NTNU

Norwegian University of
Science and Technology

Self-Adaption in Lego-Mindstorm Train Control System using OSGi

Alexander Svae

Master of Science in Communication Technology
Submission date: June 2016
Supervisor: Peter Herrmann, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Self-Adaption in Lego-Mindstorm Train Control System using OSGi

Student: Alexander Svae

Problem description:

Public transportation systems are becoming increasingly important in today's society, especially in big cities. Due to the dynamic behavior of the vehicles in operation, the system ability to adapt to spatial and temporal changes is critical.

The Department of Telematics has a Lego-Mindstorm model railroad layout on which the trains and junction plates can be operated using software created by our IDE Reactive Blocks. In a project assignment, a control system was developed to allow the trains to operate without an external coordinator. Furthermore, in parallel with this thesis, another student is working on modulate and improving the existing system.

We want to develop a module that allows the system to handle changes of contextual conditions. The extension needs to be able to respond quickly and adapt at runtime to changes due to the dynamic behavior mentioned above.

We will use parts of the ideas presented by Amir Taherkordi et al. in their paper Scalable Self-Adaptation in Mobile Cyber-Physical Systems. The paper proposes a scalable self-adaptive framework design to be used with Mobile Cyber-Physical Systems (mCPS) to enable Raspberry Pi based DiddyBorg robots to respond to spatial and temporal changes at runtime. To control code adaptation the OSGi framework is proposed. The OSGi framework is a service-oriented component-based framework that adds a dynamic module system to Java.

At first, the system will be analyzed and a design of the module will be developed. Then, implementation options will be discussed and the module will be implemented. When the system module is running, its performance and correctness will be tested.

Responsible professor: Peter Herrmann, ITEM

Supervisor: Peter Herrmann, ITEM, Amir Taherkordi, UiO

Abstract

Public transport has become an important part of everyday life in today's society. With the advances in information and communication technology has ITS become an important research field. Because of this, we have in recent years seen an increasing number of autonomous transportation systems. These autonomous systems operate in dynamic environments where unpredictable events may occur. These events may affect the vehicle's ability to operate safely. It is therefore essential that these systems have a way to reason and react to these events. Since the vehicles often operate under high speed is it important that adaptation to these events can happen quickly and while the system is running. This paper presents an adaptation module design intended for autonomous trains that operates on a Lego Mindstorm based model. The module's task is collect contextual information and use this information to adapt the trains behavior accordingly. To enable this adaptation during runtime is the Java based framework OSGi used. The module uses a state machine that is implemented with the State Design Pattern. This state machine is used to as the context reasoning component for the adapter. OSGi based services are used to facilitate retrieval of the contextual information and to perform the adaptation actions. The module is implemented using the modular development tool Reactive Block and was tested on the Lego Mindstorm based train model mentioned above with good results.

Sammendrag

Offentlig transport er blitt viktig del av menneskers hverdag i dagens samfunn. Med de siste års fremskritt innen informasjon og kommunikasjons teknologi har ITS blitt et viktig forskningsfelt. Ved hjelp av denne forskningen har vi i de senere årene sett et økende antall autonome transport system. Disse autonome systemene operer i dynamiske miljøer der uforutsigbare hendelser kan skje. Disse hendelsene kan påvirke kjøretøyenes evne til å operere på en trygg måte. Det er derfor essensielt at disse systemene har en måte å analysere og reagere på disse hendelsene. Siden kjøretøyene ofte operer under høy fart er det også viktig at adaptasjonen til disse hendelsen kan skje raskt og mens systemet kjører. I denne oppgaven presenteres et adaptasjon modul design beregnet på autonome tog som operer på en Lego-Mindstorm basert modell. Modulens oppgave er å samle inn kontekstuelle data og bruke denne dataen til å adaptere togenes atferd deretter. For å muliggjør denne adaptasjonen mens toget kjører er det Java basert rammeverket OSGi benyttet. Modulen tar i bruk en tilstandsmaskin implementert i henhold til State Design Pattern. Denne tilstands maskinen er ansvarlig for kontekst resoneringen i modulen. OSGi baserte tjenester blir også brukt til å forenkle innhenting av den kontekstuelle informasjonen samt å utføre adaptasjons tiltak. Modulen er implementert ved hjelp av det modul baserte utviklingsverktøyet Reactive Block og ble testet på den Lego-Mindstorm basert tog banene modellen nevnt ovenfor med gode resultater.

Preface

This thesis was conducted by Alexander Svae during the spring of 2016 at Norwegian University of Science and Technology (NTNU), department ITEM. The idea behind this thesis originated from professor Peter Herrmann.

I would like to thank my supervisors Peter Herrmann and Amir Taherkordi for all their guidance throughout this thesis. I will also thank my co-student Henrik Heglund Svendsen for his invaluable input.

Contents

List of Figures

List of Tables

List of Glossary

Callback function	A callback function is a piece of executable code that is passed as an argument to other code, which is expected to execute the code at some convenient time..
Enum	An enum type is a special data type that enables for a variable to be a set of predefined constants.
JSON	JavaScript Object Notation is a lightweight data-interchange format that is easy for humans to read and write..
GSON	Gson is a Java library that can be used to convert Java Objects into their JSON representation and vice versa.

List of Acronyms

AMQP Advanced Message Queuing Protocol.

ESM External state machine.

IDE Integrated Development Environment.

ITS Intelligent transportation system.

JAR Java Archive.

LDAP Lightweight Directory Access Protocol.

NTNU Norwegian University of Science and Technology.

UML Unified Modeling Language.

XML Extensible Markup Language.

Chapter 1
Introduction

Over the last years have public transportation systems become an essential part of peoples everyday lives, specially in bigger cities. With the advances of information and communication technologies has ITS become an important field of research. ITS systems aims to provide innovative solution to improve the efficiency, performance, safety and security of transport systems[Par10]. One of these solutions is the introduction of autonomous vehicles into public transport systems. A critical property for these vehicles is the ability to maintain safe operation. Due to the dynamic operating environment of the vehicles, their ability to adapt to spatial and temporal changes is critical in order to maintain this property.

1.1 Problem outline and scope

NTNU has in the last years increased its research into ITS, specially with the establishment of its ITS Lab[NTN16]. The lab has numerous ongoing projects concerning developments in the ITS sector. One of these projects is the development of an fully autonomous train system running on a Lego-Mindstorm based railroad model.

Autonomous vehicles are reliant on sensor information in order to recognize contextual changes. To maintain safe operation it is essential that the vehicles can use this information to adapt their behavior accordingly.

The goal for this thesis is to design and implement a module that allows the autonomous trains to adapt to contextual changes. These changes can be software and hardware defects, environmental properties and they will affect the way the train operates. Furthermore, these changes can take place while the vehicle is operating at high speed. This mandates that the adaptation must be done quickly and at run time. These requirements demand an efficient context reasoning system in order to guarantee those properties.

The proposed architecture should serve as an example of how context awareness and reasoning can be implemented into an autonomous system. Since this module is operating in an event driven environment is Reactive Blocks be used to handle synchornization and concurrency.

An important part of the adaption is reconfiguration and modifcation of code running on the system. To be able to do this at runtime is the module Java based framework OSGi proposed.

The adaption module will use ideas from the self-adaption framework presented by Amir Taherkordi et al. in their paper *Architectural Virtualization for Self-Adaptation in Mobile Cyber-Physical Systems*[AT]. In the problem description the name *Scalable Self-Adaptation in Mobile Cyber-Physical Systems* used of the paper. The reason for this was the paper went through a revision that lead to its name being changed.

1.2 Methodology

The work done in this thesis was conducted in four phases. First phases consisted of a litterateur study, mainly focusing on the technologies used by the module. As far as the knowledge of the author goes does it not exist a working adaption module for an autonomous railroad system using Reactive Blocks and OSGi. Due to this was limited time spent investigating existing solution with the exception of the paper mentioned above. The technology study was focused on the OSGi specification, as the author was already familiar with Reactive Blocks. OSGi is a massive framework that contains lots of functionality. The author had some basic knowledge about how the frameworks in theory, but had little experience in using it in practice.

Next, was the train control system where the module was being implemented into analysed. In parallel to this thesis was fellow student Henrik H. Svendsen conducting his master thesis that consisted of improving the train control system. Due to this was a conversation with Svendsen held about what these changes would be. With this information was the system, or in this case, the module requirements found.

Next phase was making an abstract design of the module and its components. Since Reactive Blocks is a graphical development tool, it is natural to merge the design and implementation phase together. The way this was done was that and the abstract design was made and then drawn up in Reactive Blocks. Then was each component of the module implemented in a prototyping fashion [Wik16e], meaning that the components were developed in iterations. When the module was fully implemented was performance tests conducted.

1.3 Limitations

As developing a working adaption module is time consuming work, this paper will not look into efficiency factors like battery consumption or system load. Nor will it look at the configuration options that OSGi provides.

As with the sensor reconfiguration we will only look at software reconfiguration and not hardware reconfiguration. The reason for this is that the hardware and its control software used in this thesis was implemented by a co-student, Henrik H. Svendsen in his master thesis[Sve16].

Unfortunately was it not possible to test the adapter module with a complete train control system. Therefor was a simpler version of the control system used in the tests presented in Chapter 10.

1.4 Terminology

In this thesis we will use the term *adapter* to refer to the adaption module. When referring to the *framework* we mean the OSGi framework running on the trains. When referring to the *system* we mean the train control system running on the OSGi framework. The terms *method* and *function* is used interchangeably.

1.5 Structure of the thesis

Chapter 2 goes through background information that is useful to understand the following chapters.

Chapter 3 presents information gained from analysis of existing train control system and its infrastructure. Form this analysis was the requirements for the module derived.

Chapter 4 show the high-level design of the module and goes through its the components.

Chapter 5 describes how communication is implemented and handled by the module.

Chapter 6 contains information about how the sensors are implemented in the system. This work was not done by the author, but it is included in this thesis as it influences the adaption module.

Chapter 7 explains how sensors are handled by the module and how their information is gathered. This information plays a significant role in allowing the trains to be aware of their context.

Chapter 8 describes how context reasoning is done by the module.

Chapter 9 shows how the module components are combined to form the adaption module.

Chapter 10 show the results obtain by the tests that was ran on the system.

Chapter 11 contains the discussion about the result from the previous chapter as well as general thoughts about the module. It also presents some ideas for further research that can be done with the module. The chapter also contains a conclusion that sums up the work done in this thesis.

Chapter 2
Background

2.1 The Lego-Mindstorm Train System

The Department of Telematics has a Lego-Mindstorm model railroad layout. The model has been used in several theses and is now running a control system developed by Henrik H.Svendsen [Sve15]. The control system allows trains to operate autonomously without an external coordinator and is implemented with the help of Reactive Blocks.

The railroad model consists of tracks and intersections. The model also has two types of physical entities, namely trains and a set of point switches controlling intersections. The switches are connected to a EV3 brick[Gro16] which is running software that can be used to control the them. The trains running on the model has a set of sensors that it uses for self-localization as well as a motor controller. The system uses a topic based AMQP exchange over Wifi for its communication. Messages are sent wireless to a Raspberry Pi implementing a RabbitMQ message broker that is directly connected to a Wifi router residing inside of the lab.

To enable coordination between the trains do each train have an internal representation of the railroad layout. The layout of the railroad has been modeled in a Lego Modeling tool called BlueBricks. The BlueBricks software can generate XML file based on these models. The trains have a parser module that can take in this XML file and convert it into an internal representation of the railroad track.

To avoid collision is a two-phase commit protocol used by the trains to reserve track legs that it must travel to reach its destination.

2.2 Related work

Amir Taherkord et al. paper *Architectural Virtualization for Self-Adaptation in Mobile Cyber-Physical Systems* proposes a self-adaption framework for mCPS units operating in a cooperative environment. The framework addresses the issues related

to the complex modeling of and adaptation to highly dynamic location and time aspects of cooperative mCPS. The framework uses a virtualization design technique, named Virtual Adaptation Framework (VAF), to provide a unified application-level view to adaptation requirements. Contextual conditions are defined as a set of rules by the VAF and the BeSpaceD tool-set analyzes the rules at runtime. The analyses are used to guide the reconfigurations of mCPS.

BeSpaceD has the ability to automatically deduce spatiotemporal analysis at runtime, which makes it an eligible candidate to reason about the current context of applications. Logical quantifiers can be applied to specify and check the existence of a spatiotemporal condition in a specification, or to prove that a certain property holds for a distinct time and space area. Furthermore are algorithms and tools such as an external SMT solvers proposed to help resolve geometric constraints such as overlapping of different areas in time and space.

2.3 Reactive Blocks

Reactive Blocks is a developer plug-in for the Eclipse IDE that combines model based engineering together with the Java language in order to make efficient event driven applications. The plug-in provides a graphical drag and drop interface where developers composes so called *building blocks* together to make complete applications. Behind the graphical interface lies a powerful automated compiler that generates complete and executable code from the building blocks. In addition, will all blocks be formally analysed and verified so that interaction errors do not occur [AS16c].

Reactive Blocks is not a substitute for conventional programming, rather it is a tool that takes care of the concurrency and synchronization parts of an application in a understandable and efficient way. Advanced business logic such as algorithms should still be implemented in code [AS16c].

The graphical representation provides an abstract view where synchronization and concurrency logic is abstracted the way. This representation is especially useful in collaborative projects where developers work on separate parts of an application. With this abstract view can developers get a basic understanding of an application's or component's behavior and its capabilities without having to go through complex code.

Reactive Blocks has built-in support for OSGi and BitReactive, the company developing Reactive Blocks, is also part of the OSGi Alliance [AS16b].

In the following, will we explain the fundamentals of Reactive Blocks and go to through some features that were used in this thesis. The complete documentation of Reactive Blocks can be found at their homepage [AS16h].

2.3.1 Building Blocks

Building blocks is reusable software modules that are composed of a behavior diagram, a block contract called *External State Machine* (ESM) and Java code. The must common way to define a block's behavior diagram is with a UML style activity diagram. The activity diagram is built up by *token flows* represented by edges, *block operation* and *activity nodes*. There are two types of token flows, control flows and object flow, with the latter being able to carry data. Activity nodes are used to control the token flow in a block. Examples of activity nodes are timers, forks, merges and event reception nodes. A full list of all, including description of the nodes can be found at [AS16a]. Each block do also have a set of *pins*. These pins can be either incoming or outgoing and is connected to the token flows inside of the block. The pins allow tokens to be passed into or out of the block.

The ESM describes the abbreviated interface behavior for the block. It states which tokens that can pass through certain pins in a given state, in addition to show potential state transition for a given token flow.

Each block does also have a corresponding Java class file where the blocks operation, variables and more is defined. The class file is just a regular java file with the only exception that methods also can be accessed from the block. For example, when a token is passed to the operation **parseMessage** in Figure 2.1, the corresponding method in the java file will be invoked.

There are though some restrictions one has to consider when programming with Reactive Blocks [AS16f].

- Every method accessed by the block must have a unique name, even though Java allow for methods with same name but different method signatures.
- If an exception can occur inside a method it must be handle inside the method, usually done with a try/catch clause.
- Methods should not block or wait. Since Reactive blocks is event driven it is important that blocks stay responsive and is able to handle events. If a method has to block or wait do notification patterns exist that can be used. In our system we use separate threads to run waiting or blocking methods and event receptions to propagate events generated by these threads to the blocks.
- Reactive blocks provides a special method named **sendToBlock(String signalname)**. When using the method a token will be generated and sent to a event reception node residing inside the block. The signalname parameter defines which event reception node the token should be sent to. For example,

in Figure 2.1 will the use of sendToBlock("SPEECH_READY") send a token to the event reception node named SPEECH_READY inside the Speech block.

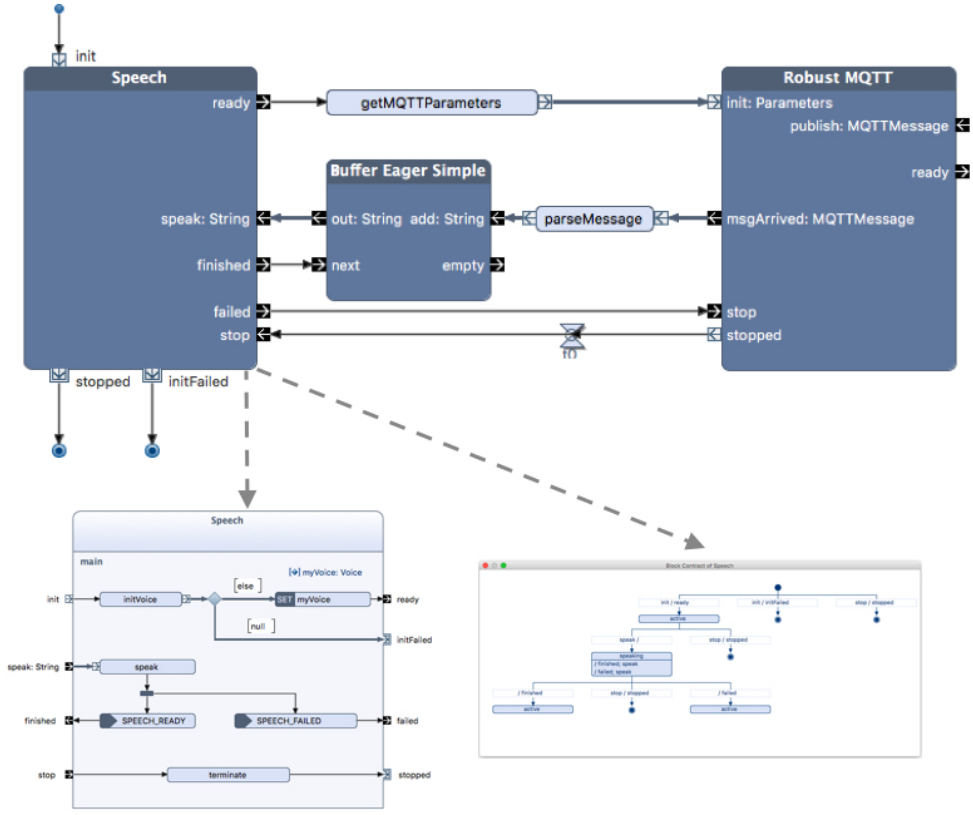

Figure 2.1: A Reactive Blocks example application, showing an application block (top), the inner activity diagram of the building block Speech(bottom left) and its External State Machine (bottom right). The Speech and Buffer Eager Simple blocks are provided by Reactive Blocks

In Figure 2.1 we can see an example of a simple Reactive Block based application. At the top part of the figure are three *building blocks* composed together inside of an *Application block*. An *Application block* specifies an application and is required to run an application. The block type is similar to a *building block* with the exception that it does not have pins or an ESM. Instead, it has initial nodes to start the application and activity final nodes to terminate it [AS16i]. In the bottom left corner of the figure we can see the activity diagram for the **Speech** block and at the bottom right corner is its ESM.

2.4 OSGi

'The OSGi technology facilitates the componentization of software modules and applications and assures remote management and interoperability of applications and services over a broad variety of devices. Building systems from in-house and off-the-shelf OSGi modules increases development productivity and makes them much easier to modify and evolve. ' - *The OSGi Alliance* [All16c]

The OSGi specification, from now on referred to as OSGi is a open standard specified and maintained by the OSGi Alliance. OSGi provides a dynamic modular framework for the Java platform allowing for dynamic component based systems [Wik16d]. The framework is divided into three main layers that we now will explore.

The aim for this section is to give you a basic knowledge of OSGi. We will not go through all the nuances of the framework, for that one can look up books like **OSGi in Action** by Richard S.Hall et al. [HPMS11]. Most of the information presented in this section is taken from this book. The OSGi Javadoc reference was also used[All16f].

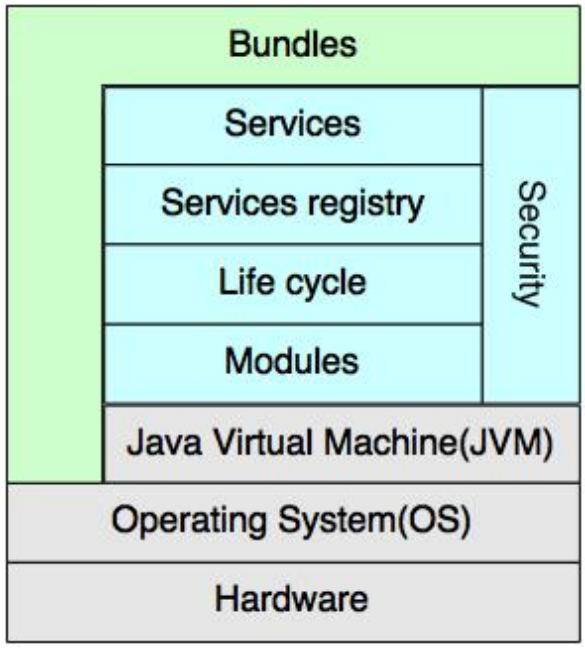

Figure 2.2: Overview of the OSGi

2.4.1 Module layer

OSGi introduces the concept of bundles. A bundle can be look upon as a module for your system. It is much like a regular JAR file with the addition of extra information in its *manifest file*. This information grants powerful advantages in comparison to regular jar files. With bundles, one can explicitly declare which packages are visible externally. This is done by declaring *exported packages* in the manifest file. This improves the encapsulation of code as declaring a class public, which is very common, no longer means that external packages can access it without it being explicitly declared in the *manifest file*. The same is done with bundles and packages required by the bundle. These specifications is also used by the framework to manage and verify bundles, making up for Java's leaking dependency handling.

2.4.2 Life cycle layer

The *module layer* provides class loading for bundles, but with only this layer cannot a bundle be managed. In order to do this one must use the *life cycle layer*. The life cycle layer defines an API for bundle management. The API provides methods to install, start, update, stop and uninstall bundles. Since the life cycle layer operates on bundles, one can manage separate bundles without taking done the rest of the system. The layer does not mandate any particular mechanism of interacting with the life cycle API. This gives great flexibility in how one wants to use the layer. Internally does the life cycle layer grant bundles access to their execution context called **BundleContext** and thereby the opportunity for the bundles to interact with the framework.

2.4.3 Service layer

The *service layer* introduces the concept of services. Simply stated is a service a bundle that provides, as the name implies, services that other bundles can use. Services expose a service interface to its consumers. It is used as a contract between the service provider and service consumer stating what methods, constants and enums that are provided by the given service.

One problem that can occur when using interfaces to define services are that there could be multiple service implementations available, all only exposing the same interface. Thous, as a consumer, one cannot know which actual service implementation is being used. A simple example is that you have two services, providing encryption of data. One of them provides a strong encryption, but is slow. The other one is fast, but provides a weaker encryption. So, as a consumer of the service, one cannot easily choose the wanted service implementation. Of course, one could add a method or constant fields that states the different properties of the service implementation. The problem with this is that this is hard-coded information

and therefore cannot be changed during run-time. In addition, one would have to go through all available implementations and then find the desired one.

OSGi solves this problem with their *Service Registry*. It acts as a mediator between service providers and consumers. The Service Registry also allows for finer grained control over services since all OSGi services is registered to this registry and with their registration, they can provide a set of properties that describes their characteristics. When a consumer wants a particular service implementation, one can provide the desired service properties and the registry will find a service that has matching properties, solving the issue mentioned above.

As a service consumer one has not direct access to the service implementation. This may seem strange, but it assures that the framework have control over the services and follows the dynamic nature of OSGi. This also allows on to dynamically bind bundles together at run-time.

2.4.4 Service Tracker

The Service Tracker is a utility class provided by OSGi that allows for easy monitoring and access to services in a well-defined way. Its constructor takes in a bundle context, the service type you want to track and optionally a *customizer* object. Defining the service that will be tracked can be done in three ways. One can provide it with a **ServiceReference**, a class or interface name or an **LDAP** filter [All16d]. LDAP search filters is a human-readable string format described in RFC 1960 [For16].

ServiceTrackerCustomizer

The **ServiceTrackerCustomizer** is an interface with three methods. The three methods corresponds to the addition, modification and removal of a service and the Service Registry will trigger these methods when one of these events occurs for the service being tracked by the ServiceTracker [All16e].

ServiceReference

Every service registered to the Service Registry has a unique **ServiceReference** object. This can be obtained via the registry and used to examine service properties and to access the service.

2.4.5 Event Admin Service

The Event Admin Service is a compendium service that allows one to distribute event in a publish/subscribe fashion to other bundles running on the framework. An event consists of a topic, which is a structured string, and event properties. When an event is sent to the service it will distribute it to all *Event Handlers* subscribed to the topic.

An Event Handler is a class implementing the Event Handler interface[All16a]. It is registered to the service registry as a regular service but with a special property that allows the Event Admin to recognize it. This property contains the topic of which the Event Handler wants to receive events from. Figure 2.3 is an example of how the topic structure works.

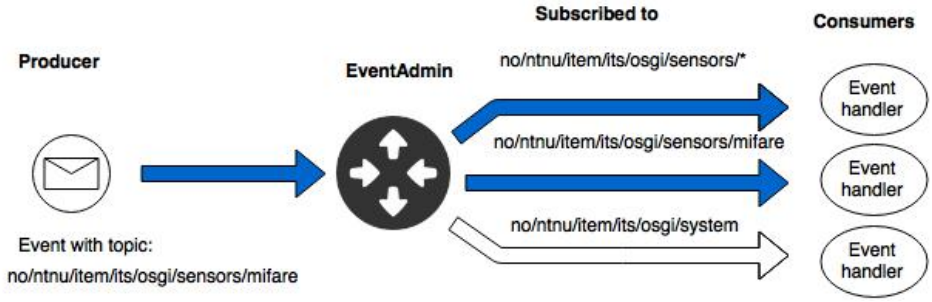

Figure 2.3: An example of how a event is routed to a system using the OSGi EventAdmin service

2.4.6 Apache Felix File Install

The Apache Felix File Install is a OSGi management agent. The user sets up a directory where all non-core system bundles are placed and the File Install bundle will monitor this directory. If a new bundle is added to the directory, it will be installed into the framework and started. It updates a bundle if you update the bundle file in the directory and if the bundle file is deleted it will stop and uninstall the bundle. This automation goes well together with Git. One can update the bundles running on the framework simply be pulling down the updated bundle files from a code repository and have the File Install bundle manage them[Fou16c]. All this can be done during runtime.

2.5 Git

Git is a distributed version control system that can be used to manage code projects [Git16a]. In this thesis is Git together with the File Install bundle mentioned above is used to distribute and manage bundles running on the trains.

All code developed in this thesis is located in public Git repositories on GitHub.

List of repositories:

– The runtime environment used by the trains [Gitb].

– The code developed in this thesis [Git16b]

– The code for the modified sensor control software used in this thesis [Gita].

2.6 Eclipse

The code in thesis was developed in the Eclipse IDE (4.5.1). To be able to run Reactive Blocks inside of Eclipse one can follow the steps described here [AS16e]. OSGi is already integrated into Eclipse but in order to run it together with Reactive Block some set up is required. Instruction on how this is done can be found here [AS16d].

2.7 AMQP

Advanced Message Queuing Protocol is a open standard application layer message queuing protocol designed to support a variety of messaging application and communication pattern in an effective way. It provides a rich feature set that include support for different communication pattern, flow control, message delivery guarantees, authentication and more [Wik16a].

AMQP version 0.9 is based on the AMQ model. The model consists of three components; exchanges, message queues and bindings. Exchanges are the routing mechanism of the model and routes incoming messages to their respective message queues. Message queues are FIFO-buffers that hold messages on behalf of one or more consumers. In order for the exchange to know which message queues a message should be sent to are bindings used.

Exchanges can be of different types, one of them being topic based. When using a topic based exchange will each message queue subscribe to one or more topics. A topic is a structured string delimited by dots. When a message is sent to a topic will all message queues subscribed to this topic receive the message [SA16].

2.8 RabbitMQ

RabbitMQ is open source polyglot message broker implementation. It supports multiple messaging protocols like AMQP, STOMP, HTTP and MQTT.

RabbitMQ also provides a client library that can be used to interact with a message broker. At its top-level do the library contain four packages [Inc16a].

Channel is an interface that is used to do protocol operation, like sending messages

Connection is used to manage the channels and register connection lifecycle event handlers.

ConnectionFactory is used to instantiated connections and is used to configure various information like hostname, port number, username and so forth for the connection.

Consumer is used to all messages to be pushed to receiver instead of having the receiver explicitly request it.

The client library is OSGi ready as it is also a OSGi bundle [Inc16b].

Chapter 3
Analyse of the existing infrastructure

System analysis is a cornerstone in the development of all types of systems. The analysis helps to form an understanding of how a system works, what its goals are and its properties.

Since the adaptation module to be developed shall be implemented into an already existing infrastructure, is it natural to divide the analysis into two parts. In the following we will parts of the existing infrastructure and control system relevant to the adaption be analysed. Having an understanding of the system's capabilities and limitations are factors that influences design decisions made later. In addition, you get a better understanding of how the adapter will affect the system.

After that, we will look at the requirements for the adaption module.

3.1 Improvements to the train system

In Section 2.1 we briefly described the infrastructure that the trains will be running on and its control system. As mentioned earlier is Svendsen conducting his master thesis on improving this system. In particular, he is looking into how to improve self location for the system [Sve16]. As apart of this work is most of the train's Lego-Mindstorm based hardware changed out. In this new iteration of the system are Raspberry Pi's [Fou16a] used as the processing unit on board the trains. In addition, has he introduced a new set of sensors and software to utilize them. In particular, a new improved color senor has been added along with a RFID/NFC reader chip and a magnetometer. The control software for these sensors was developed to work with OSGi.

A decision had to be made on whether the adaption module should be design to be used with the improved version of the system. This was discussed with both the supervisors and Svendsen. Using the old version of the system meant that we would work on a finished system, reducing the risk of unexpected problems. However,

there are some major drawbacks with this version of the system. The system was developed without OSGi in mind, meaning that the entire system had to modified. In Svendsen paper [Sve15] he discuss performance issues with the EV3 brick. This performance issue is also a deal breaker, as introducing the adaption module together with OSGi would increase the system load further.

3.2 Analyse of the train system

The goal of the train control system is to allow trains to operate autonomously in a safe manner. In the current state of the system, each train will reserve a track leg when it travels from one station to another. This means that while the trains are traveling there should not be other trains in close proximity.

There are already formed a communication system that can be used to communicate with the trains and the adapter. Furthermore, each train will have a set of sensors. These sensors will be a crucial for the adapter since this is where information about contextual changes will come from. As said earlier shall the trains be able to operate without a central coordinator, this must be taken into account when designing the module. The adaption module can fail and it is therefor important that the system is still able to operate to a certain level without it.

One of the improvements being done by Svendsen is the introduction of OSGi into the system. OSGi offers a powerful toolkit that the module can and should utilize.

3.3 Analyse of the adaptation module

Trains operate in highly dynamic environments where contextual properties may unpredictably change. The goal of the adapter is that it should make trains able to respond to these changes in a safe and efficient manner. Time is a crucial factor for the system, it is therefor important that the adapter reacts in a timely manner. This also means that the adaption must occur during runtime.

Another fundamental characteristic of the adapter is that it must always make correct unambiguous decisions. Wrong decisions can lead to unwanted behavior and may have serious consequences. Information about these changes will primarily come from the sensors as mentioned earlier and it is thus necessary to find a way to use them without affecting the rest of the system.

The train control system and its infrastructure is not static, but constantly evolving and improving. It is therefore beneficial if the adapter is able to deal with these changes and utilize them. This applies particularly to the introduction of new

sensors, as this is a natural progression for the system. Ideally, this should not lead to major changes in the code and it should be intuitive how one can incorporate new sensors into the adapter.

Although trains are autonomous situations may arise where is remote operator must take control of the adapter. As previously mentioned there is communication architecture already set up. Adapter should use this and allow for an operator to remote control it.

The architecture of the module should be made in a way that allows it to be used in similar systems. A way to reach this goal is to follow the high cohesion and low coupling design principal.

3.3.1 Discussion with Henrik H. Svendsen

Initially it was discussed with the developer of the train control system how the train adapter module best can be implemented into the system. As mention in Section 2.1 is the train control system implemented in Reactive Blocks, therefor is it natural that the train adapter is developed with Reactive Blocks. The adapter can be implemented in smaller modules distributed throughout the system or as one single building block. Due to the train control system already being complex, it was agreed that the train adapter should be implemented into a single block making it easier to include into the system. In addition, with having the adapter contained in one block would allow for it to be used in other systems without major changes.

3.4 System requirements

3.4.1 Non-functional system requirements

1. Correctness - When a given event occurs should the correct action always be carried out.
2. Timeliness - All events must be processed and action should be executed in a timely manner.
3. Non-intrusive - The adapter module should not interfere with the existing control system, meaning that the control system should be able to operate without the adapter.
4. Adaptability- The adapter should be able to adapt to contextual changes during runtime.
5. Flexibility - The adapter should be able to work in similar type of system without extensive modifications.

6. Modifiability - The design of the module should allow code to be modified while the adapter is operating.
7. Configurability - The adapter should provide an easy way to configure properties about the trains, environmental properties and so forth.

3.4.2 Functional requirements

1. The adapter module must be able to receive sensor input from all relevant train sensors.
2. It must also be able to reason about the input and make decision based on them.
3. The adapter must be able to change sensor properties without interfering with the rest of the system.
4. The adapter must be designed in a way that allows modification of code related to sensor, behavior and properties without having to restart the module.
5. It must be possible to control the module from a remote location.

Chapter 4
Adaption module design

We will in this chapter show the high-level design made for the module and go through each of the components.

4.1 High-Level Design

In the system requirements specified in Section 3.4, it was stated that the adapter module should be able to be used in similar systems. A good way to do this is to allocate sensor management in a separate block. This will decouple the sensors from the contextual reasoning and wrap sensor specific characteristics inside of one component. The same arguments can be used about communication. The design consists of four components, where one of them is used to wrap the three other components into a single component. The design is presented in Figure 4.1.

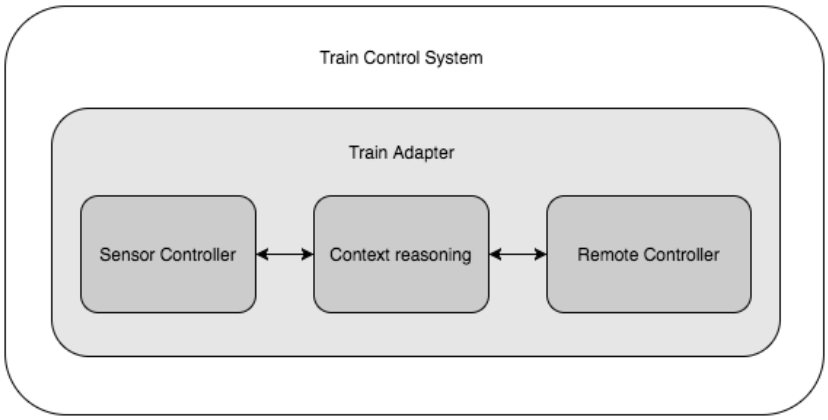

Figure 4.1: High-level design of the adaption module

- **Sensor Controller** is responsible for handling and controlling the train sensors. It must ensure that all sensor readings is received by the adapter as well as provide a way to reconfigure the sensors. The Sensor Controller should also provide the Context reasoning component with notification about sensor status changes.

- **Remote Controller** is responsible for all communication to and from a remote the adapter. The Remote Controller should take use of the existing communication infrastructure and provide a well-defined way to interact with the adapter module. Moreover, a message format must be defined that both the adapter and the remote operator can understand and use. A set of commands should be specified so that the remote operate knows what actions are available to him. The remote operator can use these commands to controller and override the context reasoning component. The component should also be made in a way that also it to be used for train-to-train communication.

- **Context reasoning** is responsible for processing the sensor inputs and reason about them. The component must keep track of contextual properties, access necessary resources and react to changes in a correct and efficient way. It should also be able to make use of the other components residing inside of the adapter. By this, we mean that is should be able to receive commands sent to the Remote Controller and act upon them. In addition, it should provide a way for the Sensor Controller to send sensor readings and notifications to it.

- **Train Adapter** contains the three sub-components mentioned above. The component is responsible for providing a way for the sub-components to cooperate. The component will define the external API of the module.

Both the Remote Controller and the Sensor Controller are passive component in that they only will respond to event internally. The Context Reasoning component on the other hand will take action that changes the trains properties and behavior.

4.2 Services

In addition to components is a collection of services needed in order for the Context Reasoning model to do its job. The services are developed without Reactive Blocks. There are two reasons for this. Reactive Blocks is based on tokens being passed around to the different blocks. This is not a problem when all modules can be coupled in one application block. The problem arises when you have modules that is residing outside of the system block. Since OSGi services are not implemented directly in a system block, but rather is accessed through the service registry the notion of token passing disappears. The second reason is that the services are mostly comprised of

advanced logic and do not require the synchronization and concurrency handling that Reactive Blocks provides.

These services will be presented in the following chapter together with a complete module design and implementation.

Chapter 5
Communication

The author and Svendsen had a conversation regarding how communications were handled within the system. The control system used a single block that the author made in a project assignment to handle communication. One problem noticed with this was that it was difficult to route incoming messages around the system. With the introduction of OSGi we could take advantage of its service platform to make communication handling easier. It was therefore determined early on that a communication service would be developed. Having a separate communication service ensures that the system uses the same means of communication. It will also make it easier to update the communication protocol, as you only need to create a new service implementation. It is also possible to change to a different message queuing protocol like MQTT it relatively ease.

5.1 Communication protocol

AMQP was chosen as the communications protocol. The reason for this is mainly that it was already used with success in Svendsen system[Sve15]. In Section 2.7 is a brief description of AMQP. To ease the development of the service was the RabbitMQ client library used for interaction with the message broker. The reason RabbitMQ was used over other client libraries like ActiveMQ and QPID was mainly that the authors had used the library in an earlier project and that the library is already been prepared for OSGi as it is also a OSGi bundle.

5.2 Service requirements

Although the service is mainly used in the train system, it was designed so that it can be used in other systems. The service demanded four things in order to work.

1. The system must use OSGi.
2. The message broker must be able to handle version 0.9.1 of AMQP protocol.

3. The broker must support topic based exchanges.

4. Message sent must either be strings or be possible to serialize to JSON objects.

5.3 Implementation

Technically, two communication services were implemented. Both uses the same core code, but are designed for different purposes. The service uses AMQP version 0.9.1 and topic based exchanges, which are both described in Section 2.7. All classes and interfaces that are referred to in this section can be found in Section A.1

5.3.1 TrainAMQPSendService

TrainAMQPSendService is a simple service that allows one to send but not receive messages. Its service interface only offers two methods, one to connect to a broker and one for sending messages. Connecting to a broker is done by providing the service with an AMQPProperties object. The object contains information like broker hostname, port number, username and so forth. The service requires only one service consumer to connect to the broker. After that can all bundles use it to send messages to the broker. The idea behind this was that one could set up the service when the system started and then use the send method directly elsewhere in the system. To simplify this event further was a ServiceTrackerCustomizer made that automatically connected to the broker used by the trains.

To send a message one simple provides a string containing the topic on which the message should be sent to and a serilizable object. This service was mainly used to communicate with the adapter during testing.

5.3.2 TrainAMQPService

TrainAMQPService is a more advanced service. In addition to letting on send and receive messages it also gives you more fine-grained control over the connection. The service provides two ways to connect to a broker dependent of the level of control one wanted.

When using openConnection(AMQPProperties properties) will the service check if a connection factory with the given AMQPProperties exist. If the factory does not exist will a new factory be initialized and a connection to the broker will be opened. When the connection is made is a TrainAMQPConnection object be returned.

With the TrainAMQPConnection can the service consumer open new TrainAMQPChannels. TrainAMQPChannels is needed in order interact with the broker and contains methods for sending and receiving of messages and to (un)subscribe

to topics. The openConnection method is used when one wants to allocate multiple channels to one connection. If only one channel is needed can the openChannel(AMQPProperties properties) method be used instead. It will automatically create a connection and open a channel for the user, meaning that the user do not have to be concerned about connection management.

When a consumer receives the channel it can immediately start using it to send messages via the provided send(Object message, String topic) method. As with receiving it is a bit more complicated. Since reception of a message is triggered by an external source one most decide if messages should be pushed to or pulled by the receiver. Having the consumer pull for messages is the simplest way. For example could the service offer a getMessage() method that would return all received messages since last invocation. The obvious flow with this is that would have to periodically call the service. This is especially a bad fit with time critical systems like the one presented in this paper. In addition can it lead to lots of unnecessary calls eating up resources. A better way to solve this is to push messages to the consumer via a Callback function. This is done by using the TrainDefaultConsumer utility class provided by the service. It allows the consumer to set a method that it wants to be invoked when a message is received. The method must be defined with a Function<AMQPMessage, Void> object. A benefit with having the consumer set the method to be invoked with a Function object is that it can be changed during runtime. The reason for this is that the service will call the run the method inside of the Function object of the TrainDefaultConsumer and this Function object can be changed by using the provided setFunction(Function<AMQPMessage, Void> function) method.

5.3.3 AMQPMessage

AMQPMessage is a utility class is used by the service and by the RemoteControl block, as we will see later. It acts as a multi purpose container object. When a message is received, a set of properties is also received containing information about the message. The AMQPMessage gather all this information so that is contained in one object. When someone uses the RemoteControl block it takes in a AMQPMessage object where the topic and the message is contained within the object.

5.4 The RemoteControl block

Some will argue that the communication service could be used directly in the Context reasoning components. In addition to the arguments made in Section 4.1 will implementing communication directly into the ContextChecker increase the complexity of the block, which is already complex as it is. So, a communication block was made tailored to work with the adapter. Its activity diagram is shown if Figure 5.1

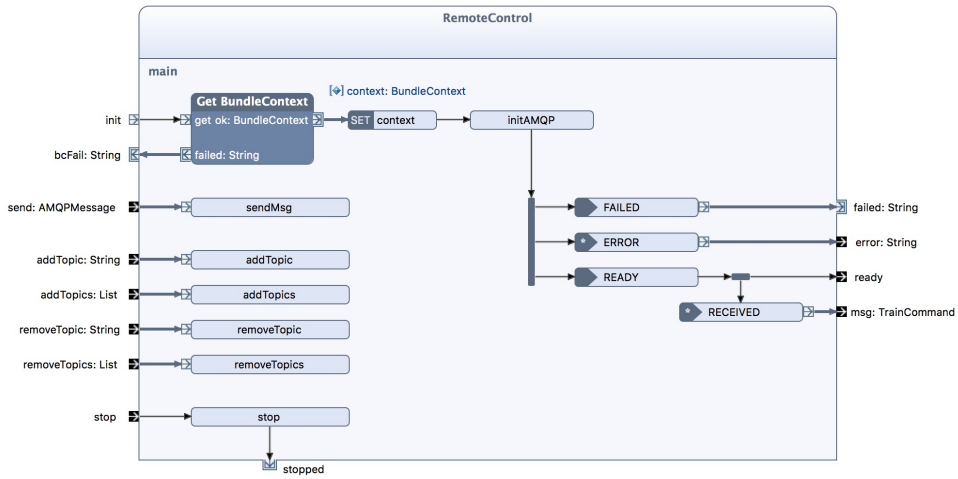

Figure 5.1: The RemoteControl block is responsible for all remote communication with the train adapter

The block uses the TrainAMQPService described above to set up a two way communication to a broker. A problem encountered during the development of the block was how received messages were going to be passed to the block. To be more specific, when a message is sent to the adapter it will be received by the service, and not the block itself. Fortunately does the TrainDefaultConsumer allow one to provide it a Function object that is invoked on message reception. This means that whenever a message is sent to the adapter, a method defined in the Java class associated with the block is executed. We now have gotten the message to the block, but we still need to send it to the *RECEIVED* event reception node in order for it to leave the block. Reactive Blocks already provides a method that does exactly this, namely the *sendToBlock* method that was described in Section 2.3.1. Listing 5.1 shows the Function object used in the RemoteControl block.

```
private Function<AMQPMessage, Void> getCallbackFunction(){
            return new Function<AMQPMessage, Void>() {

                    @Override
                    public Void apply(AMQPMessage t) {
                            TrainCommand cmd =
                                deserilizeBody(t.getRawBody());
                            logger.debug("Received message");
                            if(cmd == null) return null;
                            sendToBlock(received, cmd);
                            return null;
                    }

            };
    }
```

Listing 5.1: The RemoteControl uses this method to generate a function object to be used with the TrainAMQPService

Figure 5.2 shows process of connecting to a message broker and how the Function object generated by the function is used.

5.4.1 Message reception

When a message is received by the block it will deserialize the message using the GSON library into a TrainCommand and send it to the ContextCheckes as shown in Figure 5.3. The TrainCommand is a utility class to be used between a remote operator and the adapter. The class defines a way for how one can control the adapter. It is implemented as a tuple, with an Enum describing what action should be taken and optionally a value.

5.4.2 Sending a message

To send a message must one provide an AMQPMessage object to the block containing a topic and a message object. The block will simple then provide this information to TrainAMQPService that will send the message.

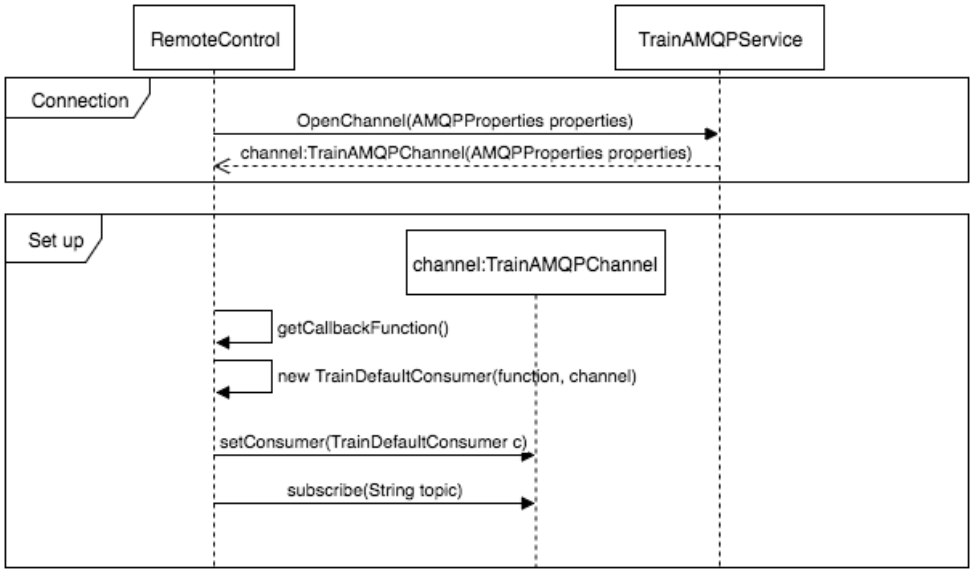

Figure 5.2: The sequence diagram for the set up of the AMQP connection inside of the RemoteControl block

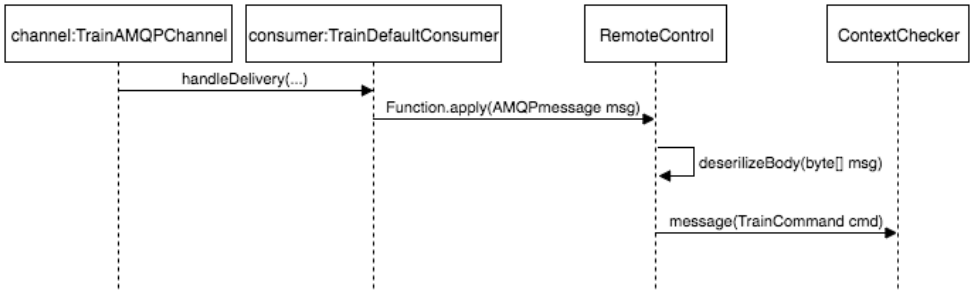

Figure 5.3: The sequence diagram for the describing how the message reception is handled

Chapter 6
Sensor software

Before we can go into the workings of the sensor controller, it is useful to know how the sensors is realized in the system. As mentioned before is the software controlling the train sensors was developed by Svendsen. We will in this chapter go briefly through how data is retrieved and published to the system. For a more thorough explanation please see Svendsen master thesis [Sve16].

Type	Hardware
Color	TCS34725 [Ada16b]
NFC	PN532 [Ada16c]
Magnetometer	MAG3110 [Ada16a]

Table 6.1: A list of the sensor hardware running on the trains

6.1 SensorSchedulerService

The SensorSchedulerService is a simple service that lets you schedule tasks to be run periodically. The scheduler used by the service is defined by the ScheduledExecutorService and uses a Java ScheduledThreadPoolExecutor object to handle the scheduling. Each of the tasks will be run on a separate thread managed by the ScheduledThreadPoolExecutor.

To schedule a task one must provide an object implementing the Runnable interface, a initial delay which can be set to 0 for immediate execution and the period duration in microseconds between each execution. The service has developed by Svendsen to be used by the publishers, this is explained below in Section 6.3. The author expanded the service to enable it to remove already scheduled tasks. To do this must the already scheduled task be provided and a Boolean value indicating if the task should be interrupted if it is currently running.

6.2 Sensor implementation

Each sensor has a bundle containing methods for retrieving raw sensor date. This bundle is registered to the framework as a service. To publish the sensor data to the system do each sensor also has a separate publisher bundle.

6.3 Sensor Publishers

The sensor publishers job is to publish sensor data to the system. It does this by utilizing the Event Admin service described in section Section 2.4.5. All publishers are registered as services to the Service Registry with a common interface named *PublisherService* and a property describing the type of the sensor it is publishing data on behalf of.

When a publisher is started, it will generate a task that access the sensor, retrieves its raw data, processes the raw data and publish it to the Event Admin. This task is sent to the SensorSchedulerService for it to be executed periodically.

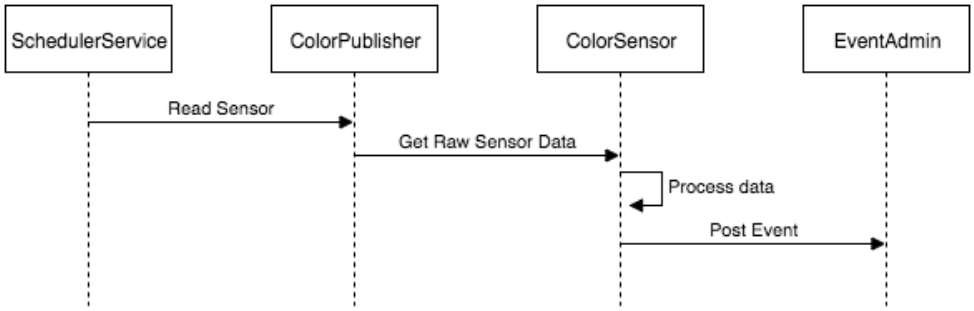

Figure 6.1: The sequence diagram for how a sensor data is published to the system

As with the SensorSchedulerService added the author some features to the publishers. These features is related to changing the time between readings, called publish rate and support for doing a single sensor reading was also added. The author added methods that notified the system if a sensor changed its status.

With the current implementation of the sensor software, it is not possible to reconfigure a sensor directly; instead must its publishers be used. This limits what the adaption module can do with regards to sensor reconfiguration.

6.4 Sensor Problems

Henrik H. Svendsen discovered some issues related to the sensor hardware during his work. These issues will impact what the adapter is capable to do and how it can interact with the sensors.

The first problem he discovered was that all sensor shares the same data buss. In order to get values from the sensors one has to explicitly read its values. The problem with this if a multiple sensors have their values read frequently, it can cause the sensors to block each other, meaning that events can be lost. This is especially critical for the NFC sensor, as missing a NFC beacon will mean that context information is lost. It was therefor decided that the best way to circumvent this was to use the color sensor as the *main* train sensor and have its events trigger the other sensor.

There where also an issue with the magnetometer sensor being to sensitive. For example would the electrical wiring in the lab effect the sensor. This limited the usefulness of the sensor and it is only used to demonstrate certain features of the adapter.

Chapter 7
Sensor handling

Sensor information is essential to gain an understanding of a train's contextual properties and forms the basis for the context reasoning. In this chapter will we explain have the sensor handling is solved in the adapter.

7.1 Tracking the sensors

In Chapter 6 we described how the sensors are implemented into the system and that each publisher is registered as a service with a common interface. We can utilize this to monitor the publishers. This is done by the CustomeServiceTracker block.

7.1.1 The CustomeServiceTracker block

The CustomeServiceTracker is flexible block that can be used to track services. It uses a ServiceTracker described in 2.4.4 to track the service since it is robust and thread safe. . A LDAP filter is used to define what service should be tracked. LDAP filters can be used to state what class or interface one wants to track. In addition to this do they also allow service property requirements. This means that that one has more fine-grained control over which service one wants to track.

When the block is started, it takes in a string representing the LDAP filter. It will then get the BundleContext with the help of BitReactive's Get BundleContext block[AS16g]. When the BundleContext is acquired is the string validated and used to create the LDAP filter. This again is used to set up a Service Tracker. Using a ServiceTracker will only give access to already registered services. To be able to get notification about services registration, updates and unregistrations a ServiceTrackerCustomizer is added to the tracker. Inside the ServiceTrackerCustomizer is code that will send notification to the block when a tracked service is registered, updated or unregistered.

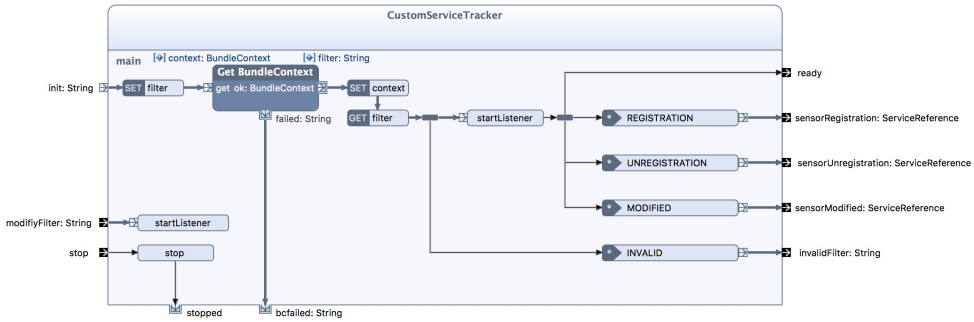

Figure 7.1: The activity diagram of the CustomServiceTracker block

When a service's state is changed will the block forward its ServiceReference instead of the service object. The reasoning for this is that the block can not use generics since the class or interface of the tracked service is not decleared when the block is started. A benefit with sending the ServiceReference is that it gives access to the properties provided when the service was registered. This information is used by the SensorController as we will see later. Lastly, since the block uses LDAP filters and only sends ServiceReferences out does the block not need to have access to the interface or class it is tracking.

7.2 Receiving sensor readings

Now that we can track the sensors publisher we need to retrieve the information published by them. The sensor publisher uses the Event Admin to push sensor events to the system. In Section 2.4.5 we described how event handlers are used with the Event Admin to receive outgoing events. We will in in this section describe how these Event Handlers is implemented. We will use the term *handler* to describe a class implementing the EventHandler interface.

7.2.1 SensorHandlerController service

To simplify the management and allow for runtime updates of handlers was a service containing the handlers made. A question that came up when mocking up the service was whether a controller should be used to administer the handlers. If the service was implemented without a controller then all handlers would have to be registered to the framework. Although OSGi allows one to filter out services based on their service properties it would make the bundle activator unnecessary complicated. Furthermore, this would require the Sensor Controller to implement possible complex logic in order for it to retrieve the correct handler for a sensor. For these reason was a controller

included into the service. The controller implements a set of methods that can be used to retrieve handlers.

In order for the Context Reasoning component to be able to understand the sensor data it must define how it except different sensor readings to be formatted. This is done by having a sensor reading class for each individual sensor type. The sensor handlers job is to receive the sensor data from the publishers and convert them in to an appropriate reading object. In most cases is this a trivial task, as shown in listing A.2. The benefit of using the handlers as converters is that you can change sensors and/or their output format without changing the SensorController, Context Reasoning component or the train states described later. Instead, one can simply add a new handler for the sensor or modify the existing one. For example if the train has backup sensors that gives out data in different format, one only need to update the handler for that sensor.

7.2.2 Sensor reconfiguration

Sensor reconfiguration is one of the actions the adapter can take. As trains may have different types of sensors, it will be unwise to implement the reconfiguration logic inside of a block. We will therefor again use a service to handle this.

SensorConfiguratorController services

The SensorConfiguratorController service is responsible for reconfiguring train sensors. The service is similar to the SensorHandlerController service in its architecture. It has a controller that is registered to the system as a service. Each sensor will have its own configuration class that is responsible for that sensor. In order to have a defined way to reconfigure sensors is a utility class named *SensorReconfiguration* used. The class contains properties regarding what sensor that should be reconfigured along with what should be done and optionally a value.

To reconfigure a sensor must a SensorReconfiguration be provided to the controller. The controller will route this object to the correct configurator that will execute methods in the appropriate publisher.

As we mention earlier do all sensors publisher implement the same interface. However, it is likely that the sensors will have custom properties that only can be configured on one specific sensor type. A way to handle this is to let the publisher implement two interfaces, one being the PublisherService A.3 and a specialized interface defining the configuration option for a given sensor.

7.3 The SensorController block

The SensorController block is a intermediary between the sensor publishers and the ContextChecker 8.5. Its main purpose is to ensure that the ContextChecker receives all sensor readings published to the system in addition to do sensor reconfiguration on behalf of it.

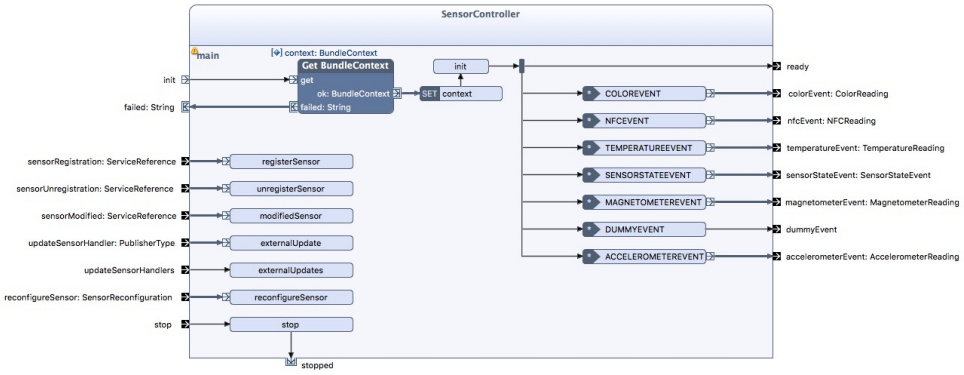

Figure 7.2: The activity diagram of the SensorController block

When the SensorController is started it sets up two ServiceTracker 2.4.4 to gain access to the SensorConfiguratorController and SensorHandlerController service. A ServiceTrackerCustomizer is added to the ServiceTracker related to the SensorHandler service. The custiomizer is need in case the SensorHandlerController service is down when the block is started and the SensorController is not able to get its handlers. Without a customizer, one would have to periodically check if a SensorHandlerController service has been registered. With a customizer, this is no longer a problem as it allows to add a function that will be executed when a tracked service is registered to the framework. An added benefit with this is that if the service is updated will the SensorController automatically update all the registered handlers.

The block is connected to the CustomServiceTracker so it can receive events regarding publisher registration, updates and unregistration. This information is used to maintain a set of handlers. It does this by using the SensorHandlerController service to get appropriate handlers 2.4.5 for all sensor publisher registered to the framework. In order to keep control over which handler is registered the different publishers is a last containing all the parings used. This list is used when a publisher is unregistered so that the block can remove and unregister the handler associated with that publisher from the framework. It is also used when updating a handler.

In Figure 7.3 can we see how the sensor controller is related to the other blocks and services. Green indicates services and blue indicates blocks.

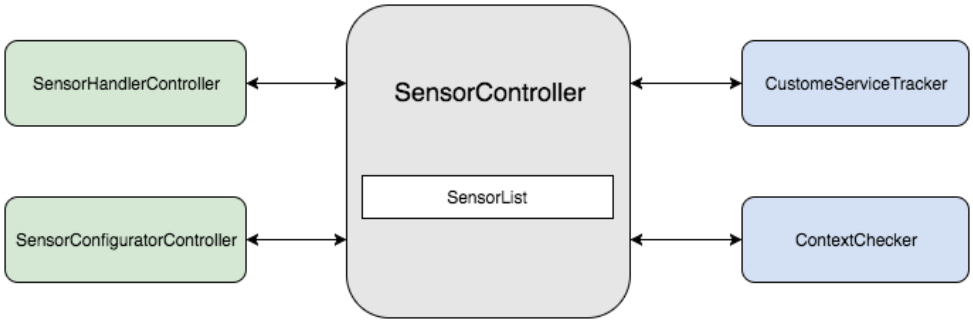

Figure 7.3: Overview of the SensorController

EventReceiver

A problem that came up when developing the SensorController was that the handlers did not have a way to send sensor readings to the SensorController. This was solved by introducing the EventReceiver interface shown in listing A.4. The interface contains a series of method that can be used to forward the different sensor readings from the handlers to the block. With this, can the handlers can take in an EventReceiver object as a parameter in its constructor and use the methods declared in the interface. The SensorController implements this interface and uses the sendToBlock() method to get the event to the event receptors residing inside the block. When the Sensor Controller is setting up an event handler, it sends a reference of itself to the SensorHandlerController services that passes it to the constructor of the handler.

Sensor registration events

When a sensor publisher is registered to the framework will the CustomServiceTracker be notified. It will get the ServiceReference for the publisher and send it the SensorController. The SensorController will then look up the PublisherType for the sensor and send this to the SensorHandlerController. The controller will return the appropriate handler to the SensorController, which then registers it to the framework.

Figure 7.4: The sequence diagram for how the SensorController retrieves the appropriate EventHandler when a train sensor is registered to the system

Chapter 8
Context modeling and reasoning

So far have we explained how sensor readings are received inside the adaption module with the use of event handlers and how these handlers are controlled. The last piece of the puzzle is to use the actual readings and reason about them. In the following, we will explain how this is done.

8.1 Keeping track of the train

In order to make the adapter context aware it needs a way to model the trains contextual properties. There is numerous ways to go about this, but for this component, we opted to use an object to contain the information. The object, named TrainInfo, contains variables for a train properties like its speed, heading, current state, location and so fourth. To access the properties is variable getters and setters used.

8.1.1 Train restrictions

In Section 8.2 we will describe how states are used to dictate a trains behavior. As one of the goals for this system is to allow for code reuse, it is not desired that a train state contains hard coded values related to specific train types. To tackle this problem is a service containing characteristics and restrictions concerning the physical train using the module proposed. The service, named TrainRestrictionsChecker inhabits methods to be used by the train states to determine if the trains properties is valid in regards to its context. The service can for example be used to check whether a train is operating at a valid speed for the zone it is traveling in. Other information, like sensor information is also contained in this service.

The service exposes a common interface, with each train type having its own service implementation of the interface. To keep the code cleaner does each train type implementation have two classes. The first class is an abstract class encompassing the properties. The class follow the constant interface pattern described at [Wik16c].

The second class is comprised of methods used to access these values as well as check certain properties.

With this service can now the same train state implementation be used by different train types running in the same environment.

8.1.2 Map properties

In Section 2.1 we explained that the train control system uses a BlueBrick map for navigation. As of now do these maps only contain identification information, like intersection id's and how the track legs are connected. In order to make a train's location influential for its behavior we wanted to introduce zones and other location information into the system. In order to do this was a service containing this information made, named MapChecker.

The MapChecker provides a means to get information regarding the zones for which the train operates. This information can for example be zone classification, like City zones where stricter restrictions applies. These restrictions can then be used to adapt the trains behavior accordingly. In practice, will the train use this service upon entering a new zone to gain information about its classification. The service also holds a simplified model of the railroad. This can be used to predict what the next zone should be in case of a sensor error. The service is built up in the same way as the TrainRestrictionChecker in that each railroad map has one class file containing the its properties and a class containing methods to get and check these properties.

8.2 Context reasoning

Now that we have a way to track and check a train's properties all we need is to find a way to reason about them. An important question for this thesis was how context reasoning should be done.

A simple approach is to list up a set of logical clauses and then go through them on an event. There are several problems with this approach. For one, every time an event occurs you have to go through many unnecessary clauses. Gathering the clauses into bulks concerned with certain events can solve this problem. Secondly, the complexity of the logical clauses would increase significantly as the systems complexity increases. Especially making sure that clauses are not conflicting can be a major challenge. Lastly, code additions would require refactoring of the whole code.

Amir et. al [AT] adaption framework uses the BeSpaced tool-set for its context reasoning. The BeSpaced is a implemented in Scala, which the author has little

knowledge about. Due to the limited time frame of this thesis it was decide that using BeSpaced would be a to comprehensive task. After discussions with the supervisors it was decided that using state machines could be a good solution for the context reasoning. The implementation of the state machine used by the adapter is based on the State Design Pattern [Jos15, Chapter 18].

State Design Pattern

"The State Design Pattern allows an object to alter its behavior when its internal state changes" - *Java Design Pattern, chapter 18*

The State design pattern is a behavioral pattern, meaning that it encapsulate behavior in an object and delegates request or event to it[Wik16b]. In the State design pattern is the behaviors implemented in states which all implements a common interface. The pattern fits well with systems where an objects behavior changes in accordance with its internal state during runtime.

The object containing the state, often referenced to as the *Context object*, can alter its behavior dependent on its internal state by only replacing its state object. This replacement or *state transition* is triggered by the state them self and it is their responsibility to keep the *Context object* in the correct state. Each state is defined with a regular java class implementing a common interface. Having states implemented in separate classes significantly simplifies additions and modifications of states. It is important to notice that the state objects only contains the context behavior that is specific for that state and not the overall behavior of the *Context object*.

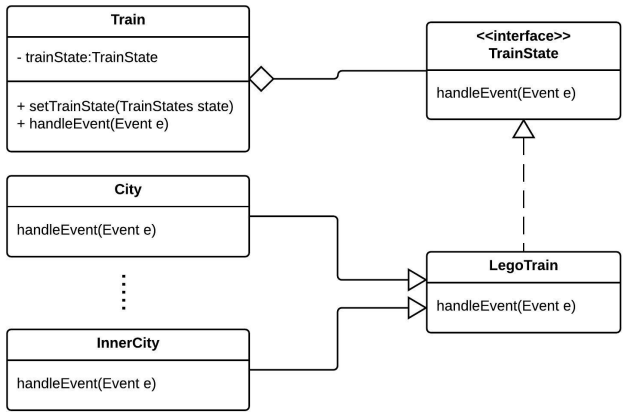

Figure 8.1: Structure of a state pattern system

In listing 8.1 is a simple example of the state design pattern. The *Context* object, in this case a Train, has three fields. The first one is to keep track of its current state. The two other ones is to be used to change train state as we can is in the stop() method inside the Running class.

```
public class Train{

    private TrainState state;
    private TrainState running;
    private TrainState stopped;

    public Train(){
        this.running = new Running();
        this.stopped = new Stopped();
        this.state = running;
    }

    public void setState(TrainState state){
        this.state = state;
    }

    public void start(){
        state.start();
    }

    public void stop(){
        state.stop();
    }

    public TrainState getRunning(){
        return running;
    }

    public TrainState getStopped(){
        return stopped;
    }

}

public class Running() implements TrainState{

    private Train train;

    public Running(Train train){
```

```
        this.train = train;
    }

    @Override
    public void start(){
        System.out.println("Already running");
    }

    public void stop(){
        System.out.println("Stopping train");
        train.setState(train.getStopped());
    }
}
```

Listing 8.1: An example implementation of the state design pattern

In the code example listed above one may notice that all train states are initialized when the *Context* object is created. This means that it is not possible to change the states during runtime. This in itself is not a problem but we could do even better if we utilize the tools given to us by OSGi. Firstly, instead of keeping reference to all the different TrainStates we could have a controller object that keeps track of the states for us. In listing 8.2 is the modified code. We have taken out all references to the different TrainStates and instead we use a TrainStateController to get the different TrainStates. An enum, named States, is also introduced to be used by the TrainStates to tell what the next TrainState should be.

```
public class Train{

    private TrainState state;
    private TrainStateController controller;

    public Train(BundleContext context){
        controller = new TrainStateController();
        controller.open();
        this.state = running;
    }

    public void setState(States state){
        this.state = controller.getState(state);
    }

    public void start(){
        state.start();
```

```
    }

    public void stop(){
        state.stop();
    }

}

public class Running() implements TrainState{

    private Train train;

    public Running(Train train){
        this.train = train;
    }

    @Override
    public void start(){
        System.out.println("Already running");
    }

    public void stop(){
        System.out.println("Stopping train");
        train.setState(States.Stopped);
    }
}
```

Listing 8.2: An example implementation of the state design pattern using a state controller

So far have we decopuled the TrainStates from the adapter module, but we still do not have a way to update the states during runtime. This is where OSGi comes in. If we implement the TrainStateController as a OSGi Service we are suddenly capable of changing TrainStates without stopping the adapter module. Listing 8.3 show the modified code that takes use of the service. Another benefit with this approach is that all things state related are located in one place. This will help reduce the complexity of the adaptation module.

```
public class Train{

    private TrainState state;
    private BundleContext context;
```

```
    private ServiceTracker<TrainStateController, TrainStateController>
         controller;

    public Train(BundleContext context){
        this.context = context;
        controller = new ServiceTracker<>(context,
            TrainStateController.class, null);
        controller.open();
        this.state = running;
    }

    public void setState(States state){
        this.state = controller.getService().getState(state);
    }

    public void start(){
        state.start();
    }

    public void stop(){
        state.stop();
    }

}
```

Listing 8.3: An example implementation of the state design pattern using a state controller with OSGi

8.3 Train State Implementations

8.4 Scope of states

A question that arose when work with the state implementation was how general they should be. In simpler terms, what dictates a trains behavior running in our system?

8.4.1 Location based states

As we talked about earlier in 8.1.2 can the trains location affect its behavior. For example, in certain zones should certain sensors be read more often in order to detect events faster. By location we mean the *zone* the train is operating.

8.4.2 Sensor based states

In addition to its location do also a trains available sensors influence its behavior. How a sensor's data is reasoned about is dependent on its intended use and responsibility. If a sensor should fail its responsibility must be delegate to the others sensors so that the train still can maintain safe operation.

8.4.3 Train operation status based states

The last factor that can affect a trains behavior for our system is it operational status. For example if a train is used it may ignore certain sensor readings as they are no longer relevant.

8.4.4 Hierarchical states

To reduce code redundancy hierarchical state is used. Hierarchical state uses inheritance between states so that common behavior does not need to be implemented in all state, rather do the state classes inherits the behavior from a parent class.

8.4.5 TrainState Interface

As we mentioned above must all states in the state machine implement a common interface. For the states used in the adapter was the interface shown below used.

```
public interface TrainState {

        public void colorUpdate(ColorReading color);
        public void accelerationUpdate(AccelerometerReading acc);
        public void magnetometerUpdate(MagnetometerReading reading);
        public void temperaturUpdate(TemperatureReading temp);
        public void nfcUpdate(NFCReading hex);
        public void sensorUpdate(SensorStateEvent event);
        public void dummyUpdate();

}
```

Listing 8.4: The TrainState Interface

8.4.6 States

In total was nine states implemented into the state machine. An overview of the states is show in Figure 8.2. The double lined circles indicate that the state is abstract and can not be used by the adapter. The states name indicates which zone types the state is used in and the states with names ending with NFC are used when an NFC

reader is not available or has failed. The train state named Stopped is used when a train, as the name implies, as stopped or failed. More information about the states and their interaction is provided in Chapter 9 and Chapter 10.

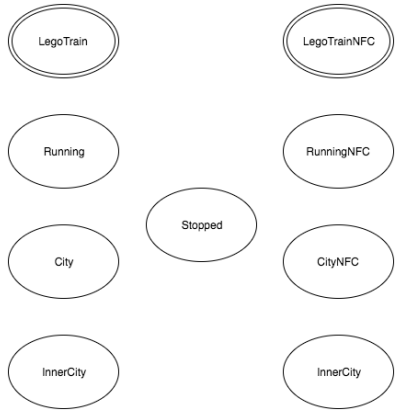

Figure 8.2: Overview of the states contained in the state machined used by the adapter

8.4.7 The TrainContext interface

The TrainContext states what methods the context object must provide in order for the states to be able to reason about inputs and take actions.

```
public interface TrainContext {

        public void setTrainState(TrainStates state);
        public TrainStates getCurrentTrainState();
        public void stopTrain();
        public void sendSpeedRestriction(SpeedRestrictionLevel level);
        public SpeedRestrictionLevel getSpeedRestrictionLevel();
        public double getSpeed();
        public TrainRestrictionsChecker getTrainRestrictionChecker();
        public TrainStateController getTrainStateController();
        public MapChecker getMapRestrictions();
        public boolean isInTurn();
        public void setInturn(boolean b);
        public void increaseSpeedForTurn();
        public void decreaseSpeedForTurn();
        public double getHeading();
        public void setHeading(double heading);
        public String getCurrentLocationID();
        public void setCurrentLocationID(String locationID);
        public Status getSensorState(PublisherType type);
        public void setSensorState(PublisherType type, Status status);
        public void reconfigureSensor(SensorReconfiguration reconfiguration);
        public void setLastSleeperColor(SleeperColor color);
        public SleeperColor getLastSleeperColor();

}
```

Listing 8.5: The TrainContext Interface

8.5 ContextChecker

The ContextChecker is responsible to bind together all the resources need by the state machine. It is this block that will receive the sensor information and make the adaptation module context aware. As we explained in 8.2 is a TrainState object used to hold the trains current state. In order for the states to interact with the block is the TrainContext interface implemented by the block. In addition to the TrainState object does the block also hold a TrainInfo object described in Section 8.1.

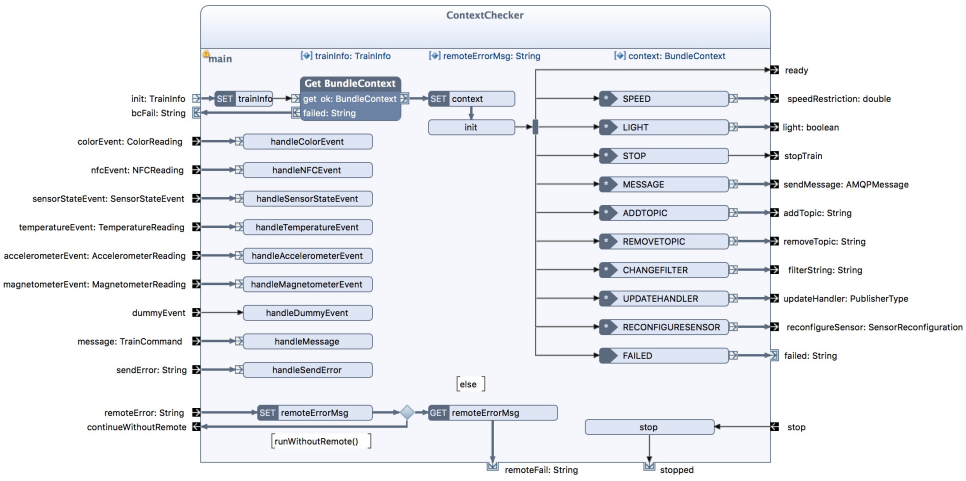

Figure 8.3: The activity diagram of the ContextChecker block

In order for the states to get access to the MapChecker, TrainStateController and TrainRestrictionChecker is ServiceTrackers set up when the block is activated. To allow both the SensorController and the RemoteControl to interact with it does the block expose a set of incoming and outgoing pins. Each of the incoming pins have a separate method for handling the data received. The outgoing pins are used to send notifications and commands to the other blocks as well as to the train.

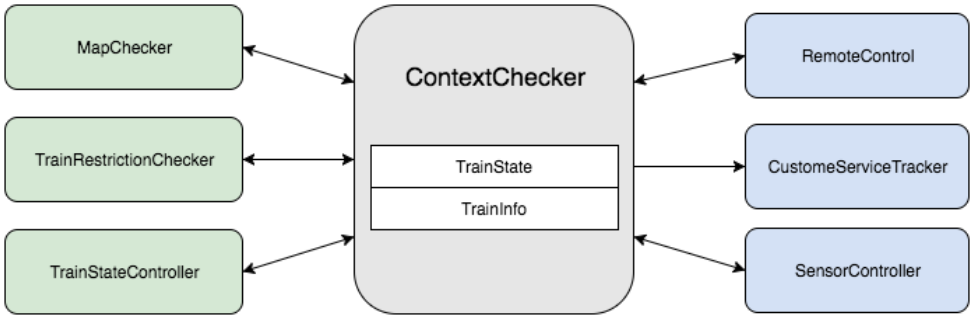

Figure 8.4: Overview of the ContextChecker block

Chapter 9
Train Adapter

This chapter will describe how the components were composed together. We will also explain how certain events are handled by the adapter.

9.1 The TrainAdapter block

The TrainAdapter block is the block that will be used by the train control system. The block connects the three components described in the previous chapters into one single block.

One of the blocks job is to ensure that the components are started in the correct order and handle any errors related to this. The initialization of the block takes

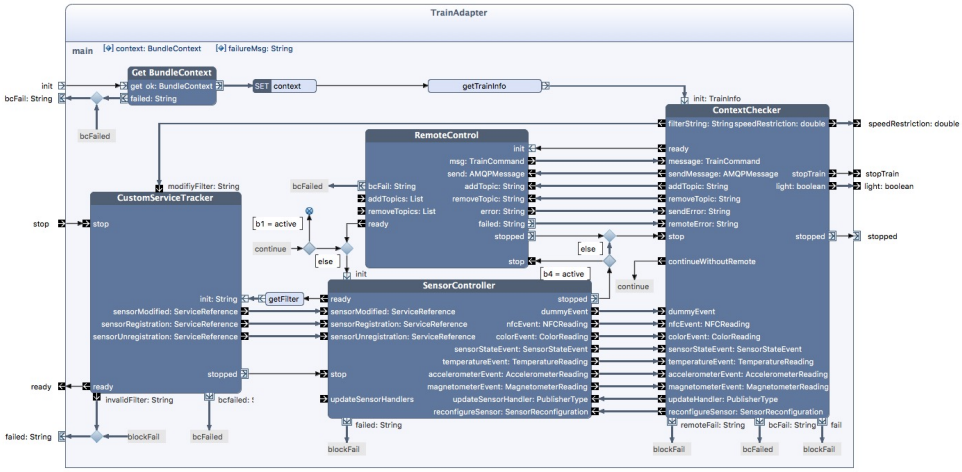

Figure 9.1: The activity diagram for the TrainAdapter block

place in four steps.

1. When the block starts up it will first collect information about the train and store these properties in a TrainInfo object A.5. This is then given to ContextChecker block in order for it to be able to start. If the start up fails, then the adapter will interrupt its start up since the ContextChecker is crucial to the TrainAdapter.

2. Second step is to establish contact with the message broker to allow remote control of the block. If a connection cannot be established will an event be sent to the ContextChecker. It checks with the TrainRestrictionChecker if the train is allowed to run without without the RemoteControl block. If the block should fail while the system is running, the same check is made. This is mainly implemented in order to be able to conduct simulated tests without a broker present. It is not recommended to allow the adapter to run without the RemoteControl block since you then have little control over the block. Ideally should the RemoteControl block try to connect to a backup broker that it can use if the primary broker goes down.

3. When a connection to the broker has been established will the SensorController be activated. Since this component is critical for the ContextCheker will the block be closed if the SensorController cannot be activated. The Sensor Controller is started before the CustomeServiceTracker to ensure that it receives all sensor registrations.

4. Finally will the block obtain a string containing an LDAP filter. This filter ensures that only the train sensors will be tracked by the CustomeServiceTracker. This filter will be given to CustomeServiceTracker so it can start. The block assumes that the filter is valid, else will the block be closed. This could be handled in a better way, but was not prioritized.

When the block is active will it operate by itself and is not dependent on inputs from other blocks. The only control you have over the block is that one can stop it.

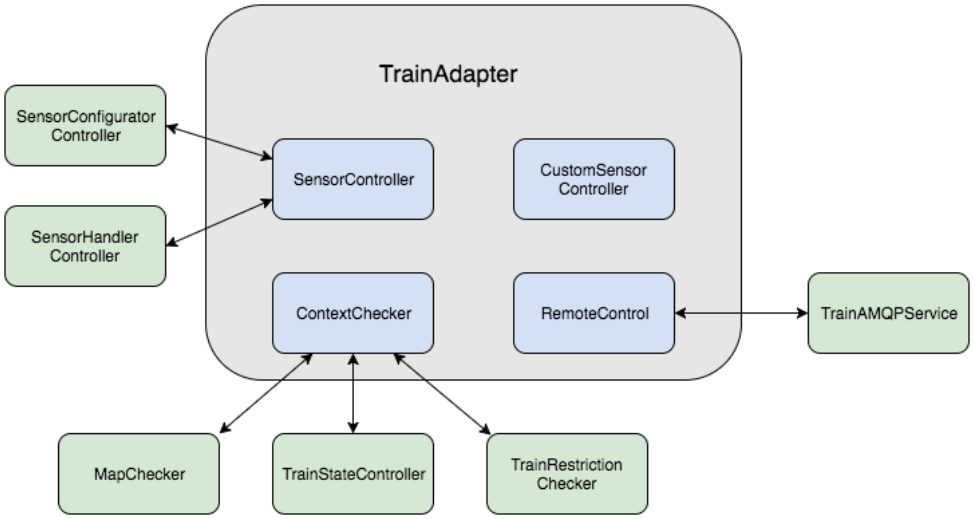

Figure 9.2: The figure shows how the blocks and services are linked together. Blue squares are used to represent blocks while the green squares represents services

We have in earlier chapters described how each individual components handles different events. As it can be hard to see how it all fits together we will in the following describe how

9.2 Processing a sensor reading

When a sensor publishes a reading will the following events be triggered. As an example is a color reading used.

1. The color publisher gets the color sensor data and sends it as an event to the Event Admin.

2. The handler responsible for the color sensor will receive the event from the Event Admin. It process the data and creates a new ColorReading object. This object is then sent to the SensorController.

3. When the SensorController receives the object will it simple forward it to the ContextChecker via the colorEvent pin.

4. The ContextChecker receives the ColorReading object via its ColorEvent pin and sends it to its current state.

5. The state will reason about the data inside of the ColorReading object and then take appropriate actions if needed.

9.2.1 Changing train state

The process of changing a train's current state is straight forwards. When an event which requires a change of state occurs will the current state use the setTrainState(TrainStates state) method defined in the TrainContext interface. The TrainStates parameter is a Enum which contains a list of all the available states for the module. The ConxtextChecker will then take the TrainStates object and send it to the TrainStateController. The TrainStateController will return a new TrainState object linked to the provided TrainStates enum. Finally will the ContextChecker set the new TrainState object as its state.

9.3 Reconfigure a sensor

1. The current train state will call the reconfigureSensor(SensorReconfiguration reconfiguration) inside of the ContextChecker with a SensorReconfiguration object as a parameter.

2. The ContextChecker send this object to the to SensorController that forwards it to the TrainSensorConfiguratorController.

3. The TrainSensorConfiguratorController routes the object to the appropriate SensorConfigurator.

4. The SensorConfigurator will then access the publisher for the sensor and call the necessary methods.

5. If the reconfiguration leads to a status change for the sensor, will the publisher send out an event with the sensors new state.

6. This event is be picked up by the event handler for the sensor. The handler will send the event to the SensorController.

7. The SensorController creates a new SensorStateEvent object and send it to the ContextChecker via its SensorStateEvent pin.

8. The ContextChecker will send the SensorStateEvent object to the train state which will in most cases update the TrainInfo inside of ContextChecker with the new sensor status.

Having the publishers send out an event after a reconfiguration is important so the rest of the system knows that the properties of the sensor has changed.

9.4 Perform a sensor reading

As we talked about in Section 6.4 is the color sensor used to trigger the other sensors. An example of this is that the NFC reader be in idle mode until a blue sleeper is passed. Whenever a blue sleeper is passed will the current train state receive the color reading as described in Section 9.2. The state will recognize that a blue sleeper has been passed and that the NFC reader should be activated. The activation will follow the step 1 to 4 of described above Section 9.3.

9.5 Handling failed sensor readings

During initial testing of the train adapter there was discovered some with the issues with the NFC reader. Theses issues is critical as missing a NFC beacon can lead to the train running with the wrong state. Thus was preventive measurements was implemented to handle this.

9.5.1 The NFC sensor was not able to read the data from the beacon

If the sensor reading is corrupted, will the train state request the MapChecker service for the excepted value for the beacon. If multiple sensor readings are corrupted, it is an indication that the there are something wrong with the sensor. In this case will the adapter assumes that the senor is faulted and change state.

9.5.2 The NFC sensor was not able to detect the beacon

The states have a Boolean variable indicating if a NFC reading has been done. Whenever a blue sleeper is passed, this variable will be set to false. It will then start a timer in a separate thread. If the train state does not receive a NFC reading when the timer expires, it will fetch the excepted value of the NFC beacon from the MapChecker.

9.6 Handling sensor failure

Situation can arise where a sensor fails. It is important that the adapter can response to this in a safe way. In the train restriction property file is the importance of each sensor defined. A sensor can either be *vital*, *important* or *peripheral*. If a peripheral sensor fails, no action is taken. If an important sensor fails it requires that the train change its behavior, meaning that a change of state must be done. In the case of a vital sensor failure will the train be stopped immediately. For the sensor available for the trains in our systems are the color sensor defined as vital, the NFC reader as important and the magnetometer as peripheral.

Chapter 10
Performance Tests

A series of test was conducted in order to test the performance of the adapter. This chapter will go through each test and present the results.

10.1 Runtime environment used by the trains

To be able to use OSGi, one must of course have an OSGi framework implementation. There are several options, most notably Apache Felix, Equinox OSGi and Knopflerfish. For the trains was the Equinox implementations used. Equinox is the reference implementation of OSGi, it is included with Eclipse and is also supported by Reactive Blocks. A list of the included bundles can be found at [Fou16d].

In addition to the framework was some additional compendium service bundles needed for the trains to work. To be able to use the Event Admin 2.4.5 is Apache Felix's implementation of the services used [Fou16b]. To allow automatic installation of new bundles and to update bundles during runtime is the Apache Felix Fileinstall2.4.6 service bundle used. The entire runtime enviroment can be found at [Gitb].

10.2 Logging results

The Equinox framework includes a logging service named Log Service [All16b]. The service provides a general purpose message logger for the OSGi environment [Fou16d]. As default was the runtime environment set up to log all messages to console, which is useful under testing. But their was also a need for custom log files for testing. A useful feature of the Log Service is that it allows one to register a LogListener to the service. The LogListener receives all messages being logged to the system. These messages comes in form of a LogEntry [All16b]. Each LogEntry contains properties that can be used to filter out wanted messages. It also contains the timestamp for when the entry was created. In the following tests is this used to time the system. A

bundle was made, named TrainAdapterLogger, that contains all the LogListeners used under the testing.

10.3 Overview

The mean reason for this testing is to see if the adapter is able to react within a reasonable time. One thing to keep in mind is that the results of the tests if specific for this system and it hardware and results may differ if ran on a similar system.

The tests were divided into two parts. First was the reaction time for NFC and color sensor tested, in other words how long it took from a sensor publishes a reading until it is received by the train state. In the second part was the fully featured adapter performance tested.

10.4 Response time on color events

10.4.1 Setup

- The trains ran on a circular track where blue sleepers was passed approximately every 3 seconds.
- The color publisher was set to do a sensor reading every 10 ms.
- The state machine used in the test contained one state which only cared about color events. When the state received a color event for a blue sleeper with would send a speed restriction command to the train.

10.4.2 Results

The result of the test is presented in table 10.1 and 10.3. All times presented in the table is in milliseconds.

From	To	Avg.	Max	Min
Publisher	Event Handler	3.08	640	<1
Event Handler	Train State	1.12	19	<1
Train State	ContextChecker	0.47	8	<1
ContextChecker	Train	0.49	10	<1
Publisher	Train	5.17	641	<1
Event Handler	Train	2.08	20	<1

Table 10.1: Response time on color events

From the results can we see that on average it takes 5 ms for the adapter to react to a sensor event. This result was better then what the author expected. However, there seems to be an issue with some events being significantly delayed from the EventAdmin. In table 10.2 is information about the time it takes form an event is published to it is received by the handler

Interval	Number of occurrences
Less then 2ms	2824
Between 2 ms and 10 ms	48
Between 10 ms and 100 ms	7
Greater then 100ms	18
Number of readings	2897

Table 10.2: Time for an event is published til it is received by the event handler

As we can see is more then 97% of the events received within 2 ms. However, there where 18 instances where it took more then 100 ms, with the longest delay being 640 ms. The most likely reason for this is that the EventAdmin can not handle all the events sent to it. To reduce this problem was could the publish rate for the color sensor be increased as we will see in the following test.

If we assume that the EventAdmin is well working and remove the 18 instance mentioned above we get even better results for the adapter, as shown in table 10.3.

From	To	Avg.	Max	Min
Event Handler	Train State	1.07	6	<1
Publisher	Train	3.09	86	1

Table 10.3: Response time on color event where the events that took more then 100ms to get from the publisher to the event handler is filtered out

10.5 Using color events to trigger NFC readings

10.5.1 Setup

- The trains ran on a circular track where blue sleepers was passed approximately every 3 seconds. Under each blue sleeper was a NFC beacon.

- The color publisher was set to do a sensor reading every 20 ms.

– The state machine used in the test contained one state. When the state received a color event for a blue sleeper it would activate the NFC sensor by following the steps described in Section 9.4

The result is presented in table 10.4

10.5.2 Results

From	To	Average	Max	Min
Color Event				
Publisher	Event Handler	3.12	364	<1
Event Handler	Train State	1.34	15	<1
NFC Event				
Train State	Start reading	0.69	5	<1
Start reading	Finished reading	118.63	164	71
Finished reading	Event Handler	4.86	416	<1
EventHandler	Train State	1.44	9	<1
Number of readings			541	

Table 10.4: Results from the test of the NFC and Color sensor

From the results we can see that increasing the publish rate of the color sensor helped with the issues related to the EventAdmin. Out of the 541 readings, only five them took more then 100 ms with the highest value being 364 ms. As with the NFC readings we can see that it takes on average less then 1 ms from the train state receives the event til it has activated the sensor. This means that the adapter performed well enough to use the color events as triggers for the NFC event.

10.6 Complete performance test

10.6.1 Setup

– The train ran on the track displayed in Figure 10.1. The colored lines represent sleepers with that color and the boxes indicates the different map zones. Under each blue sleeper was a NFC beacon placed.

– The train traveled only on the outer perimeter of the track.

– The adapter reacted to different sensor readings in accordance to the adaptation table shown in 10.5.

– The color sensor publish rate was increased to 25 ms.

– The magnetometer publish rate was between 600 ms to 680 ms depending on with zone the train was traveling in.

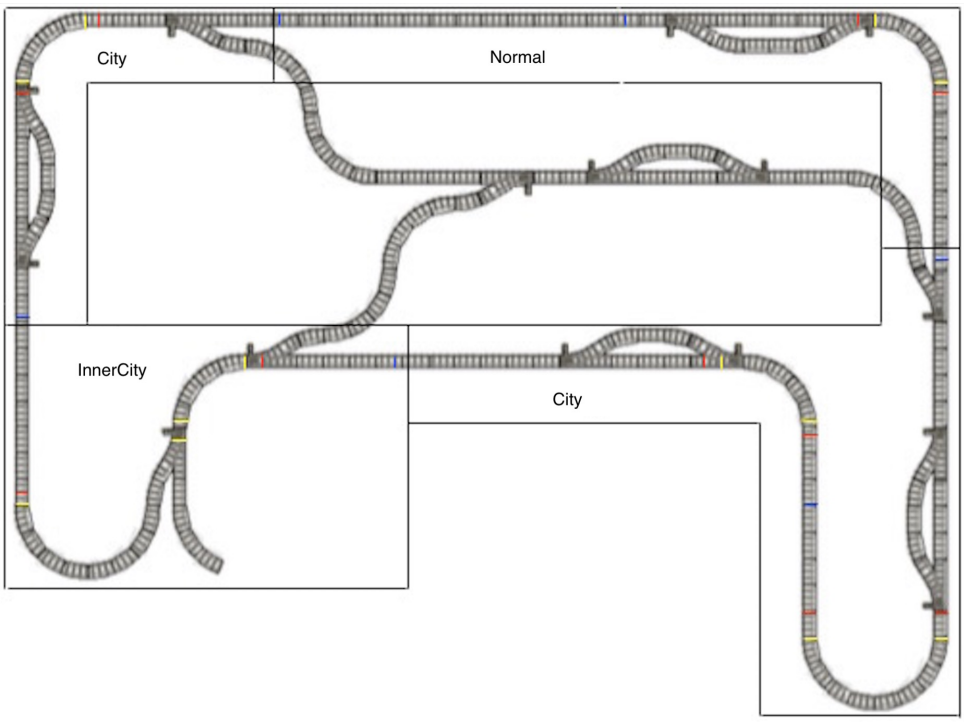

Figure 10.1: Figure of the track layout. The figure is a modified version of a Figure 6.1 found at [Sve15]. The colored lines indicate the location of a sleeper with that color. The boxes indicate track zones

State	Event	Condition	Action
Out of turn	Color	Red	Turn off magnetometer
In turn	Color	Red	Turn on magnetometer
Out of turn	Color	Yellow	Notice train about incoming turn
In turn	Color	Yellow	Notice train that it is no longer in a turn
Everywhere	Color	Blue	Read from NFC sensor
Everywhere	NFC	Location ID	Change train state

Table 10.5: Adaptation table for sensor reading used by the train adapter

The results is presented below and as before are the times in ms. All the results was obtained from the same test run.

10.6.2 Noticing trains about turns

In this test we wanted to see how long it took the adapter to notify the train about an incoming turn.

From	To	Avg.	Max	Min
Publisher	Event Handler	1.05	156	<1
Event Handler	Train State	1.02	4	<1
Train State	Train	0.47	3	<1
Publisher	Train	2.55	157	1
Event Handler	Train	1.49	6	<1

Table 10.6: Response time on color events

From the results we can see that the adapter uses on average less then 3 ms to notify the train, which is satisfactory. There is still a problem with the Event Admin, however with the further increase of the publish rate was there only two occurrences out of 287 yellow color readings where the Event Admin used more the 10 ms to send the event to the handler. This se

10.6.3 Reconfigure a sensor

This test was used to see how long time it took to perform a sensor reconfiguration. The recofiguration performed during this test was to start and the magnetometer whenever a red sleeper was passed.

From	To	Avg.	Max	Min
Publisher	Event Handler	2.79	247	<1
Event Handler	Train State	1.06	8	<1
Train State	Sensor started	0.69	3	<1
Train State	Sensor stopped	0.65	3	<1

Number of color events	Starts	Stops
234	117	117

Table 10.7: Time used to reconfigure a sensor after receiving a color reading

Again we see that the adapter performance well. It uses less on average then 1 ms from it receives the event until the sensor is started/stopped.

10.6.4 Perform a NFC sensor reading and changing state

In this test we wanted to see how fast the adapter was able to change its state when entering a new map zone.

From	To	Average	Max	Min
Color Event				
Publisher	Event Handler	0.46	10	<1
Event Handler	Train State	0.99	3	<1
NFC Event				
Train State	Beacon read	110.53	142	89
Publisher	Event Handler	0.85	4	<1
EventHandler	Train State	0.87	3	<1
Train State	State changed	0.52	3	<1
Color Publisher	State changed	114.22	148	91
Number of readings				123

Table 10.8: Time used for the adapter to change state when entering a new map zone

Again are the results very good for the adapter. One thing to notice is that the EventAdmin used at max only 10 ms to send the event to the color handler. As reason for this could be that in the track layout are the red and yellow colored sleepers fairly close, while the blue colored sleepers is not close to any of them.

Chapter 11
Discussion and Conclusion

11.1 Discussion

11.1.1 Correctness

The adapter is reliant on sensors in order for it to be aware of contextual changes. If the sensors provide erroneous information, it will affect the basis of the context reasoning. However, this effect can be mitigated by having services that can provide information that can be used to correct or predict the information. An example of this is how the MapCheckerService is used when a NFC beacon reading fails. However, this depends on the nature of the sensor.

Another factor for the adapter's correctness is the state machine used to by the train. It is the states that dictate the trains behavior and if they are implemented poorly this can lead to unwanted behaviors. By defining the scoop of the train states and using the services presented in this paper is the addition and modification of states made more intuitive.

11.1.2 Performance

As seen in Chapter 10 do the adapter in it self perform very well and fulfills the performance requirements required by the system. One of the reasons for this may be the use of Reactive Blocks. Another positive thing to notice is that the OSGi framework and the use of services to not seem to affect the performance in any significant way with the exception of the Event Admin.

As we have seen in the tests, does the Event Admin present a bottleneck for the module and is something that should be investigated further.

As mention before is the adapter performances dependent on the hardware it is running on. These hardware limitations must be taken in to consideration when using the module in other systems.

11.1.3 Using services

Services is used to a large extend in the adapter. For now are most of the services fairly simple, but it shows the potential of what services can be used for. The SensorHandler service can for example be used to change handlers depended on some contextual properties. Also, since the services are accessed through ServiceTrackers can they all, except for the communication service, be updated at runtime without having to restart the adapter. This together with the File Install bundle can for example be used to dynamically download new map information.

11.1.4 Using Reactive Block and OSGi

Throughout the development process has Reactive Blocks together with OSGi shown to be useful tools to reduce development time and open opportunities that else would not be possible. Furthermore, both technologies fit well with ITS system. Reactive Blocks abstracts away most of the synchronization and concurrency management that would else had to be developed. From the tests can we also see that it generates efficient code as it promise. OSGi is a powerful framework that made runtime configuration and modification of code possible.

Without these technologies would it be very hard, if not impossible, to develop a module that has the same possibilities as the adapter presented in this paper.

11.2 Conclusion

The adapter module in it self fulfils the functional requirements and serves as an example of how context reasoning can be solved in an autonomous train system. The test result shows that the adaption module performs well. With this and the use of states it is able to maintain safety properties that ensure safe operations for the trains.

Although the flexibility and configurablity of a system is hard to quantify until a modification to the system is made, do the system design follow the low coupling and high cohesion principle that should help with these properties. Combining this with Reactive Blocks visual representations should help with understanding the systems behavior.

The utilization of OSGi services gives a means to separate system concrete characteristics that allows for code and/or block reuse in similar systems. It also provides the opportunity to develop services that are shared by the different parts of the system, like the communication service described in Section 5.3.1. This together with blocks mostly relaying on interfaces and general propose utility classes should

make for a flexible system where changes to the existing operation environment should lead to limited code modification.

The proposed adaption model architecture makes use of modularization where sub-functionality is implemented in components that can work independent of each other. This should make it easier to allow for having multiple developers working on the module, as long as they agree on the service contract used between the components.

11.3 Further work

11.3.1 Improving the MapChecker service

The current implementation of the service only contains a simplified model of the railroad map. If the service instead could hold a complete model of the railroad map it could be used predict more events and give valuable information to the adaptation module.

11.3.2 Considering the other trains

For now does the adapter only consider itself and not other trains. A very interesting project would be to incorporate information about the other trains running on the same track. The adapter has already means to receive this information with its RemoteControl block. The only thing that needs to be done is to define how the information should be processed by the ContextChecker and optionally services that can help with this. If the improvement of MapChecker mentioned above is done, then the adapter could warn the system about trains in close proximity, reconfigure certain sensors and thereby help with collision avoidance.

11.3.3 Introducing concurrent state machines

If the complexity of the system increases further it can lead to many and complex states. A way to solve this problem could be to use concurrent state machines. With concurrent state machines can the ContextChecker use two or more state machines dedicated to different contextual properties.

11.3.4 Having a separate bundle management service

The only way for the system to updates is bundle during runtime is to manually download the updated bundles to the trains via Git. A feature that should be added to the system is a service that can automate this process. A way this could be done is to send a TrainCommand to the ContextChecker and have it contact the service. Another feature that could be useful is to add methods that allow the adapter to

start and stop bundles. As for our system, there was not much incentive to do this, but in the future it may be needed.

References

[Ada16a] Adafruit. Data sheet for the magnetometer sensor used in the trains. https://www.nxp.com/files/sensors/doc/data_sheet/MAG3110.pdf, Accessed: June 2016.

[Ada16b] Adafruit. Description of the color sensor used in the trains. https://www.adafruit.com/products/1334, Accessed: June 2016.

[Ada16c] Adafruit. Description of the nfc/rfid sensor used in the trains. https://www.adafruit.com/product/364, Accessed: June 2016.

[All16a] OSGi Alliance. The specification for the eventhandler interface. https://osgi.org/javadoc/r4v42/org/osgi/service/event/EventHandler.html, Accessed: June 2016.

[All16b] OSGi Alliance. The specification for the loglistener interface. https://osgi.org/javadoc/r4v42/org/osgi/service/log/LogListener.html, Accessed: June 2016.

[All16c] The OSGi Alliance. About us. https://www.osgi.org/about-us/, Accessed: June 2016.

[All16d] The OSGI Alliance. The documentation for the servicetracker class. https://osgi.org/javadoc/r4v42/org/osgi/util/tracker/ServiceTracker.html, Accessed: June 2016.

[All16e] The OSGI Alliance. The documentation for the servicetrackercustomizer interface. https://osgi.org/javadoc/r4v42/org/osgi/util/tracker/ServiceTrackerCustomizer.html, Accessed: June 2016.

[All16f] The OSGI Alliance. Osgi service platform core specification release 4 version 4.3. https://osgi.org/javadoc/r4v43/core/index.html, Accessed: June 2016.

[AS16a] Bitreactive AS. Activity nodes. http://reference.bitreactive.com/reference/activity-nodes.html, Accessed: June 2016.

[AS16b] Bitreactive AS. Bitreactive joins the osgi alliance. http://www.bitreactive.com/bitreactive-joins-osgi/, Accessed: June 2016.

[AS16c] Bitreactive AS. General information about reactive blocks. http://www.bitreactive.com/technology/, Accessed: June 2016.

[AS16d] Bitreactive AS. Information on have to run reactive blocks together with osgi. http://reference.bitreactive.com/reference/develop-osgi-app.html, Accessed: June 2016.

[AS16e] Bitreactive AS. Installation. http://reference.bitreactive.com/reference/installation.html, Accessed: June 2016.

[AS16f] Bitreactive AS. Java code for building blocks. http://reference.bitreactive.com/reference/java-code-for-blocks.html, Accessed:June 2016.

[AS16g] Bitreactive AS. Overview of the osgi related blocks provided by bitreactive. http://blocks.bitreactive.com/preview/#/library/_mIv8UBIvEeKMseuZHQ_fDA/1.6.2, Accessed: June 2016.

[AS16h] Bitreactive AS. Reactive blocks documentation. http://reference.bitreactive.com/, Accessed: June 2016.

[AS16i] Bitreactive AS. Types of building blocks. http://reference.bitreactive.com/reference/types-of-blocks.html, Accessed: June 2016.

[AT] Jan Olaf Blech Álvaro Férnandez Amir Taherkordi, Peter Herrmann. Architectural virtualization for self-adaptation in mobile cyber-physical systems. Unpublished at the submission of this thesis.

[For16] The Internet Engineering Task Force. The rfc1960 documentation. http://www.ietf.org/rfc/rfc1960.txt, Accessed: June 2016.

[Fou16a] Raspberry Pi Foundation. Raspberry pi 2 model b. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/, Accessed: June 2016.

[Fou16b] The Apache Software Foundation. The specification for the apache felix event admin implementation. http://felix.apache.org/documentation/subprojects/apache-felix-event-admin.html, Accessed: June 2016.

[Fou16c] The Apache Software Foundation. The specification for the apache felix file install implementation. http://felix.apache.org/documentation/subprojects/apache-felix-file-install.html, Accessed: June 2016.

[Fou16d] The Eclipse Foundation. List of included bundles in the equinox framework. http://www.eclipse.org/equinox/bundles/, Accessed: June 2016.

[Gita] GitHub. Git repository containing the modified modified sensor code. https://github.com/henrihs/osgi-train/tree/alexander.

[Gitb] GitHub. Git repository containing the runtime used on the trains. https://github.com/Svae/TrainRuntime/tree/alexander.

[Git16a] Git. Homepage for git. https://git-scm.com/, Accessed: June 2016.

[Git16b] GitHub. Git repository containing the code developed in this thesis. https://github.com/Svae/TrainAdapter, Accessed: June 2016.

[Gro16] Lego Group. Lego mindstorms ev3. http://www.lego.com/en-us/mindstorms/products/31313-mindstorms-ev3, Accessed: June 2016.

[HPMS11] Richard S Hall, Karl Pauls, Stuart McCulloch, and David Savage. Osgi in action. *Creating Modular Applications in Java*, 2011.

[Inc16a] Pivotal Software Inc. Java client api guide. https://www.rabbitmq.com/api-guide.html, Accessed: June 2016.

[Inc16b] Pivotal Software Inc. Rabbitmq java client library. https://www.rabbitmq.com/java-client.html, Accessed: June 2016.

[Jos15] Rohit Joshi. Java design patterns. *Reusable solutions to common problems*, 2015.

[NTN16] NTNU. Homepage of the ntnu's its lab. https://www.ntnu.edu/telematics/its, Accessed: June 2016.

[Par10] The European Parliment. Framework for the deployment of intelligent transport systems in the field of road transport and for interfaces with other modes of transport. *Official Journal of the European Union*, 2010.

[SA16] Matthew Arrott et al. Sanjay Aiyagari, Alexis Richardson. Advanced message queuing protocol protocol specification. https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf, Accessed: June 2016.

[Sve15] Henrik Heggelund Svendsen. Model-based engineering of a distributed, autonomous control system for interacting trains, deployed on a lego mindstorms platform. NTNU, 2015. Project assignment.

[Sve16] Henrik Heggelund Svendsen. Self-localization of lego trains in a modular framework. Master's thesis, NTNU, 2016.

[Wik16a] Wikipedia. Advanced message queuing protocol. https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol, Accessed: June 2016.

[Wik16b] Wikipedia. Behavioral pattern. https://en.wikipedia.org/wiki/Behavioral_pattern, Accessed: June 2016.

[Wik16c] Wikipedia. Constant interface. https://en.wikipedia.org/wiki/Constant_interface, Accessed: June 2016.

[Wik16d] Wikipedia. Osgi. https://en.wikipedia.org/wiki/OSGi, Accessed: June 2016.

[Wik16e] Wikipedia. Software prototyping. https://en.wikipedia.org/wiki/Software_prototyping, Accessed: June 2016.

Appendix A
Java code

This appendix contains code referenced to in the paper.

A.1 TrainAMQPService

```
public interface TrainAMQPService {


        public TrainAMQPChannel openChannel(AMQPProperties properties)
            throws IOException, TimeoutException;
        public TrainAMQPConnection openConnection(AMQPProperties properties)
            throws IOException, TimeoutException;

}

public interface TrainAMQPSendService {

        public void connect(AMQPProperties properties) throws IOException,
            TimeoutException;
        public void send(Object body, String topic) throws IOException;
}

public interface TrainAMQPChannel{
        public void setConsumer(TrainDefaultConsumer consumer) throws
            IOException;
        public void subscribe(String topic) throws IOException;
        public void subscribe(List<String> topics) throws IOException;
        public void unSubscribe(String topic) throws IOException;
        public void send(Object message, String topic) throws IOException;
        public void closeChannel() throws IOException, TimeoutException;
        public Channel getChannel();
}
```

```
public interface TrainAMQPConnection {

        public TrainAMQPChannel getChannel() throws IOException;
        public void closeConnection() throws IOException;

}

public class AMQPMessage {

        private String consumerTag;
        private Envelope envelope;
        private AMQP.BasicProperties properties;
        private byte[] rawBody;

        private String topic;
        private Object body;

        public AMQPMessage(String consumerTag, Envelope envelope,
            AMQP.BasicProperties properties, byte[] rawBody) {
                this.consumerTag = consumerTag;
                this.envelope = envelope;
                this.properties = properties;
                this.rawBody = rawBody;
        }

        public AMQPMessage(String topic, Object body){
                this.topic = topic;
                this.body = body;
        }


        public String getConsumerTag() {
                return consumerTag;
        }

        public Envelope getEnvelope() {
                return envelope;
        }

        public AMQP.BasicProperties getProperties() {
                return properties;
        }

        public byte[] getRawBody() {
```

```
        return rawBody;
}

public String getTopic(){
        return topic;
}

public Object getBody(){
        return body;
}

public class AMQPProperties {

public AMQPProperties() {
}

public AMQPProperties(String hostname){
        this.hostname = hostname;
}

public AMQPProperties(String hostname, int port){
        this(hostname);
        this.port = port;
}

public AMQPProperties(String hostname, int port, String username,
    String password){
        this(hostname,port);
        this.username = username;
        this.password = password;
}

public AMQPProperties(String hostname, int port, String username,
    String password, String exchange){
        this(hostname, port, username, password);
        this.exchangename = exchange;
}


public String getHostname() {
        return hostname;
}

public int getPort() {
        return port;
```

```
        }

        public String getExchangename() {
                return exchangename;
        }

        public String getUsername() {
                return username;
        }

        public String getPassword() {
                return password;
        }

        public boolean equal(AMQPProperties other){
                if(!this.hostname.equals(other.getHostname())) return false;
                if(!this.exchangename.equals(other.getExchangename())) return
                    false;
                if(!this.username.equals(other.getUsername())) return false;
                if(!this.password.equals(other.getPassword())) return false;
                if(this.port != other.getPort()) return false;
                return true;
        }
}

public class TrainDefaultConsumer extends DefaultConsumer {

        Function<AMQPMessage, Void> function;

        public TrainDefaultConsumer(Function<AMQPMessage, Void>function,
            TrainAMQPChannel channel) {
                super(channel.getChannel());
                this.function = function;
        }


        @Override
    public void handleDelivery(String consumerTag, Envelope envelope,
                               AMQP.BasicProperties properties, byte[] body)
                                   throws IOException {
                AMQPMessage msg = new AMQPMessage(consumerTag, envelope,
                    properties, body);
                synchronized (function) {
                        function.apply(msg);
                }
```

```
    }

        public void setFunction(Function<AMQPMessage, Void> function){
                this.function = function;
        }

}
```

Listing A.1: Interfaces and classes used by the TrainAMQPService

A.2 Sensor Event Handlers

```
public class DefaultColorEventHandler implements SensorHandler{

        private EventReceiver receiver;

        public DefaultColorEventHandler(EventReceiver receiver) {
                this.receiver = receiver;
        }

        @Override
        public void handleEvent(Event e) {
                if(e.getProperty(ColorControllerService.STATE) != null){
                        receiver.sendSensorStateEvent(
                            (Status)e.getProperty(ColorControllerService.STATE),
                            PublisherType.SLEEPER);
                        return;
                }
                if(e.getProperty(ColorControllerService.COLOR_KEY) == null ||
                    !(e.getProperty(ColorControllerService.COLOR_KEY)
                    instanceof EColor)) return;
                EColor ec =
                    (EColor)e.getProperty(ColorControllerService.COLOR_KEY);
                if(ec != EColor.GRAY && ec != EColor.UNKNOWN)
                    HandlersActivator.getLogger().log(LogService.LOG_DEBUG,
                    String.format("[%s] %s", this.getClass().getSimpleName(),
                    ec));
                ColorReading cr = new ColorReading(convert(ec));
                receiver.sendColorEvent(cr);
        }

        private SleeperColor convert(EColor c){
                return SleeperColor.valueOf(c.name());
        }
```

```
}
```

Listing A.2: The DefaultColorEventHandler used to receive events from the color sensor

A.3 Sensor Publishers

```
public interface PublisherService {

        public Status getStatus();
        public PublisherType getType();
        public long getPublishRate();
        public long getDefaultPublishRate();
        public void setPublishRate(long rate);
        public void stopPublisher();
        public void read();
        public void write(String content);
}
```

Listing A.3: The PublisherService interface

A.4 ContextChecker

```
public interface EventReceiver {

        public void sendColorEvent(ColorReading color);
        public void sendNFCEvent(NFCReading locationID);
        public void sendAccelerationEvent(AccelerometerReading acc);
        public void sendMagnetometerEvent(MagnetometerReading direction);
        public void sendDummyEvent();
        public void sendTemperaturEvent(TemperatureReading temp);
        public void sendSensorStateEvent(Status status, PublisherType type);
}
```

Listing A.4: The EventReceiver interface

```
public class TrainInfo {

        private double speed = 0;
        private boolean inTurn = false;
        private double heading = Double.MAX_VALUE;
```

```
private TrainStates state;
private SpeedRestrictionLevel speedRestrictionLevel;
private String currentLocationID = "00000000";
private HashMap<PublisherType, Status> sensorStatus = new
    HashMap<>();
private SleeperColor sleeperColor;

public void setSensorStatus(PublisherType type, Status status){
       sensorStatus.put(type, status);
}

public Status getSensorStaus(PublisherType type){
       return sensorStatus.get(type);
}

public double getSpeed() {
       return speed;
}
public void setSpeed(double speed) {
       this.speed = speed;
}
public boolean isInTurn() {
       return inTurn;
}
public void setInTurn(boolean inTurn) {
       this.inTurn = inTurn;
}
public double getHeading() {
       return heading;
}
public void setHeading(double heading) {
       this.heading = heading;
}

public TrainStates getTrainState() {
       return state;
}

public void setTrainState(TrainStates state){
       this.state = state;
}
public SpeedRestrictionLevel getSpeedRestrictionLevel() {
       return speedRestrictionLevel;
}
```

```
        public void setSpeedRestrictionLevel(SpeedRestrictionLevel
            speedRestrictionLevel) {
               this.speedRestrictionLevel = speedRestrictionLevel;
        }
        public String getCurrentLocationID() {
               return currentLocationID;
        }
        public void setCurrentLocationID(String currentLocationID) {
               this.currentLocationID = currentLocationID;
        }

        public SleeperColor getLastSleeperColor() {
               return sleeperColor;
        }

        public void setLastSleeperColor(SleeperColor sleeperColor) {
               this.sleeperColor = sleeperColor;
        }


}
```

Listing A.5: The TrainInfo utility class