component certification and its implementation in the SCP using a whitelist. The participant said that the implementation of a whitelist worked well. One benefit of using GitHub to store the whitelist is that the entire history of the whitelist is maintained and saved. Additionally, GitHub provides the infrastructure needed by such a whitelist to accept component producer's requests for certification while staying in control of the whitelist itself, lessening the amount of work needed to create infrastructure. Thus, the whitelist was proven beneficial as it is not obtrusive to the component producer's existing development practices. Nonetheless, the participant has some "qualms about the maintenance and management plans for this system after it is developed" when referring to the manual work needed by certifiers.

The participant mentioned a few challenges that arise with the current implementation. First, the learning curve is a challenge if the component producer is not used to GitHub from before. The GitHub interface provides much functionality, which for new component producers can be overwhelming. Therefore, component producers will need to follow a guide for publishing their components for certification as it is not directly intuitive. Second, the current certification process requires the source code of components to be stored in a GitHub repository with a 'tag' corresponding to the package version published for certification. The first issue with this is that repositories from other providers such as GitLab and Bitbucket cannot be certified. Additionally, creating the 'tag' as a duplicate of each package version can become tedious. As component producers create the 'tag' in GitHub manually, there is a possibility to cheat the certification by willingly 'tagging' other versions, resulting in the certification process being run on the wrong components. The third challenge is that the logs presenting the certification results are hard to read. The participant recommended splitting the certification process into multiple, distinct steps so that the certifier can comment and review each step individually. The participant explains that distinct steps make it easier for the certifier to review the package and let the component producer know which certification step went wrong.

Two reemerging concepts of importance for certification were that of hard certification and soft certification. The participant thought of hard certification as something that should not necessarily be part of the whitelist. The automatic checks that are part of this process should always be applied to all packages within the platform, not only those published for certification. Soft certification was defined as "very important" as the component is certified if it is actually "decent" and "useful" and has some level of quality. The participant thinks this should be a manual task because it is hard to automate. Therefore, hard certification is critical to guarantee that the packages are found and integrated correctly, while soft certification should be the certification requirements that check the component's quality.

A question that has been hard to answer throughout the artifact construction process is whether certification should be on the package or component level. The participant thinks this depends on what type of certification is wanted. On the one hand, having it on the component level would undoubtedly be good as it can be shown that the component in itself is useful. On the other hand, having it on the package level is easier to handle and instead shows that the package contains useful components without checking each of them individually. Based on this, keeping the certification on package level shows that packages contain useful components, which is the most useful.

The *Principle of component certification* can be implemented differently in different component management systems. For example, the SCP implements this design principle as a whitelist hosted in GitHub. The participant was asked if other options could be more suitable. The participant responded that it is possible to create a custom interface in the website that interacts with the current implementation of the whitelist behind the scenes. Using this method, component producers would not have to interact directly with GitHub and instead interact with a more understandable, custom interface.

With regards to the evaluation criteria:

- *Efficacy.* The whitelist part of the SCP reaches its goal of certifying components, with some clear challenges regarding certification requirements, the learning curve, and hard-to-read logs.
- *Utility.* While the goal is reached, the potential usefulness of certification for component users is more apparent as long as the certification requirements are well-defined.
- *Technical feasibility.* The evaluation measured an already instantiated (March & Smith, 1995) whitelist, providing evidence of its technical feasibility to be built. Furthermore, since the whitelist relies on free GitHub infrastructure, it has no direct operational costs. Furthermore, it requires maintenance only when certification requirements change.
- *Operational feasibility.* This evaluation criterion had not been considered during the final evaluation design but emerged as an essential topic during its execution. Operational feasibility measures the degree to which stakeholders will support and operate the artifact (Prat et al., 2015). The whitelist requires manual work by certifiers to check the soft requirements and manually approve the request for certification. Therefore, the social cost of operation is measured to be expensive.

In summary, a component certification is an essential, complex part of a component management system. The SCP's implementation of this design principle using a whitelist in GitHub is functional, but with notable challenges such as its learning curve for component producers and hard-to-read logs for certifiers. The automated hard certification should not be a specific part of the whitelist requirements but instead performed on all components regardless of certification. The soft certification guidelines need to be better defined so that they can be considered manually. Furthermore, its operational feasibility was measured to be expensive because of the manual work needed by certifiers. Last, it was agreed that keeping certification on package level is appropriate as it provides the needed level of usefulness in the SCP.

6.3 Principle of mechanisms for pre-certification

This section discusses the collected data regarding the mechanisms for pre-certification and its implementation in the SCP using a CLI. The participant explained that the CLI as a mechanism for pre-certification did not feel critical. As the participant described, the component producer can go through the entire workflow of creating a package, publishing, and submitting it for certification without ever using the SCP CLI. Nonetheless, the participant referred to it as “nice to have for sure” and “quite useful,” as it allows the component producer to perform pre-certification locally. Pre-certification can save time in the development process and make it easier to understand what is necessary to conform to the certification standards. The participant proposed that integration with already existing DHIS2 command-line tools can be beneficial. It reduces the number of tools component producers need to download and solves a few occurring bugs.

With regards to the evaluation criteria:

- *Efficacy.* The CLI part of the SCP reaches its goal of pre-certifying components.
- *Utility.* While this goal is reached, its usefulness can be debated. In its current form, pre-certification is "nice to have" while not critical to the component publishing process.
- *Technical feasibility.* The evaluation measured an already instantiated (March & Smith, 1995) CLI, providing evidence of its technical feasibility to be built. In addition, it requires maintenance only when certification standards are changed, and there is no direct cost of operation. As such, it is a feasible way of implementing component pre-certification.

In summary, the SCP's implementation of this design principle using a CLI worked well as it is easy and quick for component producers to use. However, the mechanism for pre-certification of components was referred to as a practical but not critical part of the SCP. Therefore, the design principle is removed and instead included in the overarching *Principle of component certification*. Again, while not critical, the designers of component management systems should consider options for local pre-certification that make it easier for component producers to understand the certification and integration requirements.

6.4 Principle of infrastructure for hosting and distribution

This section discusses the collected data regarding the infrastructure for hosting and distribution and its implementation in the SCP using the external infrastructure NPM. The participant referred to the infrastructure as an essential part of the SCP to make the components available to a broader base and make it easier for users to participate. Additionally, HISP groups are already heavily used to working within the NPM ecosystem concerning components. As such, it makes the most sense to let the components be available there.

During this evaluation, I brought up creating a layer on top of NPM to make it easier to abstract away some of the NPM infrastructure. Currently, publishing the package is outside the scope of the SCP as one does so directly to NPM without ever interacting with the SCP itself. A layer on top could be responsible for submitting the components for certification and subsequently publishing them to NPM. It could also allow for creating a more extensive component management system outside the scope of only web components. For instance, Android components, server-side components, and Docker containers could be published. The participant said that there is a need for other types of components to be reused in the DHIS2 ecosystem and that it is something worth exploring in the future. One way is to adapt the SCP CLI to handle both pre-certification and the subsequent publication process to NPM.

With regards to the evaluation criteria:

- *Efficacy.* The SCP's integration with the external service NPM reaches its goal of providing the infrastructure for hosting and distribution.
- *Utility.* With the goal reached, the usefulness of this infrastructure is evident to allow hosting and distribution.

- *Technical feasibility.* The evaluation measured an already instantiated (March & Smith, 1995) infrastructure, providing evidence of its technical feasibility to be integrated and built. Additionally, the integration with the NPM infrastructure makes the operational costs low.

In summary, the infrastructure for hosting and distribution was referred to as an integral part of a component management system. The SCP's implementation of this design principle using the external service NPM worked well. It provides all needed infrastructure, and component producers already know how to use it. Additionally, there are options for extending it further.

6.5 Principle of the package-to-component abstraction

This section discusses the collected data regarding the package-to-component abstraction and its implementation in the SCP through a JSON object. The participant was the first to develop the idea to extract the components from its packages during earlier evaluations. While the participant refers to it as an essential element of the SCP, the current implementation has some challenges. The first improvement suggested by the participant was that if a package's components are self-described in the code (i.e., through 'JSDoc' comments), the SCP CLI could read these comments and automatically create the entire package-to-component abstraction. Automating this process can reduce entry barriers, reduce the required maintenance, and incentivize good documentation practices. The second improvement suggested was concerning the manually entered 'dhis2Version' property describing DHIS2 platform version compatibility. Instead of having component producers manually input the DHIS2 platform version, it can be possible to extract this automatically.

With regards to the evaluation criteria:

- *Efficacy.* The SCP's package-to-component abstraction reaches its goal of extracting components.
- *Utility.* Reaching the goal is crucial for the SCP to provide utility for component users.
- *Technical feasibility.* Incorporating the abstraction into the SCP is in itself technically feasible. Further extending the abstraction to be automatic is also thought to be technically feasible.
- *Operational feasibility.* Similar to the principle of component certification, questions regarding operational feasibility also emerged. The package-to-component abstraction in its current implementation requires component producers to specify it manually. Therefore, the cost of integrating the package-to-component abstraction in daily practice is measured to be more expensive.

In summary, the package-to-component abstraction was referred to by the participant as an integral part of a component management system that uses infrastructure distinguishing between packages – the unit of distribution, and components – the unit of reuse. The SCP's implementation of this design principle through manual works is functional. However, it can be automated to reduce entry barriers, reduce the required maintenance and incentivize good documentation practices.

6.6 Evaluation criteria results

This section discusses the results from the evaluation concerning the three predefined criteria: efficacy, utility, and technical feasibility, and the emerging "operational feasibility" criteria. To discuss these criteria, each part from the previous sections discusses the SCP's individual goals, and here these are collectively summarized.

First, regarding the efficacy and if the SCP can solve the research problem. All parts seem to perform satisfactorily during the evaluation. Thus, the evaluation provides evidence that it has the required functionality to support HISP groups in reusing software components. Still, much more can be done on each of these individual parts and the SCP as a whole. As such, this evaluation on efficacy should be taken as a formative, ongoing assessment.

Next, regarding utility, measuring the potential usefulness of the SCP if it reaches its goals during the efficacy measurement. Most of the SCP measures well. All parts except the mechanisms for pre-certification are essential to the integral workings of the SCP. In contrast, the mechanisms for pre-certification are practical if component producers want to use it but not critical. In total, this indicates that the SCP has the potential to aid in addressing the problem that software components created by HISP groups to aid them during localization are not reused when it is finalized and put into a real context. As the evaluation is formative and involved an actor who is not part of a HISP group, it is crucial to consider utility as the potential usefulness and not clear evidence of usefulness.

Then for technical feasibility, the ease with which the SCP can be built and operated from a technical point of view. The SCP and all the parts described are already instantiated, which provides evidence that it can be built. Further, the evaluation was performed using the instantiated SCP with no direct challenges to its operation, which provides evidence that it can be operated from a technical point of view.

Finally, regarding the emerging criteria of operational feasibility, the ease with which the SCP is operated by actors and integrated into daily practice. When discussing technical feasibility, issues with its operational feasibility emerged. Its maintainability and cost of operation have been questioned by certifiers regarding the manual approval or declining of components in the whitelist, making it essential to consider other options that are not manual for its continued sustainability. Additionally, the cost of operation applies to the component producers in HISP groups and the broader localization problem that developing apps as an approach to localization is resource-intensive. Therefore, the resources spent to participate in reusing software components through the SCP need to be low for it to be useful. Conclusively, this challenge creates a balance, where the value for participation must be higher than the cost of integration and operation.

To summarize the final evaluation, the SCP performs satisfactorily in all the three original evaluation criteria through this formative, expert evaluation with an expert in developing tools for the DHIS2 ecosystem. However, the emerging criteria of operational feasibility illuminate challenges concerning the cost of operation for certifiers and the integration cost in daily practice for component producers.

The first goal of this evaluation was to measure the evaluation criteria to get feedback on the SCP's further construction. The second goal was to provide evidence that supports or refutes the initial design principles. Conclusively, the evaluation primarily justifies the existence of the design principles and allows the elaboration of their content. The exception is that the *Principle of mechanisms for pre-certification* is removed based on its less-than-critical nature and instead included in the principle of component certification.

6.7 Final design principles

This section presents the final design principles created by elaborating on those presented in section "5.6 Initial design principles" and the subsequent final evaluation. As mentioned in section "4.1.2 Research contribution", the content of the design principles follows the structure presented by Gregor et al. (2020). This structure splits the design principles into four sections. The first, "Aim, Implementer, and User," describes who the design principle is for, the overarching aim, and who should implement it. The second, "Context," describes for what context the design principle is created. The third, "Mechanism," describes what must be accomplished to serve the aim. Finally, "Rationale" is the justification for why the mechanism will serve to accomplish the aim. This section presents the final design principles using this structure. The design principles are discussed with literature in section "7.1 Design principles".

6.7.1 Principle of an interface for component discovery

Principle of an interface for component discovery	
Aim, Implementer, and User	For vendors to design effective component management systems for use by implementation groups…
Context	in enterprise software platform ecosystems…
Mechanism	ensure there are interfaces for component discovery…
Rationale	providing component users with the tools to discover suitable components and enabling their reuse.

Table 28. Principle of an interface for component discovery.

6.7.2 Principle of component certification

Principle of component certification	
Aim, Implementer, and User	For vendors to design effective component management systems for use by implementation groups…
Context	in enterprise software platform ecosystems…
Mechanism	ensure component producers can submit components for certification…
Rationale	to determine, based on defined requirements, if they have a high quality and should be promoted.

Table 29. Principle of component certification.

6.7.3 Principle of infrastructure for hosting and distribution

Principle of infrastructure for hosting and distribution	
Aim, Implementer, and User	For vendors to design effective component management systems for use by implementation groups…
Context	in enterprise software platform ecosystems…

Mechanism	ensure there is infrastructure for hosting and distributing packages…
Rationale	enabling their cataloging, management, distribution, and maintenance.

Table 30. Principle of infrastructure for hosting and distribution.

6.7.4 Principle of the package-to-component abstraction

Principle of the package-to-component abstraction	
Aim, Implementer, and User	For vendors to design effective component management systems for use by implementation groups…
Context	in enterprise software platform ecosystems using package manager infrastructure…
Mechanism	ensure there are mechanisms for exposing the individual components within a package…
Rationale	because the component user needs to know the contents of a package to discover its components effectively.

Table 31. Principle of the package-to-component abstraction.

Chapter 7: Discussion

This chapter discusses this thesis' contribution with existing literature and takes an overarching perspective on the addressed research problem. This chapter first discusses the four design principles that form this thesis' contribution with the CBSE literature. Then, the component management system is considered concerning the context of enterprise software platform ecosystems and their actors concerning its socio-technical nature. Furthermore, this chapter looks at bootstrapping the component management system through internal reuse. Last, this chapter presents reflections on team management before discussing the limitations of the thesis.

7.1 Design principles

This section discusses the final design principles with the CBSE literature and explicitly specifies this thesis' contribution. The following research question was asked: *"What are design principles for component management systems within enterprise software platform ecosystems?"*. To answer this research question, this thesis contributes four design principles for component management systems in enterprise software platform ecosystems: the *Principle of an interface for component discovery*, the *Principle of component certification*, the *Principle of infrastructure for hosting and distribution*, and the *Principle of the package-to-component abstraction*.

7.1.1 Principle of an interface for component discovery

As discussed in section "3.2 Component-based software engineering", component identification is the process of identifying suitable and useful components to reuse in applications (Sommerville, 2016). To do this, component users search for components either in the company's component repository or through other open-source repositories (Sommerville, 2016). Although reusable components can exist, the burden of finding them is often up to the individual component user (B et al., 2010), which also resonates with the HISP groups' challenges in discovering suitable components. Understanding and supporting the "discovery" of components are essential to leveraging CBSE's promises of shorter development time and increased reliability (Kumar & Tiwari, 2020). Additionally, Kumar & Tiwari (2020) presents a list of issues concerning selecting suitable components, i.e., the component's level of reusability, its requirements, and its scope of maintenance. These issues need to be considered and explored further within the component management system's residing ecosystem to understand what its users consider essential.

Therefore, understanding what impacts the component users' component discovery process is essential in designing interfaces to support that process. The *Principle of an interface for component discovery* explicitly specifies the need for an interface to discover suitable components from the underlying infrastructure to enable reuse.

7.1.2 Principle of component certification

As discussed in section "3.2 Component-based software engineering", component certification is the process of certifying if a component meets its specification and if it has reached an acceptable quality standard before it is made available for reuse (Sommerville, 2016). One difference between the certification implemented in the SCP and the literature is that literature mentions certification as an event occurring before the component is made available for reuse. Throughout the research process, there has been a consensus from both the research team and the DHIS2 Core Team to not gatekeep contributions from the ecosystem more than necessary. Because of this, certification is performed voluntarily if the component producer wishes to submit it for certification and is not a requirement for it to be available in the SCP. The result is two sets of components, one uncertified and one certified, where both sets need to have the integration requirements described in section "5.5.1 Package requirements" implemented. The hope is that more components are published by keeping the bar of entry low for component producers.

In the previously discussed literature, certification is presented as necessary (Sommerville, 2016) and a requirement (Heineman & Councill, 2001), but that the lack of specifically established certification procedures is one of the major issues within CBSE (Kumar & Tiwari, 2020). This thesis's findings resonate with this challenge; the DHIS2 Core Team brought up certification as essential to perform quality assurance on components and increase their trustfulness. However, defining specific certification procedures was challenging both for the research team and the DHIS2 Core Team. In the end, the concepts of hard certification and soft certification were developed through discussion, where hard certification is the integration requirements described in section "5.5.1 Package requirements" and are required by all packages. In contrast, soft certification is the requirements which are hard to automate and requires manual certification. One issue with these soft certification requirements is that they require subjective decision-making. Making such certification fair and equal for all components is, therefore, a challenge. Kumar & Tiwari (2020) present qualitative models with pre-defined assumptions and guidelines to address this challenge.

Another topic of interest discussed in the literature related to certification is that of certifiers and who should be responsible for certifying components (Kotonya et al., 2003; Sommerville, 2016; Szyperski, 2002). Prior literature describes leaving such certification to third-party certification authorities. In the context of a company, the component producer is often the certifier of their components based on the resources needed to perform third-party certification (Sommerville, 2016). The design principle on certification does not say who should be the certifier because it is a topic filled with uncertainty. From the start first focus group with the DHIS2 Core Team, it was expected that members of the DHIS2 Core Team would be responsible for this.

However, based on the amount of tedious, manual work needed to perform certification in the SCP, "qualms about the maintenance and management plans for this system after it is developed" were brought up by one member. Therefore, the discussion went from using a centralized certification authority such as the DHIS2 Core Team to the possibilities of crowd-sourced certification. Crowd-sourced certification could be an approach to solving the issue with time-consuming work. However, it brings other challenges regarding the trustfulness of the certification, evaluating each contribution from the same point of view, and the possibilities of cheating the system.

The conclusion is that if the procedures for certification are implemented to require less manual labor, and there are not a significant number of applicants, a centralized certification authority could potentially be responsible. On the other hand, if the procedures for certification cannot get less manual labor-intensive because of the soft certification requirements, then crowd-sourcing the certification is an option worth exploring.

7.1.3 Principle of infrastructure for hosting and distribution

One can compare the design principle to the concept of a component repository, which section "3.2 Component-based software engineering" presents as a type of database for storing components and their information for reuse (Kumar & Tiwari, 2020). In addition, component repositories are used to catalog, manage, and maintain components (Kotonya et al., 2003; Kumar & Tiwari, 2020; Sommerville, 2016; Szyperski, 2002) and form the foundation of a component management system on which other processes rely.

The *Principle of infrastructure for hosting and distribution* describes infrastructure similar to the component repositories discussed in the CBSE literature. First, the components' cataloging is essential to discover suitable components (B et al., 2010; Sommerville, 2016). Second, the components' management and maintenance are essential for the component repository's consistent efficiency (Kumar & Tiwari, 2020). Kumar & Tiwari (2020) describes the component repository's maintenance as a major concern within CBSE. The first two issues revolve around updating components and keeping multiple versions of the same component (Kumar & Tiwari, 2020). NPM addresses these issues by not allowing any changes to published components and instead requiring new versions. Thus, NPM keeps a complete history of all versions, which can be reused independently. Another issue is identifying outdated components (Kumar & Tiwari, 2020), which resonated as an issue in a HISP group's component discovery process. NPM addresses this issue by giving components a maintenance score, indicating recent updates. These solutions from NPM infrastructure can be further integrated into the SCP and can guide other component management systems in addressing these concerns. Therefore, a high emphasis on components' cataloging, management, and maintenance is essential for the infrastructure be sustainable over time and to provide utility.

The primary difference is the higher emphasis on distribution. Sommerville (2016) explains component repositories to handle an organization's reusable components instead of an ecosystem's. Within one organization, one can, for example, keep the component repository local, and it is easier to control access. While it is possible to directly adapt the notion of a component repository and apply it to a broader ecosystem, this brings the challenge of handling multiple actors with different goals, ideas, and processes. We can already see different maturity levels in the component reuse process and the distinct challenges they face by exploring two HISP groups' app development processes. I think one can expect more differences if larger parts of the ecosystem were surveyed. Additionally, the technical capabilities of the component management system must support a diverse set of geographically dispersed actors to enable their reuse of components. As it is created to support multiple, distinct implementation groups, having a well-performing distribution infrastructure is more critical than for a component repository within one organization. Therefore, a high emphasis on distribution is essential for a component management system to succeed in an enterprise software platform ecosystem. This section has first discussed the importance of infrastructure for hosting to be sustainable over time and provide utility. Finally, it discussed the importance of infrastructure for distribution to support actors in a diverse ecosystem.

7.1.4 Principle of the package-to-component abstraction

As discussed in the literature in section "3.2 Component-based software engineering", components are encapsulated within packages such as function libraries and frameworks (Kotonya et al., 2003). These packages are easy to distribute and subsequently easy for component users to reuse. However, for the component management system to provide utility in a diverse ecosystem, it must be easy for implementation groups to discover the components within packages. Popular component libraries achieve this through documentation or by allowing component users access to the source code. Unfortunately, this creates a barrier where component users need to check each package independently to know their content. To my knowledge, the CBSE literature provides no guidance on how to remove that barrier. The *Principle of the package-to-component abstraction* addresses this by suggesting that each package exposes its components in a computer-readable format. Thus, the package layer is temporarily removed, allowing the component user to explore components from all packages in the same location. This thesis argues the importance of the package-to-component abstraction to provide utility for component users and illustrates its important dimensions.

To summarize, this section has described four design principles that form this thesis' contribution for component management systems in enterprise software platform ecosystems.

7.2 Enterprise software platform ecosystems

This project has used the DHIS2 platform and its ecosystem as the object of study. DHIS2 has been conceptualized as an enterprise software platform where the vendor and implementation groups collaboratively develop and commercialize the DHIS2 platform to provide a constant stream of offerings for end-users (Foerderer et al., 2019). Nicholson et al. (2021, p. 3) introduce the concept of digital global public goods (DGPG) as digital goods designed to be "non-rivalrous, non-excludable, [and] locally relevant on a global scale." Further, Nicholson et al. (2021) conceptualize DHIS2 as a DGPG based on its open-source license, availability, and digital nature in being programmable, modularized, and re-combinable. Because of the free of charge policy, DHIS2 relies on donors to support its further maintenance and development (Nicholson et al., 2021). DHIS2's open-source and non-profit nature contrasts with other major for-profit enterprise software platforms such as Oracle and SAP. This section looks at the nature of DHIS2 and the actors in its ecosystem to consider challenges with implementing a component management system in enterprise software platform ecosystems.

The first challenge concerns the ecosystem's actors and their wish to collaborate. HISP has a long history of focusing on building local capacities and regional support in the DHIS2 ecosystem (Adu-Gyamfi et al., 2019; Nicholson et al., 2021). Additionally, HISP focuses on building partnerships with local implementation groups, Ministries of Health, and NGOs (Adu-Gyamfi et al., 2019; Nicholson et al., 2021). Furthermore, an Annual DHIS2 Conference where representatives share learnings, innovations, best practices, and other experiences with implementing DHIS2 incentivizes collaboration by actors in the ecosystem (Adu-Gyamfi et al., 2019; Nicholson et al., 2021). Considering DHIS2's ecosystem of actors and its conceptualization as a DGPG, this sets the stage for actors to have the mindset of wanting to collaborate. Thus, other enterprise software platform ecosystems also need to consider this challenge if implementing a component management system.

The second challenge concerns the incentives for participation and the prospects of payment. Looking back to section "5.2 SCP's role within the DHIS2 ecosystem", the SCP was conceptualized as a transaction platform for producers and consumers of components, with one notable difference from most transaction platforms: components are for free. This begs the question; can a component management system be sustained if the component producer's goal is not to earn money from transactions? Prospects for payment were avoided in the SCP as the focus was on gathering contributions from the DHIS2 ecosystem and not creating a marketplace. Avoiding prospects of payment was the most suitable for the DHIS2 ecosystem, considering what was described in the previous paragraph regarding the ecosystem's focus on collaboration. Nevertheless, this can differ in other ecosystems depending on their nature. There is nothing inherent about the component management system that requires its components to come free of charge. Therefore, the prospects of payment may need to be considered to create enough incentives for a component management system's use in enterprise software platform ecosystems.

While there are challenges to implementing component management systems in DHIS2 and other enterprise software platforms, I argue that the presented challenges can be solved within each platform's ecosystem. Furthermore, this thesis found nothing inherent about component management systems indicating that these challenges cannot be successfully addressed. Therefore, I argue that a component management system's success lies in the ecosystem's success in addressing these challenges.

7.3 Bootstrapping and internal reuse

As with network effects, if enough component producers are publishing quality components, component users are attracted based on their usefulness. Subsequentially, having more component users finding the component management system useful might incentivize them to publish their components and start contributing, creating a self-reinforcing loop. However, how can we get to that point? As a member of the DHIS2 Core Team said: "You can build the nicest market in the world, but if you do not have any people to buy or sell there, it is not going to work," referring to actors engaging over a market, and not the specific act of earning money. This section looks at the concept of bootstrapping and internal reuse to initialize the component management system.

The primary issue is that there must be available components of acceptable quality for the component management system to provide value to potential users. To do so, the first users need to be component producers who populate it with components. This issue resembles the bootstrapping problem discussed by Hanseth & Aanestad (2003), consisting of how one can design networks while leveraging the heterogeneity in environments. Hanseth & Aanestad (2003, p. 10) argue that it is easiest to start with "motivated and knowledgeable users who possess the necessary resources." Thus, Hanseth & Aanestad (2003) introduce design as bootstrapping, where one should first enroll those that are easiest to enroll before moving on to the more challenging.

In the case of the SCP, following design as bootstrapping, one should first attempt to attract component producers with existing components widely available. These component producers already know the value of reusing components and are knowledgeable of the component reuse process. Hanseth & Aanestad (2003, p. 10) further explains that the system should be designed to support those users "in their least critical and simplest practices." Therefore, to bootstrap the SCP, I argue it is beneficial to first implement it in one implementation group with motivated and knowledgeable users and make it suitable for their simplest practices. There, the component management system can be used to improve internal component reuse practices as an incentive. Focusing on internal reuse means the implementation group first only needs to consider themselves. Thus, if the component management system provides utility in addressing internal reuse, one can later focus on enrolling more implementation groups. The upside of focusing on internal reuse is that the component management is populated by components outside actors can utilize.

Following design as bootstrapping, the first goal would be to enable internal component reuse with a knowledgeable and motivated implementation group. Thereby, the implementation group's incentives for adoption can primarily be to improve their internal component reuse practices. Afterward, one can enroll other actors for whom the component management system will be more attractive based on its existing components and its potential success in improving component reuse. Thus, reaching the goal of enabling component reuse between implementation groups and within.

7.4 Reflections on team management

This section presents reflections on team management relating to the group work conducted together with my colleagues throughout the research process, which was explained in section "4.5 Team management". Throughout the project, I have had multiple conversations with Anastasia Bengtsson regarding our team management. Therefore, the foundation for the topics brought up in this section reflects those conversations and cannot be considered entirely my own.

The first challenge was the difficulties in establishing a rigorous and formal development process. As described earlier, we attempted to create a rigorous and formal process a while into the project. However, it did not last because of the team's inability to enforce strict deadlines and the challenges in planning data collection activities. Furthermore, the challenges in data collection activities meant it was difficult to structure the process in such a way to, i.e., iteratively have some weeks of design and development before concluding the iteration with an evaluation to inform the next iteration. Additionally, a more rigorous and formal process should describe the occurrence and structure of team meetings, a more concrete division of tasks between team members, and stricter deadlines for these tasks. In summary, such a development process would have made it easier to perform team activities such as design and development and discussions while also allowing for easier collaboration with other actors.

The second challenge concerns the fact that we did not formally appoint one team member or external actor to the role of team leader. In hindsight, this resulted in challenges with equal contributions potentially due to misaligned expectations. Avoidance of accountability was brought up in our project as a significant barrier to an effective process. Avoidance of accountability is when team members are hesitant to call out others on performance that hurts the team (Lencioni, 2010). Additionally, since we had not appointed someone to the role of team leader, such issues were left unaddressed. This issue was also noticeable during discussions over the collaborative tools Slack and Mattermost. Digital collaboration tools work well if all team members engage and contribute. However, keeping up team motivation online is often harder than meeting in a physical location as one does not directly engage with team members. This distancing can, as a side-effect, exclude team members.

These challenges combined resulted in a less efficient process that could have been solved earlier if handled. A more coherent, rigorous, and formal development process with clear goals to incentivize equal contributions could keep each team member accountable for their performance through, i.e., more use of the Kanban board and a rigid meeting structure. Based on a team assessment meeting where these challenges were brought up, we attempted to introduce such an approximated process but with limited success, primarily because it was introduced too late in the development process.

The first aspect that went well was splitting our development efforts so that each team member could primarily focus on only one part of the SCP. In my case, having a more significant focus on the SCP Website allowed me a greater understanding of its inner workings and functionality. Then during discussions, we attempted to keep the gaps in knowledge between each member as small as possible by involving each other in what we were doing. These different focuses sped up the development process by giving each team member more independence, as one had more room to develop effectively while still discussing requirements. The thesis also reflects this, as it discusses the SCP Website in greater detail compared to the SCP Whitelist and SCP CLI.

The second aspect that went well was collaborating on data collection. We were relatively new to collecting empirical data. Thus, collaborating allowed us to plan and execute data collection activities by discussing ideas and learning goals. Additionally, it allowed us to discuss which actors to collaborate with and why.

In conclusion, to avoid such challenges occurring, it can be wise to set some guidelines that all team members agree upon at the project's inception. Agreeing creates the foundation on which the collaboration is based, making it easier for team members to bring up issues they are experiencing. On the positive side, the separation of development efforts gave us a high degree of independence, speeding up artifact construction. Additionally, collaborating on data collection activities allowed for discussion of ideas and learning goals.

7.5 Limitations

I argue that the most significant limitation of this thesis is the lack of summative, naturalistic evaluations performed on the SCP in a real user context. The project has created a component management system through design and development and formative evaluations that further inform artifact construction. These evaluations have primarily been in collaboration with the DHIS2 Core Team members and have been vital in guiding the SCP's design. However, while one can argue that these evaluations are naturalistic based on the DHIS2 Core Team's potential role as certifiers and their active stake as an entity in the DHIS2 ecosystem, they do not address 'real problems' with 'real users' in a HISP group. The reason for this is primarily the lack of time because of the project's deadline and the difficulties in establishing a naturalistic research site out in a HISP group. Nonetheless, I argue that the evaluations with the DHIS2 Core Team are a good approximation and provide evidence that the artifact has the ability to support 'real users' in their component reuse processes in addressing 'real problems.'

This thesis provides evidence that the SCP has the ability to potentially provide utility in solving the problem that software components created by implementation groups to aid them during localization are not reused. The next natural step in this process is to introduce the SCP to a real user context where it can be used to address real problems. Addressing real problems in a real user context is vital to measure if the SCP actually provides utility in solving the research problem. So while this thesis provides evidence that the SCP has the ability to solve the problem, until such summative, naturalistic evaluations are performed, this thesis’s design principles can be considered work in progress and will either be justified or refuted by such an evaluation’s results.

Chapter 8: Conclusion and Future Work

This thesis has explored the research problem that software components created by implementation groups to aid them during localization are not reused. The goal is that by addressing this research problem, developing apps as an approach to localization becomes less resource-intensive. In turn, this can help address the overarching problem that generic enterprise software does not work optimally if not localized. By applying the design science research methodology, this thesis has explored creating a component management system for use in enterprise software platform ecosystems. The following research question was asked: "*What are design principles for component management systems within enterprise software platform ecosystems?".* To answer this research question, this thesis contributes four design principles for component management systems in enterprise software platform ecosystems: the *Principle of an interface for component discovery*, the *Principle of component certification*, the *Principle of infrastructure for hosting and distribution*, and the *Principle of the package-to-component abstraction*.

8.1 Future work

Based on the limitations presented in the previous chapter, there are a few avenues for future work. As this project is part of the DHIS2 Design Lab, it offers other master's students an avenue to continue the research project. Continuing the project brings the opportunity to introduce the component management system in a real user context to perform naturalistic and summative evaluations to measure its effectiveness in solving the problem. I propose an action design research (Sein et al., 2011) project where the master's students introduce the component management system to an implementation group's context, then perform interventions to measure organizational changes. In contrast to the DSR methodology followed in this project, action design research reflects that IT artifacts are ensembles shaped by the organizational setting and the development process (Sein et al., 2011). Such a project would have a two-fold focus. The first is to address a problem situation emerging in an organizational setting through interventions and evaluations (Sein et al., 2011). The second is to continue developing and evaluating the component management system to address the problem (Sein et al., 2011). The goal is then to generate prescriptive design knowledge "through building and evaluating ensemble IT artifacts in an organizational setting" (Sein et al., 2011, p. 40). Based on the socio-technical nature of the component management system, the outcomes of such a project would provide evidence of its effectiveness.

While summative and naturalistic evaluations in an organizational setting are essential, there are multiple nuances of a component management system that can also be further explored. The first is the purpose of certification and the role of certifiers. How can valuable certification be performed? The second is on the interface for component discovery. What do implementation groups explicitly look for in components, and how can the interface support this? The third is exploring the previously mentioned incentives for contributions. What incentives are needed for contributions, and how can a component management system sustain participation?

References

Adam, T., & de Savigny, D. (2009). *Systems Thinking for Health Systems Strengthening*. World Health Organization.

Adu-Gyamfi, E., Nielsen, P., & Sæbø, J. (2019, November 15). *The Dynamics of a Global Health Information Systems Research and Implementation Project*.

Avital, M., & Te'eni, D. (2009). From generative fit to generative capacity: Exploring an emerging dimension of information systems design and task performance. *Inf. Syst. J.*, *19*, 345–367. https://doi.org/10.1111/j.1365-2575.2007.00291.x

B, J., Govardhan, D., & Premchand, P. (2010). Breaking the Boundaries for Software Component Reuse Technology. *International Journal of Computer Applications*, *13*. https://doi.org/10.5120/1782-2458

Baldwin, C., & Woodard, C. J. (2008). The Architecture of Platforms: A Unified View. *Platforms, Markets and Innovation*. https://doi.org/10.2139/ssrn.1265155

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*, 77–101. https://doi.org/10.1191/1478088706qp063oa

Capretz, L., Capretz, M., & Li, D. (2001). Component-Based Software Development. *Electrical and Computer Engineering Publications*, 1834–1837. https://doi.org/10.1109/IECON.2001.975569

Constantinides, P., Henfridsson, O., & Parker, G. (2018). Platforms and Infrastructures in the Digital Age. *Information Systems Research*, *29*. https://doi.org/10.1287/isre.2018.0794

Crang, M., & Cook, I. (2007). *Doing Ethnography*.

Eisenmann, T., Parker, G., & Van Alstyne, M. (2008). Opening Platforms: How, When and Why? *Platforms, Markets and Innovation*. https://doi.org/10.2139/ssrn.1264012

ESLint—Pluggable JavaScript linter. (n.d.). ESLint - Pluggable JavaScript Linter. Retrieved February 24, 2021, from https://eslint.org/

Evans, P. C., & Gawer, A. (2016). *The Rise of the Platform Enterprise: A Global Survey* [Report]. The Center for Global Enterprise. http://epubs.surrey.ac.uk/811201/

Fischer, C., & Gregor, S. (2011). *Forms of Reasoning in the Design Science Research*. *6629*, 17–31. https://doi.org/10.1007/978-3-642-20633-7_2

Foerderer, J., Kude, T., Schuetz, S. W., & Heinzl, A. (2019). Knowledge boundaries in enterprise software platform development: Antecedents and consequences for platform governance. *Information Systems Journal*, *29*(1), 119–144. https://doi.org/10.1111/isj.12186

Gawer, A. (2009). Platforms, Markets and Innovation. *Platforms, Markets and Innovation*. https://doi.org/10.4337/9781849803311

Gawer, A. (2014). Bridging differing perspectives on technological platforms: Toward an integrative framework. *Research Policy*, *43*(7), 1239–1249. https://doi.org/10.1016/j.respol.2014.03.006

Ghazawneh, A., & Henfridsson, O. (2013). Balancing platform control and external contribution in third-party development: The boundary resources model. *Information Systems Journal*, *23*(2), 173–192. https://doi.org/10.1111/j.1365-2575.2012.00406.x

Ghazawneh, A., & Henfridsson, O. (2010). *Governing Third-Party Development through Platform boundary Resources.* 48.

Gregor, S., Chandra Kruse, L., & Seidel, S. (2020). The Anatomy of a Design Principle. *Journal of the Association for Information Systems*, *21*, 1622–1652. https://doi.org/10.17705/1jais.00649

Gregor, S., & Hevner, A. (2013). Positioning and Presenting Design Science Research for Maximum Impact. *MIS Quarterly*, *37*, 337–356. https://doi.org/10.25300/MISQ/2013/37.2.01

Hanseth, O., & Aanestad, M. (2003). Bootstrapping networks, communities and infrastructures. On the evolution of ICT solutions in health care. *Methods of Information in Medicine*, *42*.

Heineman, G., & Councill, W. (2001). *Component-Based Software Engineering: Putting the Pieces Together*.

Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram, & Sudha. (2004). Design Science in Information Systems Research. *Management Information Systems Quarterly*, *28*, 75.

Iansiti, M., & Levien, R. (2004). *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability*. Harvard Business Press.

Jacobides, M., Cennamo, C., & Gawer, A. (2018). Towards a Theory of Ecosystems. *Strategic Management Journal*, *39*. https://doi.org/10.1002/smj.2904

Jest · Delightful JavaScript Testing. (n.d.). Retrieved February 24, 2021, from https://jestjs.io/

Kauschinger, M., Schreieck, M., Böhm, M., & Krcmar, H. (2021). *Knowledge Sharing in Digital Platform Ecosystems -A Textual Analysis of SAP's Developer Community*.

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). Springer International Publishing. https://doi.org/10.1007/978-3-030-18400-1_26

Kotonya, Sommerville, & Hall. (2003). Towards a classification model for component-based software engineering research. *2003 Proceedings 29th Euromicro Conference*, 43–52. https://doi.org/10.1109/EURMIC.2003.1231566

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys*, *24*(2), 131–183. https://doi.org/10.1145/130844.130856

Kumar, S., & Tiwari, U. K. (2020). *Component-Based Software Engineering: Methods and Metrics*.

Lencioni, P. M. (2010). *The Five Dysfunctions of a Team: A Leadership Fable*. John Wiley & Sons.

Li, M. (2019a, November 2). *Facilitating collaborative meta-design: Building a shared component library for web-app development (ikke tilgjengelig) - Institutt for informatikk*. https://www.mn.uio.no/ifi/studier/masteroppgaver/is/dhis2-design-lab/dhis2-design-lab-component-library.html

Li, M. (2019b). An Approach to Addressing the Usability and Local Relevance of Generic Enterprise Software. *Selected Papers of the IRIS, Issue Nr 10 (2019)*. https://aisel.aisnet.org/iris2019/3

Li, M., & Nielsen, P. (2019, September 18). *Design Infrastructures in Global Software Platform Ecosystems*.

March, S., & Smith, G. (1995). Design and Natural Science Research on Information Technology. *Decision Support Systems*, *15*, 251–266. https://doi.org/10.1016/0167-9236(94)00041-2

Msiska, B., Nielsen, P., & Kaasbøll, J. (2019). *Leveraging Digital Health Platforms in Developing Countries: The Role of Boundary Resources* (pp. 116–126). https://doi.org/10.1007/978-3-030-18400-1_10

Nicholson, B., Nielsen, P., Sahay, S., & Sæbø, J. (2021). Digital Global Public Goods. *Proceedings of the First Virtual Conference on Implications of Information and Digital Technologies for Development*.

Node.js. (n.d.). *About*. Node.Js. Retrieved February 24, 2021, from https://nodejs.org/en/about/

Peffers, K., Rothenberger, M., Tuunanen, T., & Vaezi, R. (2012). Design Science Research Evaluation. In K. Peffers, M. Rothenberger, & B. Kuechler (Eds.), *Design Science Research in Information Systems. Advances in Theory and Practice* (pp. 398–410). Springer. https://doi.org/10.1007/978-3-642-29863-9_29

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Prat, N., Wattiau, I., & Akoka, J. (2015). A Taxonomy of Evaluation Methods for Information Systems Artifacts. *Journal of Management Information Systems*, *32*, 229–267. https://doi.org/10.1080/07421222.2015.1099390

Purao, S., Chandra Kruse, L., & Maedche, A. (2020, December 2). *The Origins of Design Principles: Where do… they all come from?*

Pure. (n.d.). Retrieved February 22, 2021, from https://purecss.io/

React – A JavaScript library for building user interfaces. (n.d.). Retrieved February 22, 2021, from https://reactjs.org/

React-Bootstrap. (n.d.). Retrieved February 22, 2021, from https://react-bootstrap.github.io/

Redux—A predictable state container for JavaScript apps. | Redux. (n.d.). Retrieved February 22, 2021, from https://redux.js.org/

Rickmann, T., Wenzel, S., & Fischbach, K. (2014). Software ecosystem orchestration: The perspective of complementors. *20th Americas Conference on Information Systems, AMCIS 2014*.

Sein, M. K., Henfridsson, O., Purao, S., Rossi, M., & Lindgren, R. (2011). Action Design Research. *MIS Quarterly*, *35*(1), 37–56. JSTOR. https://doi.org/10.2307/23043488

Sommerville, I. (2016). *Software Engineering, 10th Edition*. Pearson.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*.

Tiwana, A. (2013). Platform Ecosystems: Aligning Architecture, Governance, and Strategy. *Platform Ecosystems: Aligning Architecture, Governance, and Strategy*, 1–302.

Tremblay, M. C., Hevner, A., & Berndt, D. (2010). Focus Groups for Artifact Refinement and Evaluation in Design Research. *Commun. Assoc. Inf. Syst.* https://doi.org/10.17705/1CAIS.02627

Tripp, D. (2005). Action research: A methodological introduction. *Educação e Pesquisa*, *31*.

Tushman, M. L., & Murmann, J. P. (1998). Dominant Designs, Technology Cycles, and Organization Outcomes. *Academy of Management Proceedings*, *1998*(1), A1–A33. https://doi.org/10.5465/apbpp.1998.27643428

Typed JavaScript at Any Scale. (n.d.). Retrieved February 24, 2021, from https://www.typescriptlang.org/

UNPKG. (n.d.). Retrieved February 22, 2021, from https://unpkg.com/

Venable, J., Pries-Heje, J., & Baskerville, R. (2012). A Comprehensive Framework for Evaluation in Design Science Research. In K. Peffers, M. Rothenberger, & B. Kuechler (Eds.), *Design Science Research in Information Systems. Advances in Theory and Practice* (pp. 423–438). Springer. https://doi.org/10.1007/978-3-642-29863-9_31

Video Conferencing, Web Conferencing, Webinars, Screen Sharing. (n.d.). Zoom Video. Retrieved March 10, 2021, from https://zoom.us/

von Hippel, E. A., & Katz, R. (2002). *Shifting Innovation to Users Via Toolkits* (SSRN Scholarly Paper ID 309740). Social Science Research Network. https://doi.org/10.2139/ssrn.309740

Wareham, J., Fox, Ph. D., Paul, & Cano Giner, J. L. (2013). Technology Ecosystem Governance. *SSRN Electronic Journal*, *25*. https://doi.org/10.2139/ssrn.2201688

Wenger-Trayner, E. (2000). Communities of Practice and Social Learning Systems. *Organization*, *7*, 225–246. https://doi.org/10.1177/135050840072002

What is Fuse.js? | Fuse.js. (n.d.). Retrieved February 22, 2021, from https://fusejs.io/

Wittern, E., Suter, P., & Rajagopalan, S. (2016). *A look at the dynamics of the JavaScript package ecosystem*. 351–361. Scopus. https://doi.org/10.1145/2901739.2901743

Yargs. (n.d.). Retrieved February 24, 2021, from https://yargs.js.org/

Appendix

Detailed work distribution

These tables detail the work distribution in this project. I have collaborated with Anastasia Bengtsson in mapping out what each person did in a collaborative repository, which has formed the foundation of these tables.

Feature	Description	Implementer
Navbar	Heskja implemented a navbar for navigation options. Bengtsson refactored to a responsive navbar (using the Bootstrap navbar component), as we needed something that could work well on different devices.	Heskja, Bengtsson
Package information page	Not in use	Bengtsson
Landing page with a search form	Not in use	Bengtsson
Page refactoring	Removed landing page and put the component gallery page as the main page.	Bengtsson
Redux integration	Added redux as the central part of data handling. Uses the redux store for data, reducers for inputting that data, and actions to call the reducers.	Bengtsson
React-router routing	Added react-router to handle routing of the app.	Bengtsson
Loader component	Added a spinner when the website is opened and when it loads during the fetching of packages.	Bengtsson
Deployment	Deployed on GitHub pages.	Bengtsson
Website title and description	Added title and description.	Bengtsson
Information and help page	Bengtsson added information and help pages with information about CLI, HISP, and the DHIS2 Design Lab. Heskja filled out information about the website. Bengtsson fixed layout issues.	Heskja, Bengtsson
Package search	Heskja implemented initial functionality. Bengtsson added styling to the input field. Tverdal added functionality to search for DHIS2 keyword and dhis2 scope. Heskja refactored to search for components instead of packages. Heskja moved the location of the search bar to the navbar.	Heskja, Bengtsson, Tverdal
Package search/filters	Heskja added a filter on the framework (react/angular), certified, DHIS2 version number. Bengtsson fixed the styling issue with the DHIS2 version number filter.	Heskja, Bengtsson
Package fetch/packages fetch	Heskja implemented an initial search through the NPMS API. Tverdal implemented functionality to search within DHIS2 keyword and scope.	Heskja, Bengtsson, Tverdal

	Tverdal implemented offset fetch based on pagination. Heskja refactored code to always search within the keyword: “dhis2-component-search”. Heskja changed from using the NPMS API to using registry.npmjs.com, suggested by Bengtsson when we ran into issues with the NPMS API. Heskja changed the fetching of packages from triggering on search to triggering on load.	
Package fetch/unpkg fetch	Heskja implemented fetching of the ‘package.json’ files of each package fetched earlier through unpkg.	Heskja
Package fetch/list.csv fetch	Heskja implemented a fetch of the “list.csv” file from the whitelist GitHub repository.	Heskja
Package fetch/component list	Make the components available in two arrays in the application state: Array1: Contains all fetched components Array2: Contains all searched components	Heskja
Package list	Heskja implemented initial functionality to display a list of the fetched packages. Bengtsson implemented styling, including package information, keywords, information about publishing, NPM, and GitHub logo. Heskja refactored the package list to a component list and displayed some information. Tverdal changed from component list input into component cards displayed in a grid, styling, verification marker. Bengtsson enhanced the design of the cards and added the <package>/<export> part.	Heskja, Bengtsson, Tverdal
Pagination	Bengtsson implemented initial pagination functionality for fetched packages using react-paginate. Tverdal implemented offset-pagination, so that packages are fetched only when they need to be displayed, instead of all at once. Heskja refactored pagination to work with components instead of packages and integrated it with filters and search. Tverdal changed the pagination as a sticky footer.	Heskja, Bengtsson, Tverdal
Documentation	Documentation in the Readme file	Heskja

Table 32. Website work distribution.

Feature	Description	Implementer
List with verified packages	Created the “list.csv” file containing package identifiers and versions and added it to the repository.	Bengtsson
Automated verification workflow/pipelines using GitHub actions	The workflow running the whole certification process triggered by the incoming pull request. Uses the CLI for the certification checks.	Bengtsson
Documentation	Documentation in the Readme file	Heskja, Bengtsson

Table 33. Whitelist work distribution.

Feature	Description	Implementer
CLI bootstrapping	Bootstrapped the CLI and created initial functionality, including arguments and other infrastructure.	Bengtsson
Package.json keyword verification.	Checks if the package’s package.json file contains the “dhis2-component-search” keyword.	Bengtsson
Package.json “dhis2ComponentSearch” verification.	Verification of the dhis2ComponentSearch property. Checks the following: ‘language,’ ‘component array,’ ‘component name,’ ‘component description,’ ‘component export,’ ‘dhis2Version’.	Bengtsson
Package exports	Checks that the components stated as exported in the ‘package.json’ file are exported by the package.	Bengtsson
Pull request verification	Before running the actual certification, verifies that the pull request can be trusted. Includes the following checks for security reasons: That the event is a pull request, that only the “list.csv” file was changed by the pull request, validates the package identifier and version, that the git repository property is correct.	Bengtsson
ESLint	Bengtsson implemented the initial ESLint functionality. Tverdal improved on the ESLint functionality.	Bengtsson, Tverdal
Git repository cloning	Functionality to clone the git repository for the packages.	Bengtsson
Unit tests	Bengtsson implemented unit tests for all features in the CLI except ESLint and npm audit, which Tverdal implemented.	Bengtsson, Tverdal
Npm audit	Functionality to run npm audit on the package.	Tverdal
Documentation	Documentation in the Readme file	Heskja, Bengtsson

Table 34. CLI work distribution.