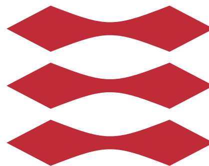
Development and Visualisation of a Distributed Railway Control System

Mads Egedal Kirchhoff



Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of this thesis is to develop a distributed control system for railways, implement and visualize it using Lego Mindstorms hardware and Lego trains. The system should uphold the safety requirements of preventing trains from colliding and derailing.

Railway control systems are traditionally centralised, but using a distributed approach could be more cost-effective. Through the model railway, issues when implementing such a system can be safely exposed and at low cost, and test whether it works and is viable.

An analysis is made of the available Lego hardware components and what kinds of model railways can be made with them, before a specific design is chosen. An algorithm is proposed that can control the trains and ensure safety. It is then refined and formally verified through formal modelling in UMC. The algorithm is then implemented on the Mindstorms hardware in Java, along with a Lego API that allows it to control the Lego railway. The final system is shown to have some erroneous behaviour, but this is most likely stemming from errors in the API or imprecision in the Lego hardware. The control system itself is working. It is concluded that the distributed control system is viable and that visualizing an implementation using a model railway has merit, although the hardware is hard to work with.

Summary (Danish)

Målet for dette speciale er at udvikle et distribueret jernbane kontrol system, implementeret og visuliaseret ved brug af Lego Mindstorms hardware og Lego toge. Systemet bør leve op til sikkerhedskrav der forhindrer togene i at kollidere og blive afsporet.

Sådanne kontrol systemer er tradionelt centraliserede, men et distribueret jernbane kan potentielt være mere billigere. Gennem modeljernbanen kan problemer med at implementere sådan et system i praktisk udforskes sikkert og billigt og det kan testes om det virker og om det er værd at arbejde videre med.

En analyse blev lavet af hvilke tilgængelige Lego hardware komponenter der er tilgængelige og hvilke slags model modeller der kan bygges af dem før et endeligt design bliver valgt. En algoritme fremlægges der skal kunne kontrollere togene og sikre sikkerhed. Den blev forfinet og dens sikkerhed påvist gennem modellering. Algoritmen blev så implementeret på Mindstorms hardware i Java sammen med en Lego API som kommunikere og styrer Lego jernbanen. Det endelige system har visse fejl, men disse skyldes sandsynligvis fejl i APIet og upræcision i Lego hardwaren. Control systemet i sig selv virker. Det bliver konkluderet at et distrubueret kontrol system fint kan virke og implementeringen af en modeljernbane har en vis merit, omend hardwaren er svær at arbejde med.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering. It is rated at 30 points. Anne Haxthausen and Alessandro Fantechi were supervisors.

The thesis deals with designing a model railway, modelling and implementations. Readers should have some basic knowledge of programming and modelling to fully understand the contents.

The report is intended to be read from one end to the other, following a roughly chronological flow.

Lyngby, 01-August-2016

Mads Egedal Kirchhoff

Acknowledgements

Giant thanks to my two supervisors, Anne and Alessandro for putting in all the time and effort into helping with this project. For all the weekly meetings, discussions, effort and energy.

Contents

Summary (English) i

Summary (Danish) iii

Preface v

Acknowledgements vii

1 Introduction 1

2 Railway Domain 3

3 Lego Domain 5
3.1 Mindstorms 5
3.1.1 Components 6
3.1.2 Programming 9
3.2 Power Functions & Train components 9

4 Lego Railway Design 11
4.1 Goals 11
4.2 Design Options 12
4.3 Final Design 13

5 Analysis 15
5.0.1 Requirements 15
5.0.2 Inspiration 16
5.0.3 Constraints 17
5.1 Algorithm Concept 17

6 **Modelling** 19
6.1 Goals 19
6.2 Tools 20
6.3 Design 20
6.4 Final Model 22
6.5 Verification 23

7 **Implementation** 25
7.1 Assumptions 25
7.2 Terminology 25
7.3 Goals 26
7.4 Code Quality 27
7.4.1 Don't Repeat Yourself (DRY) 29
7.4.2 Self-documenting code where possible, comments for general descriptions, intent and the rest 30
7.4.3 Practical Encapsulation 31
7.4.4 Formatting 31
7.4.5 You aren't gonna need it (YAGNI) 31
7.5 Development method 32
7.5.1 Software Tools 32
7.5.2 Work flow 32
7.5.3 Implementation Phases 33
7.6 The Code 34
7.6.1 Structure 34

8 **Testing** 41
8.1 Method 41
8.2 Results 43
8.2.1 Known bugs & issues 43
8.2.2 Documentation 44

9 **Conclusion** 47

A **Project plan** 49

B **Model Code** 53

C **Implementation Code** 67
C.1 ControlUnit 67
C.2 SwitchBox 68
C.3 TrainControlComputer 75
C.4 AutomaticStart 78
C.5 TrackLayout 83
C.6 Communicator 84
C.7 Detector 88

C.8 Switch 91
C.9 TrainMotor 92
C.10 RemoteCalibrator 93
C.11 SetNameEast 94
C.12 SetNameWest 95
C.13 ClearReservation 95
C.14 HailIncomingTrain 96
C.15 ReservationFree 97
C.16 Signal 97
C.17 SwitchBoxSignal 98
C.18 SyncFail 98
C.19 SyncReservation 99
C.20 SyncSuccesful 100
C.21 TrainControlComputerSignal 101
C.22 TrainDetected 101
C.23 TrainNoLongerDetected 102
C.24 TryReserve 103
C.25 UpdatePosition 104
C.26 WaitForReservation 105
C.27 CommTest 105
C.28 DetectorTest 106
C.29 DetectSensor 107
C.30 Ev3onNxt 108
C.31 FullAPITest 109
C.32 HelloWorld 109
C.33 MotorTest 110
C.34 SensorTest 110
C.35 SimpleTrainRun 112
C.36 SwitchTest 112
C.37 SystemTest 113
C.38 TestMultipleConnection 118
C.39 TrainSpeed 119
C.40 UltraSonic 120
C.41 DebugMessage 121

Chapter 1

Introduction

The computer systems used to manage train railways have traditionally been centralised. The whole system is monitored and controlled from a single computing hub. This means all activity and communication has to routed back to this point. This requires a very robust communication infrastructure, possibly over the very great distances a railway can stretch. These kinds of systems can be expensive to set-up, however, especially for small local networks. A possibly more cost-efficient alternative would be a distributed system. Here, each part of the network would have their own logical units that in co-operation with the trains computers could manage the system independently. Such a system is untested however, and the distributed approach has disadvantages. The lack of a global state may create conflicting information and lead to disaster, if not designed right. Such a system must be proven to be safe before it can be implemented in the real world.

Thus, it is the goal of the project to implement a simplified model of a distributed railway system. This is done via Lego Mindstorms, a much cheaper option than building a railway and banging trains together. Creating an actual, physical system, means we can visualize the functionality of the program and actually see it work in practice. Further, we will evaluate whether this approach in itself is valid, whether the model railway has any value as a tool for visualizing and proving correctness for control systems.

This paper is heavily inspired by the paper "Formal development and verification of a distributed railway control system" [1], and can be seen as an extension to it.

It consists of roughly three parts:

- Design of a Lego Railway, setting up the hardware and developing an *Lego API* that will control the railway and trains, acting as a bridge between the generic control system and the physical model.
- Proposing a control algorithm, modelling and formally verifying its correctness.
- Implementing the algorithm on the Lego railway, combining it with the API and testing the whole system.

Chapter 2

Railway Domain

We consider a quite simple railway. Each station has only two tracks with a switching track at either end that allow directing the train to the track they need to go to. Station are connected to other stations via single tracks. Each switching track has a *Switch Box*, or SB, in charge of it. A computational unit that has state for the railway in its immediate vicinity. It decides when the track is switched and when trains may enter and leave the station.

Trains are controlled through by a *Train Control Computer*, which stores information about the trains route and negotiates with the SB for access to specific pieces of track.

Chapter 3

Lego Domain

3.1 Mindstorms

Lego has for many years developed and sold the productline "Mindstorms" which are a set of programmable "bricks" that can be used to construct robots and other such. The product line consists of a central "Intelligent Brick" which is essentially a simple computer, that comes with a simple, visual programming language and the ability to interface with other Mindstorms components. These include rotary servo motors and various sensors, notably including ultrasonic, infrared, light- and color sensors. When connected to the intelligent brick, these feed measured data back to it. The programming can then make decisions based on the data, such as turning a connected motor a specific amount of degrees. Mindstorms exists in three generations, the RCX, NXT and EV3. Each provides a new intelligent brick, with more RAM, processing power and features such as bluetooth. Alongside upgraded or new sensors and motors.

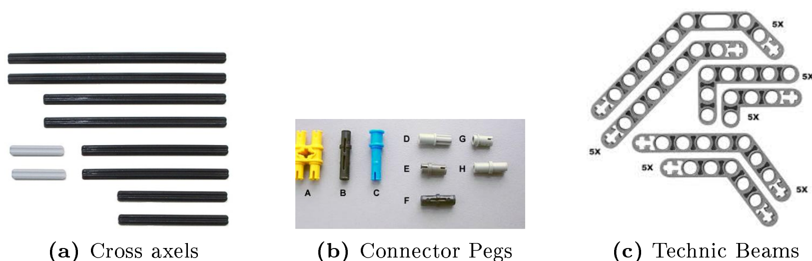
The best central resource to get more info on the specific components seems to be the official lego store's website. At the time of writing, it contains all official EV3 components, and only one NXT. Each page has some specifications written about the component, but information about the components capabilities in details is hard to come by. Such sensors granularity and precision of measurement, for instance. Info and observations about the components considered

in this project follows:

3.1.1 Components

3.1.1.1 Building components

In true Lego tradition, Mindstorms, despite being notable for being programmable, is still very physically orient, with many options for assembly. All of the components are without exception made to fit with various Lego Technic parts, so they can be assembled properly, the parts set together in rigid constructions. Traditional, actual Lego "bricks" are less useful, but can still be fitted together with Technic and Mindstorms.



(a) Cross axels **(b)** Connector Pegs **(c)** Technic Beams

Figure 3.1: Essential lego building components

For context, the connector pegs, technic beams and cross axels are worth singling out. See figure 3.2. These are mostly associated with the Lego technic line, but are part of many other Lego sets and are integral to Mindstorms in particular. All Mindstorms EV3 components have "sockets" or holes for the connector pegs, with some having additional for the axels. The former can rotate in this socket, so the latter is more useful when stability is needed. Gears are another potentially useful component.

3.1.1.2 EV3 Large Servo Motor

The primary motor of the EV3 motor. These motors are ultimately the intelligent brick's outgoing interface with the real world. It performs rotary motion and can measure tacho feedback and a built-in rotation sensor (the latter presumably through the former) and "Tacho feedback to one degree of accuracy. 160-170 RPM. Running torque of 20 N/cm (approximately 30 oz/in). Stall

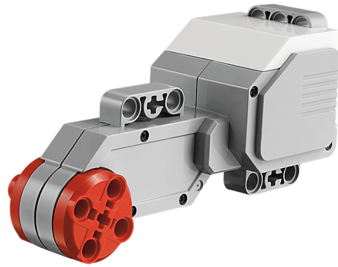


Figure 3.2: Essential lego building components

torque of 40 N/cm (approximately 60 oz/in). Auto-ID is built into the EV3 software".

The final design of the railway system, described in the next chapter, leaves no strict requirement for the strength of the motor. Lego switching tracks have som resistance, but this motor is easily able to switch, and rip apart the assembled construction if it's torque is not managed. The values mentioned are all maximums, and both speed and strength are configurable to this end.

Notably, the motor is only to sense it's rotation in relation to when it started to it started to measure. From the start of the program, basically. Whatever rotation the motor is at at start will be "0". The absolute rotation can not be measured, without assuming it always starts in the same position.

It's physical design is made with a good amount of flexibility, with four holes for pegs on either side of the rotating head, with a socket for an axel going all the way through. Many of these are situated on the body itself, meaning it can be attached with other parts in a variety of configurations.

There exists no official Lego motors that perform linear movements. If such is required, this one can with some construction be set up that performs one using a piston-like movement.

Interestingly, since the motor can both perform and measure rotations, two can be linked, attached to each their intelligent brick and their movements used as a means to communicate between the two. This can be used as a way to communicate between a NXT and EV3 intelligent brick, although it would take considerable effort implementing a communication protocol for this.

3.1.1.3 EV3 Medium Servo Motor

This motor has all the same features, tacho meter, rotating, etc. Difference it being smaller, weaker, but faster. These factors barely matter in this project. It design does to a higher degree. It only rotates a single socket for an axel. This socket faces forward, going into the motor itself, with a very shallow depth. The axel can very easily fall out, at least for the unit I got. It needs to be held fast in it's construction, or face up to avoid this. It's overall different design might make it more convenient for use in some specific situations, though.

3.1.1.4 EV3 Touch Sensor

(Should I mention the "bundle" that we ended up buying, vs buying the educations sets, individual components and so on? Analyse cost-efficiency?)

3.1.1.5 NXT Intelligent Brick

The intelligent brick is the centerpiece of the Mindstorms line. It is often referred to simply as the "NXT". It is basically a simple computer, made to be programmable and interface with the various sensors and motors. It has 3 motor ports and 4 sensor ports. Proprietary ethernet-like cables are used to connect the NXT to the sensors and motors, using the same cables for each. This being the one-generation old, the hardware is hardly impressive, but is good enough for this projects purposes: It can connect to a computer via a USB cable or Bluetooth, the latter of which can also be used to communicate to other intelligent bricks. It does cannot communicate via Wi-Fi. One notable, not readily documented limitation here is that NXT can only have one inbound bluetooth connection and three outbound. If you need more than two NXT communicating, they will therefore have to be set up in a daisy chain. This might be necessary due to the limitations on the amount of ports.

3.1.1.6 Remote

Trains can be controlled via a remote. It has two dials for controlling multiple train and a stop button for each. The dials are quite sensitive and if both are turned at the same time, the signal does not always properly come through.

3.1.2 Programming

Mindstorms has a proprietary programming language of sorts. A software suite can be downloaded from their website, which includes an IDE, capable of being programmed in, compiling and easily transferring and running the program on the intelligent brick. Accessibility is highly prioritised in it's design. The language itself is based on "programming blocks". Visual, draggable blocks of logic, such as if-statements, variable declarations, reading sensors and moving motors. The are assembled together in sequence to form a program, like lines in a conventional programming language. More info at the link. It is a nice little language for learning, a decent approximation of the real thing, but is completely lacking in libraries and functionality beyond the basics. It lacks most of the benefits of objects and functions. Any large project would quickly grow very large, in screen real estate alone, and hard to manage,

A good handful of alternatives exists, developed by the community. There exists APIs for communicating with the Mindstorms components that allow one to program in in various object oriented programs. I ultimately decided to use the Lejos framework using Java as I am quite familiar with both.

3.2 Power Functions & Train components

These run using motors powered by batteries and are controlled using infrared receivers and remotes. This technology is branded under the Lego "Power Functions" line. This line is surprisingly not at all related to Mindstorms: While both have servomotors and have components that can receive and send infrared, they use different protocols. Mindstorms component have no in-built way to communicate with Power Functions, and vice versa. Some third party Lego components capable of this does seem to exist, but are old, mostly out of stock and there could be issues with their support and programming. Like Mindstorms, Power Functions have lots of sockets for connector pegs and such.

CHAPTER 4

Lego Railway Design

4.1 Goals

For this project, the first priority for the Lego system was to create something that was complex enough that it could properly test the control algorithm, while staying within a reasonable budget. Crucially, at least two running trains was needed to allow even the possibility of collision. At least one station was required to allow trains to pass one another and test derailing, and more would make the tests more comprehensive. The budgetary limitations actually provided an interesting challenge; It is easy to dream up an efficient and secure system when one has no limitations on the components one can require for it. In the real world, one might have to make sacrifices in the design for budgetary reasons and so did I.

Secondly, it should be as realistic as possible, a reasonable simulation. Taking inspiration from real systems and components of train networks.

Thirdly, the system should be able to run automatically. This would ease testing, as the system could be run for long periods of time without much supervision, to see if any faulty behaviour occur.

Fourthly, it should be somewhat aesthetically pleasing, convincingly a train

system at a glance. Anyone should be able to verify that the system is working correctly simply by seeing it in action.

4.2 Design Options

These goals in mind, a couple options for the design of the system came up. The inspiration was an assignment in the course "Real-Time Systems". Here, an airport baggage sorter was simulated by motors running belts. Coloured bricks representing baggage trunks were manually loaded onto a running belt, driven by the Mindstorms servo motor. The belt ran the brick past a sensor, identifying the trunk destination based on it's colour. An additional, perpendicular belt would then change direction if needed, before the new trunk were loaded onto it (reference to assignment*). I considered a similar approach here, where the coloured bricks would represent trains and would be moved onto different stations. A system like this would be easy to control by the intelligent brick; It can control the "trains" very directly through it's motors, so there's a low chance of unpredictable movements and such. However, such a system would be hard to expand, without requiring a lot of components: Each track would need one belt and servo, as well as one for switching between stations at each end, and one set for each track at each station. A system with two stations would need seven servos to set up (illustration*). More if it is to be looping, which would be hard to set up geometrically. Sensors might need to be either have to be set up at each station track entrance, and Further, it would have many unrealistic traits, such as trains always moving at the same speed and direction on the same track, and having very sudden acceleration and deceleration. Lastly, the coloured bricks would not look much like trains and not be a very convincing simulation.

Something more closely simulating trains were necessary, so it was decided to actually use train tracks, and build something resembling trains. One possible design here would be having the intelligent brick as the core of the train and using the servos to drive it, as shown here: `https://www.youtube.com/watch?v=H6HYj3afSfg` This would closely simulate how the real system would work, a separate computing unit for every train, as well as for every switch, leading into the stations and enabling the trains to have extra functionality, such as using ultrasound sensors to prevent collisions. This would be prohibitively expensive though, each intelligent brick being quite expensive. Further, these trains would have to be long, heavy (thus slow moving), and have very little braking distance. It was also unclear whether this kind of train would be able to turn at switching points, as the standard Lego train axels might not be rotatable by the motors.

In the end, I settled on using the actual Lego trains. There was some challenge in getting the Power Functions parts in the trains to interface with the Mindstorm components. The intelligent brick cannot not send a signal directly to the train's IR-receiver or it's remote to start or stop it. Instead, a design was chosen were the Mindstorms motors would turn the dials of the train remote.

4.3 Final Design

The final system thus consist of:

A circular train track, with four switching tracks. Each pair of switching switch face each other with some track in between, forming a station. Two trains, each consisting of an IR-receiver, a battery and a train motor, plus normal Lego bricks for structure. An IR-remote, that can control two trains, via two dials. Eight sensors, located some distance before each switching track. Six Mindstorms servo motors, two turning the dials of the train remote, the rest each operating a switch. Four light sensors, before each switch. Two NXT Intelligent Bricks. Mindstorms wires and lots of various, normal and technic Lego bricks, axels, pegs, etc, for connecting and holding components in place.

This is the base system and can be expanded with more stations and such, but this provides a decent complexity.

A switch box is simulated by this combination of the sensor, a motor turning the switch and a thread on the intelligent brick. The train control computer is similarly hosted as a thread, and controls the train through the motor and the remote.

The sensors basically functions as a way to keep track of the position of the train. There seems to be no way using Lego components to accurately keep track of the train at all times, but it is possible to see when it is approaching a critical section. Specifically a switch in this case. At this point, we can then communicate the position to both the train itself and the switch box. What train to contact can be determined by it's color, as it's light value. This can also function as a way to simulate limited range of whatever means the SB and TCC use to communicate. Wi-Fi, for instance. Ideally, we would have a sensor on either side of each switch, on every track. So that could be accurately tracked when the train had left every part of the track. However, this would greatly inflate the number of sensors and intelligent bricks needed and make setting the whole system up harder. I choose to work within this limitation, seeing if the system could still be made reasonably safe with a budget.

Chapter 5

Analysis

With the hard limitations of the physical set-up, the actual trains, railway tracks and programming units settled, I could start to design the distributed control system.

5.0.1 Requirements

Generally, we want to ensure that the system is safe. Therefore, we require that there are no collisions, defined as two trains never physically being, or being allowed, on the same track segment. We require no derailing, defined as a switching track being switched while a train is still passing it.

In addition to these safety requirements, we also want the system to run interrupted and not stall out. Therefore, a requirement for no reservation deadlocks. That is, two trains attempting to access the track the other is occupying and thus halting indefinitely. This will be solved simply by giving the trains schedules that cannot deadlock, by each station track only ever being visited by a specific one of the trains.

5.0.2 Inspiration

The algorithm of the control system is based on the one in the report "Formal Development and Verification of a Distributed Railway Control System"[1]. This algorithm is based around reservations and locks. Each switch box have state components for:

- Which tracks the switching track connects.
- What train has permission to pass it.
- Whether it's sensors are detecting a train.
- Whether each of it's associated/neighbouring tracks are reserved and by which train.
- The direction the associated tracks can be traversed in.
- A transaction flag, that when enabled, disallows the switch box from receiving other commands while it is performing commands on multiple, other switch boxes.

Each train control computer have state components for:

- Its route, as list of track segments.
- The switch boxes along this route.
- The travelling direction of the train itself.
- The position of the train.
- What reservations the train currently has.
- What switch box it has permission to pass, if any.

In order to enter a track segment, the train has to have a reservation for it. The TCC must ask the relevant switch boxes for a reservation and be granted or denied access.

5.0.3 Constraints

The most notable difference in the railway system underlying the report and this one is the way we detect the trains. Whereas the paper assumes there are sensors capable of detecting when a train exactly have is near and leaves a switching track, I only have the capability to do the latter. The algorithm thus have to be modified to take this into consideration.

Further, differences in modelling and implementations can make it necessary or practical to change certain details in the algorithm or state components. More on that in their respective chapters.

5.1 Algorithm Concept

A train approaching a switching point from outside a switching point starts exchanging data with the switch controller once it is in range of the sensor. The sensor identifies the train and determines where it is coming from. The switch controller sends a request to the train, indicating it is approaching a station. The train tells the switch which segments it wants to enter, according to its route. The switch controller checks its reservations: If either the segment or the switch is already reserved, the train is told to stop and wait for this to change.

When or if both of these reservations are free, the switch box needs to synchronize it's state with the other switch box connected to the track. This represents one of the major complications of having a distributed system: We have no global state of reservations, so we have to ensure that switch boxes internal states are compatible and up to date. There can be a conflict, by both switch boxes simultaneously trying to reserve some track. Here, one will get priority according to an arbitrary measure, such as which reservation request came first. Both switch boxes will reserve the track according to the one with priority and the other try again once this process is finished. The reserving switch box will also reserve, or lock, the switching track itself, and switch track the track if necessary. The train is then notify it has the reservation and start if it was stopped. Switching is fast enough that it is unnecessary to stop the train to wait for it. As a safety feature, the train should still stop automatically if it does not receive a reservation acknowledgement from the switch box in due time.

Once the sensor loses sight of the train, the switch controller waits a short period for the train to fully traverse the switching track, the period length determined experimentally and based on the trains speed. Then it notify the

train it has passed the track, removes reservation for the segment the train came from, messaging the neighbouring switch box to do the same. Notably, no conflicts are possible here, as Relying on the train to travel a certain distance in a certain time seems unsafe, but is necessary due to our limitations in the amount of sensors.

The train updates its internal position once it has passed the track, then stops at the station. Optionally, the trains schedule could include a minimum stopping time at each station. When this time has passed, it will message the other switching controller, asking to reserve the next segment in its schedule. The process then continues as before, except the switch box not being currently able to detect the train. Notably, the switch will not have to be switched in this case, as the switching track is flexible enough to allow the train to pass in either position.

Chapter 6

Modelling

6.1 Goals

The overall goal of the model was to use to formally verify the correctness of the control algorithm. To ensure that the safety requirements are held. In order to do this the model should accurately correspond to the real-life system and accurately reflect the same limitations. For instance, the model algorithm cannot know the precise location of the train. It can only work with whatever input it gets from the sensors. In contrast, we can track such information for verification purposes, but no functional part of the system can make use of it.

The model should also closely correspond to a possible implementation. It should strongly suggest how the implementation should be made, so that the correctness of the model applies to it. Ideally, it could work as a blueprint or pseudo code. This can be hard to structure ahead of time as modelling languages ultimately tend to differ in their functionality compared to a given programming language. For example, the reservations in the switchboxes could be represented with a HashMap in Java. A key like a string or integer could be used to access each. This was not possible in the modelling language, so the specific reservation would have to be found through a loop or an array with a index. This ultimately goes both ways, as working on the implementation also gave insight into how the model could be updated to be more correct and

efficient.

Lastly, the model should be as simple as possible and have as few configurations as possible.

6.2 Tools

There exist a number of tools for modelling. One option would be UPPAAL. UPPAAL is made for real-time systems and have capabilities for modelling the passage of time. The railway UPPAAL has both the advantage and disadvantage of having a very visual, WYSIWYG-like interface. State machines are constructed by dragging and dropping around nodes and transitions and then adding guards, variables and other such. This effectively makes development more intuitive and less systematic.

UMC was chosen for the challenge of learning a new modelling language. It can be found here `http://fmtlab.isti.cnr.it/umc/V4.2/umc.html`.

6.3 Design

Here are listed some of the more notable design choices taken during the development. The rest of the work was mostly a matter of expressing the algorithm in UMC syntax.

6.3.0.1 Abstraction

One quite important decision was deciding on scope and abstraction level of the model. Whether the model should cover the algorithm, lego API, utilitiies and all the minutae of each. Or be very high-level and only model the communication, for instance. I ultimately decided that the focus of the model should above all be on the algorithm, proving it's correctness. It should be modelled in detail, in order to be directly translatable to the implementation.

The lego API part, the part of the code that interact directly with the sensors and motors were deemed to be less important to model in detail. Partly because it is hard, if not impossible to do so in an accurate manner in UMC, given that they rely on real world input. What input the sensors read at any time has to

given, so it cannot be proven by the model that they read or get the right data. This is ultimately one of the strengths of implementing the algorithm on a real world system: Seeing what kind of unexpected problems with input and output arises. With the real world component and Lejos API unable to be modelled, the remaining API would also be trivially simple; We can model what values we expect a sensor to read, but nothing about how these values were read and modified. Further, the Lego API is very platform dependent, the least generally applicable. We are most interested in proving the algorithm works, as it can be applied to other hardware than the Lego. As such, the Lego API is modelled very simply and naively and are effectively merged into the TCC and SB to reduce communication between class and simplify the model.

6.3.0.2 Reservation synchronization

Perhaps the largest difficulty of designing a distributed system compared to a centralized one, is the lack of a global state. We do not have one computational unit who knows everything about what is going on and is the final authority on data and decisions. One could perhaps have units constantly exchanging data, updating each other with every variable change and keeping their states perfectly aligned. However, this would crowd the network, and the model and would not be leveraging the strengths and unique properties of a distributed system. It is a strength that every unit does not have to know everything and can be simpler. Further, we cannot guarantee the trains would be able to communicate at all times with the other components, given the distances they would travelling in the real world.

The most notable challenge with a lack of global state is synchronizing reservations in the switchbox. Every track is associated with two switchboxes, which each have their own internal state of reservations. If two trains simultaneously manages to reserve a track with each switchbox they would be allowed to enter and would collide. We therefore have to make sure switchboxes synchronize their reservations and agree on them whenever a reservation is being made. They do not need to know the global state, all reservations in the entire system, only for the tracks they are associated with.

I knew going in this would be a challenge and deliberately set up the starting scenario to test this, both trains starting out wanting to enter the same track, reserving via different switchboxes. Solving this greatly impacted the model and major parts of the switchbox' code and states exists in order to deal with this. *

6.4 Final Model

The final model can be found in Appendix C.

6.4.0.1 Classes

The model consists of two classes, "SwitchBox" and "Train". SwitchBox models the computational unit itself and it's logic, including it's interactions with it's sensor and the physical switch itself. These latter are modelled only in the very abstract; I simply assume the SwitchBox can accurately detect when a train is in front of the sensor and acts upon this. A single signal represents the SwitchBox moving the switch and it is assumed to happen instantaneously.

The Train class represents both the physical train, primarily as it can be detected by the sensor, and the TrainControlComputer and it's programming. These two parts have each their own statemachine. Logically, these might be more appropriate as different classes, but this would require more signaling and references. The train would need a reference to each sensor/switchbox and vice versa, which would likely to a larger state space.

6.4.0.2 Events

Switchbox:

```
Class SwitchBox is
Signals
   tryReserve(track: int, fromStation : bool, train: Train,
        currentPosition : int),
   startReserver(reserverNo : int),
   syncReservation(trackToReserve : int, train : Train, otherSwitchBox
        : SwitchBox, priority : int),
   syncSuccesful,
   syncFail,
   trainBySensor(detectedTrain: Train),
   detectedTrain(train : Train),
   trainNoLongerDetected,
   clearReservation(track : int),
(...)
```

The tryReserve signal is sent from a Train, indicating that it wants to make

a reservation on a given piece of track. The parameters are respectively: the number of the track it wants to reserve, whether the train is in a station currently, a reference to the train itself so it can be signalled back and what track the train is currently on.

Notably, signals are always sent from the same kind class (e.g. tryReserve is always sent from a Train to a SwitchBox) and a lot are only sent from a single point in the program. Very few operations were used. In the cases where the sender has to receive an answer, such as a confirmation, this is usually done as two signals instead, so the sender could not have to answer in the immediate transition. Further, operations had a tendency to enlarge the state space. The synchronous behaviour was often not desirable, locking up the SwitchBox's state machine made it unable to receive new inputs and faulty behaviour. Lastly, at the implementation end, communication between remote components as two NXTs, and probably SBs and TCCs in a real-life system, is ultimately done by writing strings, ints and the like on streams. There is no in-built functionality in lejos to send such messages expecting a return message and implementing one would be messy. More on this in the implementation chapter. As such, signals also made the model easier to translate.

Chapter 7

Implementation

This chapter describes the development goals, process and resulting java code for the final lego railway.

7.1 Assumptions

7.2 Terminology

Definitions for basic terminology from java and object-oriented- and general programming that I use in this chapter.

- Class: A fundamental concept of object-oriented programming. A class is a logically distinct part of the program, often it's own file. It consists of data (variables) and procedures for acting on data(methods). One can create multiple *instances* of a class, different *objects* that have the same functionality. Objects are analogous to real-world objects, such as a "class Car" which have procedures such as "start()", "brake()" or data like "color" and "currentSpeed".

- Variable: A generic container of data, such as numbers, characters or text.

- Field: A variable that is available globally at least within the instance of the class, or for every instance if made *static*. Can be made available outside the class by declaring it *public*, as opposed to *private*.

- Refactor: Change codes structure, naming, appearance and other such, without altering it's functionality or external behaviour. This is most often done to improve the quality of the code in areas such as readability and maintainability.

- IDE: Integrated development environment. A software application designed to make programming and developing software easier and more efficient, with for instance: advanced text editing, automatic and refactoring tools.

- Method: A collection of lines of code, that form a specific procedure, behaviour or functionality. Can be given data in the form of parameters, variables, as input and return some output. Data in a program is essentially transformed via methods. Equivalent to the *function* in imperative programming, but part of an object.

- Exception: Java creates and "throws" exceptions when something unexpected happens that cannot properly be handled, such as calling a method on object that does not exist.

- Call stack: The collection of methods currently that have not been completed in the program.

- Stack Trace: The contents of the call stack often shown in modern IDEs such as eclipse when an exception is thrown. This allows the programmer to more easily track down where the exception occurred and fix any bugs causing it.

- Package: A feature of Java. A collection of related files, used to organize the program or limit accessibility across classes. Analogous to folders in windows.

7.3 Goals

I had the following goals for the implementation of the railway control system on the Lego NXT, in roughly this order of falling priority:

1. Develop a correct program, that is lively and safe, able to have the trains constantly running automatically without derailings or collisions.

2. Base the implementation on model and have the final implementation closely resemble it. The results of the verification done on the model should be provably applicable for the program as well. The correlation between the code and the design should be fairly easy prove and understand as well.

3. The code should make as few assumptions about the structure and lay-out of the railway as possible. Ideally, the code should be able to be ported to a completely different railway systems, with a different amount of trains and stations, by only changing the initializations. Not having tested the system as such, other such systems

4. The code should be flexible and extensible, allowing for the adding or changing of features easily. This way, the most fundamental, minimal solution can be implemented first and less critical features can be added as time allows. This also makes the code more useful for anyone else seeking to use and extend it.

5. The code should be well-structured, readable and in general follow best practices, of object oriented programming in particular. This is a goal for the above stated reasons, as well as simply being an exercise in writing good code in general. More on this in the next section.

7.4 Code Quality

It requires little in terms of skill or talent to write code for a simple program that does what it is nominally supposed to do. Writing "good" code meanwhile, code that is elegant, readable and not a nightmare to maintain, is an art. Some parameters of good code is readability, succinctness, maintainability, high performance, and correctness. The latter two affects the end user the most, and may therefore be the most important; If the program does not what it is supposed to do, is buggy or slow as molasses, it is not of much use. With modern hardware, however, many applications will run fast enough regardless of optimizations, so performance is not always a priority. The rest can be critical to the long-term success or viability of a piece of software. Bad code require more resources, in time, money or developers to maintain and update.

In the short term, though, low-quality code can be faster to develop. Simply taking less time to write because of less thought put into it and fewer revisions.

While this project has a quite limited scope and is unlikely to have much maintenance done to it after being nominally completed, good code standards can still help during development. As stated by Andy Hunt, author of the book "Practical Programmer":

> *"All programming is maintenance programming, because you are rarely writing original code. If you look at the actual time you spend programming, you write a bit here and then you go back and make a change. Or you go back and fix a bug. Or you rip it out altogether and replace it with something else. But you are very quickly maintaining code even if it's a brand new project with a fresh source file. You spend most of your time in maintenance mode. So you may as well just bite the bullet and say, 'I'm maintaining from day one.' The disciplines that apply to maintenance should apply globally."*[3]

For example, even as the sole developer, returning months later to update an ambiguously written piece of code can lead to confusion and bugs. This actually happened several times during the project.

As will be discussed further in section 7.5, debugging in this project is quite difficult and time consuming. Therefore, using programming patterns that reduce the likelihood of bugs being introduced during development was a high priority.

There exists many programming patterns, best practices and principles and that acts as guidelines to how to write good code. Ultimately, there is no single, definite authority that sets these and the standards for what quality code is. Best practices can often be mutually contradictory or come with significant costs. Internet arguments abound around what is and is not a good way to solve a particular problem. Below are what best practices, principles and such I have tried to follow for this project, based on my own experiences and reading too many stackoverflow debates.

7.4.1 Don't Repeat Yourself (DRY)

"Don't Repeat Yourself" is a fairly well-known principle, formulated in, but not originating from, the book "Practical Programmer" [2]. It states essentially that the programmer should as much as possible avoid duplicating lines of code, pieces of logic, behaviour and other such. Instead, all parts of the code that needs a specific behaviour, should refer to the same, single, authoritative source of that behaviour. For instance, a block of code repeated throughout the project was turned into a method and the method called instead. This way, only that one method has to be updated when the behaviour needs to change and common errors where not all duplicates are properly updated are avoided. Further, it may simply cut down on the amount of time spent typing and clicking when updating the lgoic. Some possible costs to DRYing code can be:

- Increased nesting of the code that makes it harder to navigate (methods calling methods calling methods).

- Later updates necessitating "unDRYing" the code, inlining methods and such, as the behaviour has become too disparate.

- The encapsulated logic may be very arbitrary and hard to describe, in comments and naming. This makes the code less readable.

As such, one should still be careful in applying this principle. It is not universally applicable. It is less relevant especially in the "outer layers" of the program, classes with no other have dependencies to, such as testing and initializers. For instance, it is a matter of judgement to determine how many lines of repeated code warrants a new method. For my part, I believe that even a single line of code, that is likely to often change and repeated once, can be worth turning into a method.

7.4.2 Self-documenting code where possible, comments for general descriptions, intent and the rest

Ideally, code should not not need to be commented to be understood by any decent programmer. The code itself should clearly indicate this intent. This is advantageous because comments bloat the code and must be constantly revised along to the code. Code ultimately cannot lie about what it does, comments can and will when outdated.
Code can be made more self-documenting by using appropriate naming for methods, classes and variables that clearly describe what they do. "int todays-Date" rather than "int aa". Also following the languages conventions for names using *camelCasing*. Also, dividing the program into logical chunks through classes and methods, so that the relation between each is apparent and intuitive. In practice, I do not believe this is wholly enough. There is a limit to how much a name of a reasonable length can explain and how much one can divide the code. As a rule of thumb, code can and should itself describe "what" it does, while comments are good for describing the "why". Why this specific algorithm or approach was taken as opposed to the alternatives, why the specific values was used. Especially convoluted pieces of logic, odd necessary but hacks and such are also worth explaining in comments. General descriptions of classes, methods and fields also helps understanding them at a glance without having to dig into the code itself.

7.4.3 Practical Encapsulation

Encapsulation is one of the fundamental tenants of object-oriented programming[?]. Essentially, classes should be fairly independent and secretive, only providing access to the fields and methods that truly need them. A standard approach to this is to make all variables private and provide *get* and *set* methods for them, that provide read and write access respectively. I disagree with this approach however, at least for a project of this nature and size. The get and set methods create a huge amount of code bloat that make navigating the actual, functional code harder. Even in larger projects I have seen very few instances where they gave any advantage over public fields. I would argue refactoring is often preferable to modifying the output or input via getters or setters. Instead, I generally declare fields and methods private if they only used and relevant inside the class, public or protected otherwise.

7.4.4 Formatting

Much like any piece of text, reading code is easier if it is formatted well. Very dense code can be tiring and hard to read. Improving formatting was mostly done by using the auto-formatter of Eclipse constantly and using plenty of whitespace to give spacing and divide code into the chunks that are the most directly related. As a rule of thumb, at most ten lines of code should be clumped together without linebreaks or other spacing. Very long methods and classes may also be broken up into multiple for the sake of formatting. Lastly, a single code line should also be kept within a reasonable width. The auto-formatter will automatically break lines above 100 characters per default, but it is oft preferable

7.4.5 You aren't gonna need it (YAGNI)

While it is good to develop the program to be expendable, it is easy to be carried away by this approach. If one tries to anticipate all the programs future need, there's no end to the redundant code one needs to implement. Further, these implementation might well be outdated and need in heavy revising when the time comes to actually use them. The YAGNI principle of *Extreme Programming*[?] is thus to avoid the common trap of creating functionality in anticipation of needing it.

7.5 Development method

This section describes the software tools I used during the development process and my general approach and flow of working.

7.5.1 Software Tools

The code was written solely in the 32-bit version of Eclipse Mars 2. A couple of plug-ins were used: ObjectAid UML Explorer were used to generate UML diagrams for the final system.
C/C++ development tools were installed since they add the functionality to create *Launch Groups*, which makes uploading programs to both NXT bricks easier.

The Lejos NXT plug-in for eclipse plug-in adds a whole host of features that makes developing for the NXT through eclipse easier. Most importantly, it adds library support, so Lejos methods are recognized and auto-completed.
Secondly, it can be configured to print out the ids of the methods in the program, which is very useful for exception handling and debugging.
Thirdly, it simplifies the process of uploading the program to the NXT, which can be done with a single click. It only works for the 32-bit Eclipse. More information about how to use the plug-in can be found at this link:

(`http://www.lejos.org/nxt/nxj/tutorial/Preliminaries/UsingEclipse.htm`)

Google Drive's desktop app was used as a simple kind of version control. It works as a back-up, syncs the code across computers and has simple, but decent tools for rolling back to previous versions of the code or comparing differences. I deemed proper version control systems such as *Git* or *SVN* unnecessary due to the limited scope of the project. There were no problems along the way that necessitated them.

7.5.2 Work flow

In this section I describe a little of the process in terms of the time was primarily spent on and how I worked.

7.5.3 Implementation Phases

The implementation happened approximately in four phases, each requiring a significant amount of work and involving different challenges.

- Setting up the project, researching and testing the Lejos API and developing a first version of the Lego API. This was done before modelling.
- Creating a system for how the two NXT to communicate and in turn how the the SBs and TCC signal each other. Done after modelling.
- Implementing the algorithm, the logic of the SBs and TCCs. Essentially translating the UMC model to Java and revising it and refactoring it according to the principles stated in section 7.4. Notably included finding a way to emulate the statefullness of UMC and handle signal queues. Debug and test and using a simulated system, faking output from the physical Lego components.
- Updating and tuning the Lego API to work together with the other parts and improve stability and reliability, including rebuilding parts of the physical railway system. Debug and test on the actual system. A large part of this phase was spent simply experimenting with different threshold values for sensors, degrees motors should turn and such.

Notably, I did not draw any UML diagrams or other such design documentation beforehand. Partly because the model already provided a very solid foundation and I did not have enough experience with the Lejos API or programs with communicating threads . I did have some vague ideas for the possible structure for the program in mind before starting, but these changed and evolved during the implementation process.

7.5.3.1 Debugging on the NXT

Debugging during this project was a special challenge. Most modern IDEs such as eclipse have advanced debugging capabilities that allow you to step through the program, line by line, while seeing the changing internal state of the program. This makes it easier finding errors in this state and their originating point easier to find. This feature does not exist for when running the program on the NXT, unfortunately. The main debugging tools here is displaying text on the screen of the actual NXT and stack traces automatically shown when an exception is thrown. It prints a simplified stack

trace, with the ids of the current methods on the call stack. Further, the screen of the NXT only has room for sixteen characters per line and eight lines, so it is limited how many of these prints can be on screen at once.
For these reasons, functionality was implemented to standardize the way both of these are printed and allowing for easy enabling and disabling of printing of specific debug messages.
As such debugging involved uploading the program to the NXT, observing erroneous behaviour, altering the program to print the values of a few variables that could be causing the errors, uploading the program again and then repeating this process until the error was found. Towards the end of the project, uploading the program took some ten seconds or longer.
I briefly considered developing a mocked PC Lejos library. This library would have all the methods I use from the Lejos library and simulate the trains moving virtually. However, this would be quite the endeavour to develop, this library likely needing plenty of bug-fixing in itself. It also would not help with tuning errors in the Lego API itself.

7.6 The Code

In this section I will go through the code and describe classes and methods of the program in general and how the code works, as well as detail some points of interest.

7.6.1 Structure

Here I describe the overall structure of the program, how the classes and files are organized and interact with each other in the broad strokes.

7.6.1.1 Packages

The project consists of seven packages. See figure 7.1 for an overview of their contents. A general description of the packages follows:

- The *controlSystem* package contains the heart of the railway control system, the algorithm for running the trains and securing the safety properties. It contains the logic for the switch boxes and train control computer within the classes with the same names. These are based on the classes

in the model. This part of the code should ideally be reusable outside of the lego domain, adaptable to actual trains and switch boxes.

- *initialization* contains classes that start the full lego railway system using the Main method in Java, initializing all the objects. These classes are where the data around how the railway system is set-up is primarily kept. What sensor is connected to which port, how many switches and SBs there are, train routes and starting position, etc..
- *legoAPI* contains all the functional part of the code that controls and reads data from the mindstorm components, interpreted into the railway domain. It essentially works as a intermediary between the control system and the Mindstorms components. In a realistic system, these files would be swapped for equivalent files that control real sensors, train motors, etc.
- *nxtUtil* contains various mostly stand-alone programs that are useful for setting up the NXT themselves. They are not actually used in the actual run of the control system, but they change or test properties on the NXT that may be required for the system to work.
- *signals* contain all the signals that are used for communicating between TCCs and SBs. They can be sent between the two NXTs as well
- *tests* are all of the test files used during development of the system. Test of Mindstorms components, of individual pieces of the Lego API and a simulated version of the whole system. None of them are used in, or tests, the final, physical system. Some of the test might be useful for setting up the system and calibrating, however. The number of files here should indicate something about the difficulty and importance of testing this system.
- *util* contains utility classes, generally made up of static methods that are useful for many disparate parts of the program.

7.6.1.2 Class Diagrams

Class diagrams for various sub-sections of the programs follows. Figure 7.2 depicts the relations between the control systems files and initializations. As the one that initializes these objects, "AutomaticStart" has dependencies to most other functional classes, while being devoid of any logic for outside use. "Reserver" is a nested class of Switchbox and it, SwitchBox and TrainControl-Computer extend ControlUnit. It standardizes how their signal queues and ids for communications work, and provides some utility functions related to these as well.

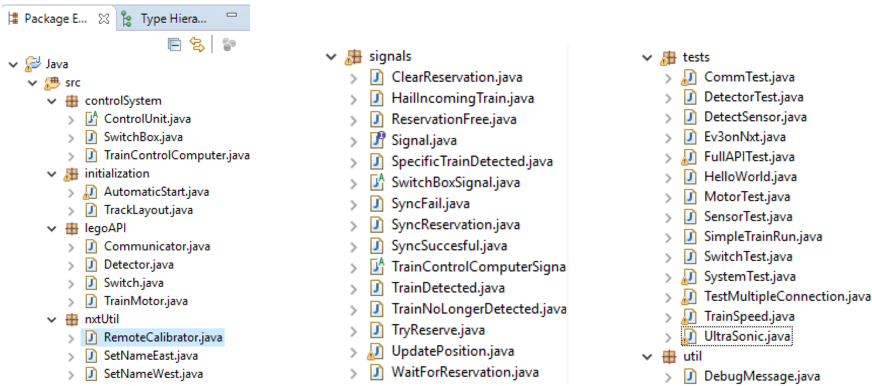


Figure 7.1: Project packages and files as shown in Eclipse

The lego API.

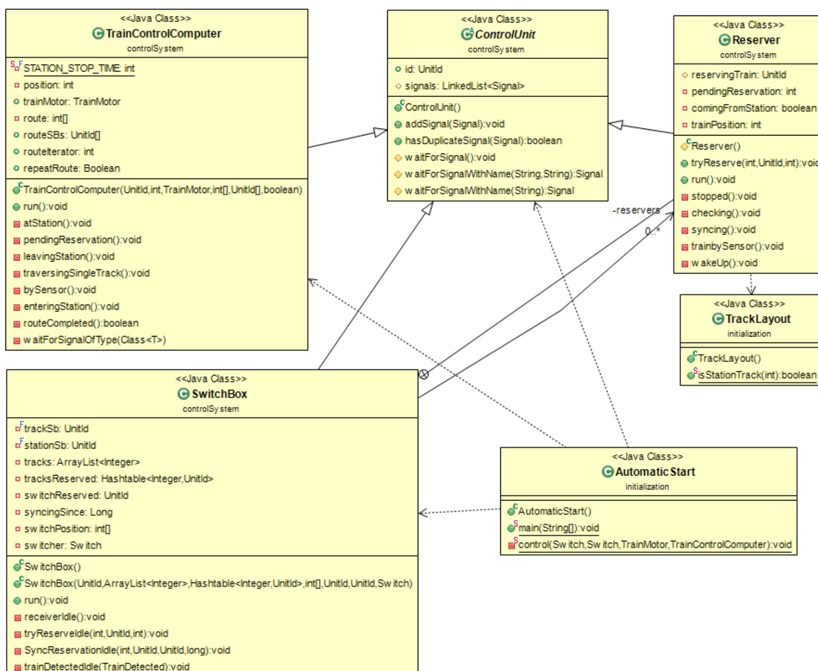


Figure 7.2: UML class diagram of the classes from controlSystem and initialization

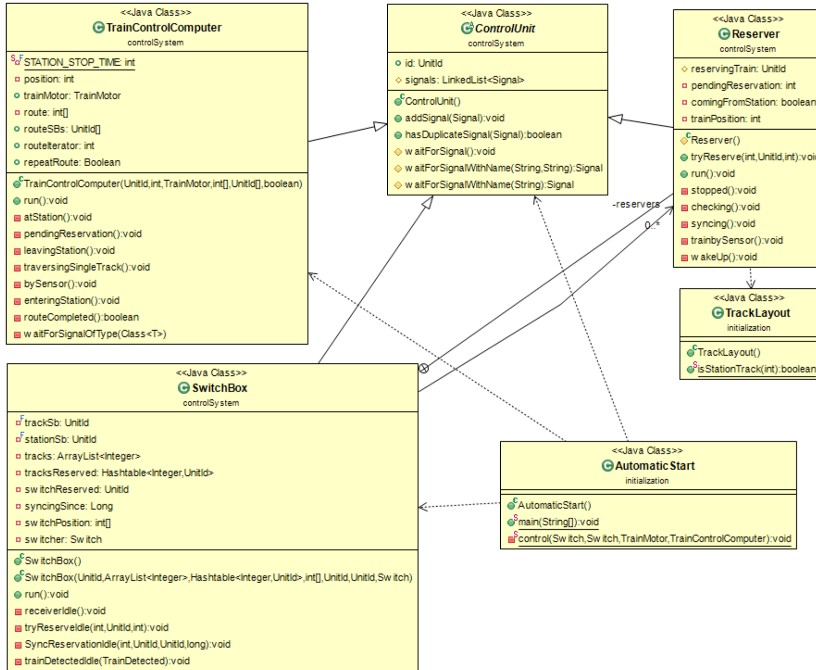


Figure 7.3: UML class diagram of

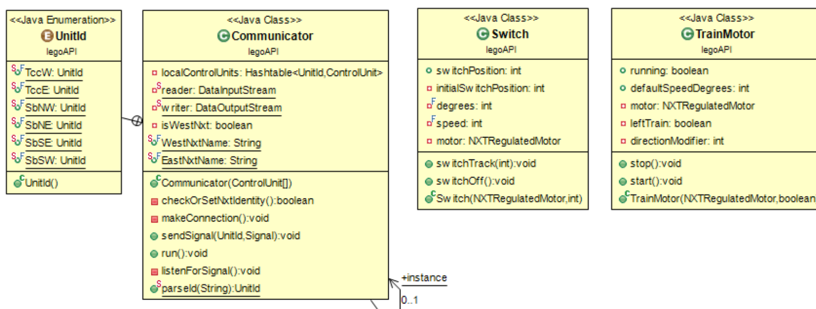


Figure 7.4: UML class diagram of the Lego API classes

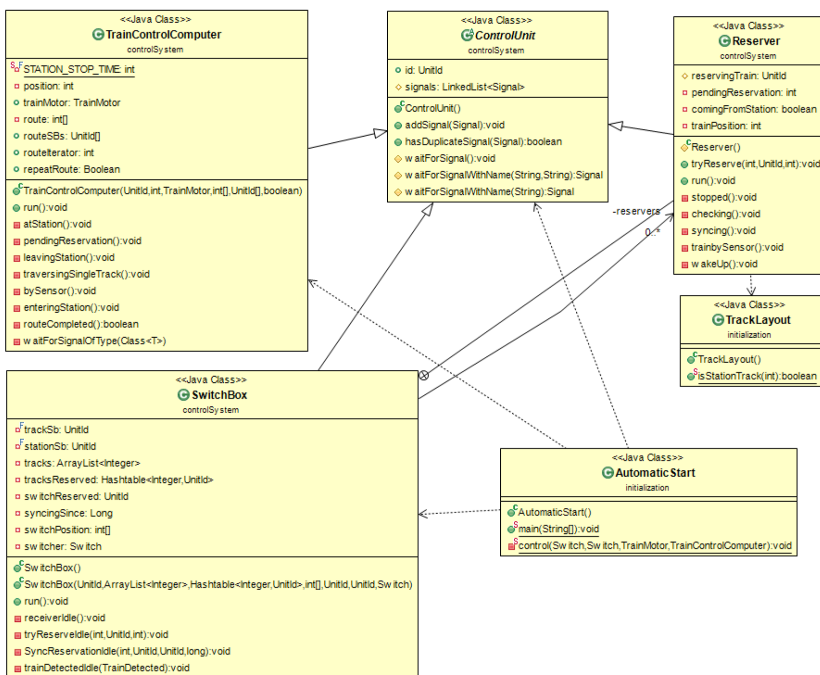


Figure 7.5: UML class diagram of

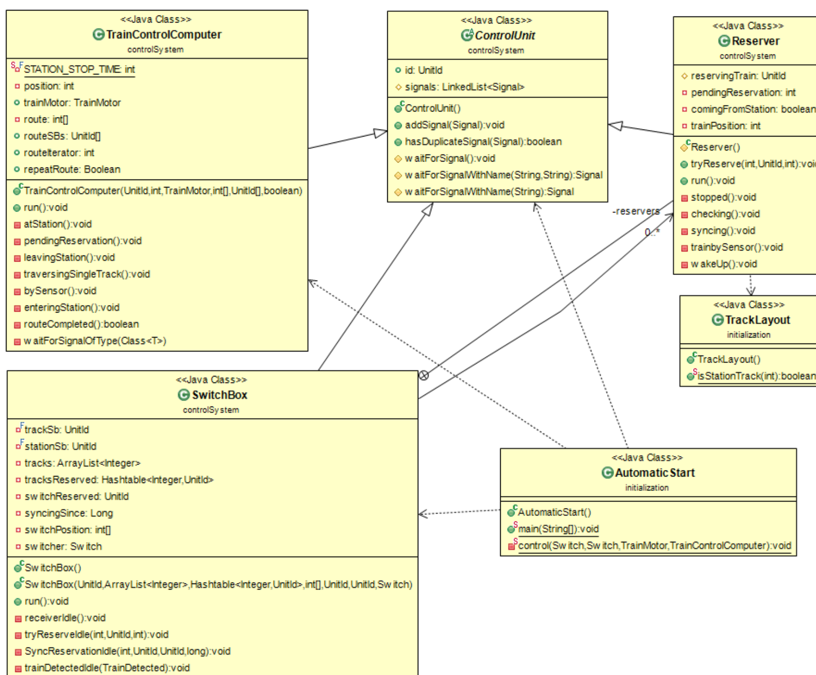


Figure 7.6: UML class diagram of

Chapter 8

Testing

This chapter will focus on my approach to testing the program, ensuring its correctness.

8.1 Method

When dealing with a safety-critical system such as the railway control system it would make sense to make use of an automatic testing framework, such as Eclipse's JUnit *unit testing* tools. This is not possible when developing for the NXT, however. At best, one could develop their own framework for the platform with capabilities for *asserts*]and the like, but it could never be fully automated, as the program still has to be manually uploaded, the NXT turned on and the trains in the right positions.

Therefore, I opted for the simple approach of manual testing. Creating classes for creating better error messages and testing specific parts of the project, described in chapter 7. Due to the limits of the debugging tools as also discussed int that chapter, the testing is mostly *black-box testing*: Testing the outwards functionality of the system, rather than the correctness of its internal logic. Only when debugging a very specific problem is it practical to expose enough of

the programs internal state that flaws in the logic can be revealed. The checks that throw exceptions littered throughout the code does provide some testing of the internal state, in a somewhat similar manner to asserts used in multiple automatic testing frameworks.

Testing was done continually throughout the project, gradually debugging and verifying the correctness of each part of the system. Testing followed parallel phases as the implementation described in section 7.5.3:

- During the developement of the first API, the Lejos library and Mindstorms components were continually tested. Observing what values was given when and such. In particular, the class SensorTest was used for this.

- The communication and signal set-up had to be tested extensively to determine messages where being sent and received properly. The CommTest was especially used here.

- The code for the SBs and TCCs are very dependent on each other and therefore hard to test separately. As such, most testing of these happened after almost all of the code for both had been written. A bit of testing were done on their early stubs however, to prove that the signal queue and state system could work. The SystemTest class was used primarily here, initializing the system according to the actual railway layout and faking signals sent from sensors. This could be considered a form of *integration testing*.

- The last phase of the implementation, improving the Lego API and getting the physical system to run, required the most testing. Very minute changes were being made to the code, while tons of test were run to track down the last bugs. The DetectorTest, SwitchTest and MotorTest was used first to each units correctness. Then AutomaticStart class was used to test the entire system, oft modified to test different set-ups: One specific train running, both doing their routes routes or repeating. This can be considered a type of *system testing*. I consider the best and "final" test to be whether any errors occur when both trains repeat their routes indefinitely, or at least more than five times. After deciding the code was finalized, I ran this test approximately fifteen times to determine the correctness of the final system.

8.2 Results

Of these last tests, about a quarter would fail quickly, the trains completing their route once or twice before stopping unexpectedly, not stopping at all, etc. Another quarter would run well for longer, around five rounds, before accumulating too much imprecision or suddenly failing. The remaining half ran completely successfully, running for as long as I allowed the trains to. This may not sound like an impressive statistic, but all failed runs seems to be the result of physical factors, problems with the sensors and motors, not the control algorithm. Under ideal circumstances, set up properly and with vigilance, the system can run quite consistently.

8.2.1 Known bugs & issues

The five remaining causes of problems in the system seem to be:

8.2.1.1 High ambient light levels

The system seems barely to barely work when there is too much light in the light. Direct sunlight through the windows is enough for the sensors not to be able to consistently distinguish a train from the general light level. This can perhaps be solved by further tuning, but in my experience, the in-station sensors are pretty much never going to accurate results in brightness. A possible solution could be to install an in-station sensor on each side of the track, as closer distance really helps.

8.2.1.2 Changes in light level

Despite using the flood light, the light sensors are quite sensitive to changes in the general light level. It is possible to see light values climb in real time during sun-rise or sunset, eventually throwing off the calibration of the sensors. Clouds moving in front of the sun can even be enough to disrupt the system in some cases. The solution would be for the sensors to recalibrate themselves periodically while the system is running. However, it is hard for the sensor to know when the ambient light has changed and when a train is parked in front of it at a weird angle. If the sensors were allowed to share data with each other or access the state of the control system, they might be able to know where the trains are and when it is safe to calibrate.

8.2.1.3 Changes in environment

Simply put, accidentally stepping in front of a sensor, or moving objects in its vicinity, might lead to it the sensors giving off wrong results. If one is careful, this one is not hard to prevent. Some shielding screens or such could be added to the railway, that provide a constant background for the sensors.

8.2.1.4 Simultaneous movement on the controller

As noted in section 3.1.1.6, the train remote does not do well when both dials are moved at the exact time. While the delay in starting times helped this problem tremendously, it can still happen at random. Getting another remote would fix this.

8.2.1.5 Remote motors not properly resetting

This error accrues over multiple runs of the program. The motors for the remote does not seem to always return to the position they started in, despite actually being programmed to overcompensate to get there. Due to how the motors and controller are clamped together, there is quite a bit of friction to reach the starting position, so this is likely due to the motor stalling out and being able to back to this position. As such the motor motor move a couple of degrees per run, until it starts away from the stop button and fails to properly control the train. The motor can easily be pushed back the correct starting position manually, however.

8.2.2 Documentation

I have recorded several videos to document both successful runs, the ways in which the program fails and the general evolution of the program in the last stages of implementation. The videos are therefore presented in chronological order.

First, a run where only one train is configured to start. At this point, the code and the algorithm itself was buggy, making both repeating routes and having two trains running at once immediately fail and working very inconsistently even with just one train. Note the lack of control car, allowing the sensors to sit closer to the rails without getting knocked off. This was also before the

trains were rebuilt to have only white bricks showing towards the railway side: https://youtu.be/o5Cq9Ms_WvY

Next video, has both trains running, one repeating and the other not. https://youtu.be/4GEq6qd60uE

In this video, both trains repeat their route indefinitely. Around the 1 minute mark, the issue discussed in 8.2.1.5 happens, causing one train to drive so slowly it loses momentum and stop entirely. This was before the start delay was implemented. https://youtu.be/FLMdGjr89pc

The system was also testing during late evenings and night, since no sun made the sensors more reliable. In this video, the issue discussed in 8.2.1.4 happens to both trains, and I manually reset the remote for one of the trains. https://youtu.be/ELu0eHqoP0s

Lastly, the best run recorded. No errors occur. Later during the video, I deliberately stall one of the trains to see if anything happens if the "rhythm" is broken and that the other train patiently waits. At the end, I instruct the trains to stop repeating their runs with a button press. https://youtu.be/lJGFt5K9rss

Chapter 9

Conclusion

I consider this project a minor success. While the final system does not work consistently, it does work and show that the distributed approach can work. There were some challenges in making the program distributed, but they in no way insurmountable.

I do believe creating the implementation for a model railway was valuable in creating understanding for how . Unfortunately, working with the low-quality Lego parts (as compared to the professional ones one would imagine they use in the actual railway industry) meant almost more time was spent wrestling with them than working on the distributed algorithm.

Appendix A

Project plan

The original project description were as follows:

Refer to figure A.1 to see the initial project plan. Note that the dates are starting points, not deadlines. So, FROM the 11. of February, the plan was to to "Developement of API for controlling/reading from Lego Parts".

The initial plan was very much that and quite pending. At the time, the lego railway system was designed only in broad strokes and it was not decided what modelling language to use. I knew at the time that the plan would be a rough estimate, as I did not have a clear grasp on how long the modelling or implementation would take. We quickly started revising the model as it also became clear it was not a logical order to tackle the project in. Once I had researched and tested the Lejos API enough to be confident in it's capabilities and constraints, it more relevant to do the modelling before the implementation. So the model could be a basis and inspiration for the program and the two more easily linked. Less than half-way the project the plan was revised to the one seen in A.2

This plan ultimately proved to be mostly a guideline as well. Some extra or overriding assignments came in some weeks, so I could get feedback on the work. With both plans I was also being deliberately being ambitious or underestimating how much time each task would take. As it is extremely easy to underestimate in the programming business, this means the time allotted ended up way

Plan & Schedule for the Master Thesis: Development and Visualisation of a Distributed Railway Control System by Mads Egedal Kirchhoff

All work and weeks includes note taking, documenting and/or writing relevant draft chapters for the thesis itself. Basically, no week should pass by there isn't some work being done on the report, in addition to implementation et.al.

Week:		
February 11	Developement of API for controlling/reading from Lego Parts	Lego Implementation and Algorithm Design
February 18	Sick week	
February 25	Finalizing API and initial set up of control capabilities & communication in the system	
March 3	Design of control algorithm and decide on modelling framework	
March 10	Full communication & threading setup	
March 17	Implementation of simple control (1 train) & scheduling	
March 24	Implementation of final control algorithm	
March 31	Bug fixing, stabilization and implementation buffer	
April 7	Nice-to-have features (allow changes in speed) and implementation buffer	
April 14	Testing and "informal" verification of correctness of system, bug fixes	Verification and Modelling (?)
April 21	Easter holiday, I am gone for most of it	
April 28	Finalized design of model/verification scheme	
May 5	Implementation of simple verification framework	
May 12	Final Implementation of verification scheme	
May 19	Proving of correctness of verification scheme and safety of the implemented system	
May 26	Research & Revising and writing out of notes, report structure and layout finalized	Thesis Writing
June 2	Introduction, Motivation, Theory written	
June 9	Design, Implementation, Verification, Results section written (most of these should be half-complete already before this poi	
June 16	Conclusion, Further Work, Abstract, bibliography and misc written. Finished first draft and sent for (content) feedback	
June 23	Revisions and second draft sent for feedback (including grammar & language).	
June 30	Final revisions and version, printing, binding and delivery	
July 7	Any work that takes longer than expected is deducted from these weeks	Buffer
July 14	Buffer	
July 21	Buffer	
July 28	Final week, final deadline 1 of august	

Figure A.1: Initial project plan

off from what was realistic. Modelling was done in about the planned time, but the control system implementation fell behind by about a month, as realizations during implementations required updates to the model and Lego API. I realized at the end that it would have been more helpful to have created the plan as a cycle, iterating through the API, model, implementation of the control system, at least twice. Further, the initially built API and the actual physical design of the system was in no way reliable enough to work with the algorithm. It took a lot of Lego assembling and dissembling and experimenting with values in the program for it to work decent, way after the bulk of the code was written. Debugging also took longer than expected, due to having to do it on the NXT. A lesson learned here is that physical components, such as sensors and motors, are inherently less reliable than a purely virtual environment, and they take longer to develop for.

Ultimately, the plan was quite useful for the project, even if was not closely followed. It acted as a guideline and a means to think about how best to tackle the project, in what order to and an incentive to get things done. I also think it was for the best that it was not kept to religiously, the week-to-week plan based on weekly meetings with my supervisors instead. It was quite predictable that the plan would not survive reality as I would contend that the only software project that is estimated a 100% correctly is one that is no challenge to the developer.

Plan & Schedule for the Master Thesis: Development and Visualisation of a Distributed Railway Control System by Mads Egedal Kirchhoff

All work and weeks includes note taking, documenting and/or writing relevant draft chapters for the thesis itself. Basically, no week should pass by there isn't some work being done on the report, in addition to implementation et.al.

Week:		
February 11	Developement of API for controlling/reading from Lego Parts	**Lego Environment and API Set Up**
February 18	Sick week	
February 25	Finalizing API and initial set up of control capabilities & communication in the system	
March 3	Design of control algorithm and decide on modelling framework	
March 10	Full communication & threading setup	
March 17	Chapters about Lego components, programming and the chosen system drafted.	**Thesis Writing**
March 24	Easter holiday, I am gone for most of it.	**Holiday**
March 31	Experimenting, (re)learning modelling tools and set up	**Modelling**
April 7	Draft control model	
April 14	Finalizing model, code generation and integrating with implementation, exporting diagrams and others	
April 21	Proving of correctness of model and implementation correlation and safety of the implemented system	
April 28	Implementation of simple control (1 train) & routes	**Control System Implementation**
May 5	Bulk code of final control algorithm with 2 trains written	
May 12	Bug fixing and stabilization	
May 19	Testing and informal verification of correctness of system, bug fixing	
May 26	Research & Revising and writing out notes and introduction written	**Thesis Writing**
June 2	Theory, Modelling, Implementation, Testing written (most should be half completed or fully drafted by then)	
June 9	Results, Conclusion, Further Work, Abstract, bibliography and misc written. Finished first draft and sent for (content) feedb	
June 16	Revisions and second draft sent for feedback (including grammar & language).	
June 23	Final revisions and version, printing, binding and delivery	
June 30	Any work that takes longer than expected is deducted from these weeks, or is otherwise spent on optional features	**Buffer or further work**
July 7	Further work: Time tables, invariant testing for implementation	
July 14	Further work: Allow changes in speed of trains or in schedules and the like while running	
July 21	Further work: Additional implementations of control models or alternative set-ups of stations and track network	
July 28	Final week, final deadline 1 of august	

Figure A.2: Initial project plan

Appendix B

Model Code

```
/* Track shape and naming is as such:
         1

       -------
   ___/_______\___
  / NW   2   NE \
  |             |
0 |             | 3
  | SW   4   SE |
  \_______________/
     \_______/
         5
Points/SwitchBoxes are designated by cardinal direction (North West =
    NW). Stations are referred to as outer (1,5) and inner (2,4).

So, a train could start at track 2, and if it wanted to go a full, inner
    clockwise loop,
it's trackDestinations would be: 3, 4, 0, 2.
*/

Class SwitchBox is
Signals
   tryReserve(track: int, fromStation : bool, train:
       TrainControlComputer, currentPosition : int),
   startReserver(reserverNo : int),
```

```
    syncReservation(trackToReserve : int, train : TrainControlComputer,
        otherSwitchBox : SwitchBox, priority : int),
    syncSuccesful,
    syncFail,
    trainBySensor(detectedTrain: TrainControlComputer),
    detectedTrain(train : TrainControlComputer),
    trainNoLongerDetected,
    clearReservation(track : int),
//Operations
Vars
    trackSb : SwitchBox; //The one across the single track
    stationSb : SwitchBox;

    singleTrack: int;
    stationTrackInner: int;
    stationTrackOuter: int;
    tracksReserved: TrainControlComputer[] //Accessed by track name.
        Would be a map in Java
    switchReserved: TrainControlComputer := Null;

    //Reserver variables, local scope of those "threads"
    pendingReservation: int[] = [-1,-1];
    pendingTrain : TrainControlComputer[] = [Null, Null];
    comingFromStation : bool[] = [true, true]; //Slight differences in
        the logic depending on whether the train is coming into or out
        from a station.
    trainPosition : int[] = [-1,-1];

    syncing:bool := False; //Failure if this is ever false
    reservingPriority : int; //Arbitrarily given, just used as a
        "tie-breaker" for simultaneous, conflicting track updates.
        Higher = higher priority. Could be a time-stamp in Java.

    switchPosition : int[]

State Top = SwitchBox
State SwitchBox = Receiver / Reserver1 / Reserver2 //A receiver for
    handling polling of the sensor and in-coming signals/messages from
    other components. And two Reserver thread for actually doing logic,
    send messages and such.
//State Receiver = Idle //Necessary to have Receiver be a composite
    state, since it changes what happens when transitioning into
    Reserver
State Reserver1 = Stopped //Starting state
State Reserver2 = Stopped //Starting state

Behavior
```

```
Receiver.Idle -> Receiver.Idle
    { tryReserve(track, fromStation, train, currentPosition) /
        reserverNo : int;
        if( pendingTrain[0] == Null) then { reserverNo = 0; }
        else { reserverNo = 1;
            if(pendingTrain[1] != Null) {OUT.BOTH_RESERVERS_BUSY();};
        };

        pendingReservation[reserverNo] := track;
        pendingTrain[reserverNo] := train;
        comingFromStation[reserverNo] = fromStation;
        trainPosition[reserverNo] = currentPosition;

        if(!comingFromStation[reserverNo]) then {
             pendingTrain[reserverNo].waitForReservation(); };
        self.startReserver(reserverNo);
        if(track != singleTrack && track != stationTrackInner && track
             != stationTrackOuter) then
            { OUT.TRAIN_CONTACTING_WRONG_SB(); };
    }

Receiver.Idle -> Receiver.Idle
    { syncReservation(trackToReserve, train, otherSwitchBox, priority) /
        if(tracksReserved[trackToReserve] == Null && (!syncing ||
             priority < reservingPriority) then {
            tracksReserved[trackToReserve] = train;
            otherSwitchBox.syncSuccesful();
        } else {
            otherSwitchBox.syncFail();
        };
    }

Receiver.Idle -> Receiver.Idle
    { clearReservation(track) /
        tracksReserved[track] = null;
    }

Receiver.Idle -> Receiver.Idle
    { detectedTrain(train)[switchReserved != train] /
        train.hailIncomingTrain(self);
    }


Reserver1.Stopped -> Reserver1.Checking
    { startReserver(reserverNo)[reserverNo == 0] /
    }
```

```
Reserver1.Checking -> Reserver1.Syncing
    { -[tracksReserved[pendingReservation[0]] == Null &&
        switchReserved == Null && !syncing] /
        syncing := true;

        if(comingFromStation[0]) then
            { trackSb.syncReservation(pendingReservation[0],
                pendingTrain[0], self, reservingPriority); }
        else
            { stationSb.syncReservation(pendingReservation[0],
                pendingTrain[0], self, reservingPriority); }
    }

Reserver1.Syncing -> Reserver1.WaitingForSensor
    { syncSuccesful[comingFromStation[0]] /
        if(tracksReserved[pendingReservation[0]] != Null) then {
            OUT.ALREADY_RESERVED( pendingReservation[0],
            tracksReserved[pendingReservation[0]] ); };
        if(switchReserved != Null) then { OUT.SWITCH_RESERVED(
            switchReserved ); };

        tracksReserved[pendingReservation[0]] = pendingTrain[0];
        switchReserved = pendingTrain[0];

        pendingTrain[0].reservationFree();
        pendingReservation[0] = -1;
        syncing = false;
        pendingTrain[0].timePasses()
    }

Reserver1.Syncing -> Reserver1.TrainDetected
    { syncSuccesful[!comingFromStation[0]] /
        if(tracksReserved[pendingReservation[0]] != Null) then {
            OUT.ALREADY_RESERVED( pendingReservation[0],
            tracksReserved[pendingReservation[0]] ); };
        if(switchReserved != Null) then { OUT.SWITCH_RESERVED(
            switchReserved ); };

        tracksReserved[pendingReservation[0]] := pendingTrain[0];
        switchReserved = pendingTrain[0];

        switchPosition[0] = singleTrack;
        switchPosition[1] = pendingReservation[0];
        OUT.turnPhysicalSwitch(switchPosition);

        pendingTrain[0].reservationFree();
        pendingReservation[0] = -1;
```

```
            syncing = false;
            pendingTrain[0].timePasses()
        }

    Reserver1.Syncing -> Reserver1.Checking
        { syncFail /
            if(!comingFromStation[0]) then
            { pendingTrain[0].waitForReservation(); };

            syncing = false;
        }

Reserver1.WaitingForSensor -> Reserver1.TrainDetected
    { detectedTrain(train)[] /
        pendingTrain[0].timePasses();
        if(switchReserved != train) then { OUT.WRONG_TRAIN_ON_SWITCH() };
    }

Reserver1.TrainDetected -> Reserver1.Stopped
    { trainNoLongerDetected /
        pendingTrain[0].updatePosition(stationSb);
        switchReserved = null;
        pendingTrain[0] = null;

        tracksReserved[trainPosition[0]] = null;
        if(comingFromStation[0]) then {
             stationSb.clearReservation(trainPosition[0]); }
        else { trackSb.clearReservation(trainPosition[0]); }
    }

//Precise copy of Reserver1, no new logic, just different places to
     store the "local" variables.
//Alternate solution: pass the "reserverNo" along in all signals and use
     it for the arrays. OR pass along all the needed variables.
Reserver2.Stopped -> Reserver2.Checking
    { startReserver(reserverNo)[reserverNo == 1] /
    }

    Reserver2.Checking -> Reserver2.Syncing
        { -[tracksReserved[pendingReservation[1]] == Null &&
             switchReserved == Null && !syncing] /
            syncing := true;

            if(comingFromStation[1]) then
                { trackSb.syncReservation(pendingReservation[1],
                     pendingTrain[1], self, reservingPriority); }
            else
```

```
            { stationSb.syncReservation(pendingReservation[1],
                pendingTrain[1], self, reservingPriority); }
    }

Reserver2.Syncing -> Reserver2.WaitingForSensor
    { syncSuccesful[comingFromStation[1]] /
        if(tracksReserved[pendingReservation[1]] != Null) then {
            OUT.ALREADY_RESERVED( pendingReservation[0],
            tracksReserved[pendingReservation[0]],1,0 ); };
        if(switchReserved != Null) then { OUT.SWITCH_RESERVED(
            switchReserved ); };

        tracksReserved[pendingReservation[1]] = pendingTrain[1];
        switchReserved = pendingTrain[1];

        pendingTrain[1].reservationFree();
        pendingReservation[1] = -1;
        syncing = false;
        pendingTrain[1].timePasses()
    }

Reserver2.Syncing -> Reserver2.TrainDetected
    { syncSuccesful[!comingFromStation[1]] /
        if(tracksReserved[pendingReservation[1]] != Null) then {
            OUT.ALREADY_RESERVED( pendingReservation[1],
            tracksReserved[pendingReservation[1]],pendingTrain[1],self
            ); };
        if(switchReserved != Null) then { OUT.SWITCH_RESERVED(
            switchReserved ); };

        tracksReserved[pendingReservation[1]] := pendingTrain[1];
        switchReserved = pendingTrain[1];

        switchPosition[0] = singleTrack;
        switchPosition[1] = pendingReservation[1];
        OUT.turnPhysicalSwitch(switchPosition);

        pendingTrain[1].reservationFree();
        pendingReservation[1] = -1;
        syncing = false;
        pendingTrain[1].timePasses()
    }

Reserver2.Syncing -> Reserver2.Checking
    { syncFail /
        if(!comingFromStation[1]) then
        { pendingTrain[1].waitForReservation(); };
```

```
            syncing = false;
        }

Reserver2.WaitingForSensor -> Reserver2.TrainDetected
    { detectedTrain(train)[] /
        pendingTrain[1].timePasses();
        if(switchReserved != train) then { OUT.WRONG_TRAIN_ON_SWITCH() };
    }

Reserver2.TrainDetected -> Reserver2.Stopped
    { trainNoLongerDetected /
        pendingTrain[1].updatePosition(stationSb);
        switchReserved = null;
        pendingTrain[1] = null;

        tracksReserved[trainPosition[1]] = null;
        if(comingFromStation[1]) then {
            stationSb.clearReservation(trainPosition[1]); }
        else { trackSb.clearReservation(trainPosition[1]); }
    }

end SwitchBox;


Class TrainControlComputer is
Signals
    waitForReservation,
    reservationFree,
    timePasses,
    stopForever,
    hailIncomingTrain(nearbySb : SwitchBox)
Operations
    updatePosition(otherSb : SwitchBox),
    stopIndefinitely()
Vars
    //Physical & logical variables
    position: int;
    running: bool := false;

    route: int[]; //The tracks to go to in order
    routeIterator: int = 0;
    repeat: bool := false;
    routeSwitchBoxes : SwitchBox; //Switchboxes in (supposed) order
        encountered. Only used by the physical system, the program
        doesn't actually have these references
```

```
    nextSwitchBox: SwitchBox;
    //otherSwitchBox: SwitchBox; TODO being able to reverse direction

State Top = Train
State Train = Physical / Logic
State Logic = AtStation
State Physical = NotDetected, Detected //Whether the phsyical train is
     in front of a sensor

Behavior
Logic.AtStation -> Logic.PendingReservation
    { -[] /
        nextSwitchBox.tryReserve(route[routeIterator], true, self,
            position);
    }

Logic.PendingReservation -> Logic.LeavingStation
    { reservationFree /
        running = true;
    }

Physical.NotDetected-> Physical.Detected
    { timePasses [running] /
        routeSwitchBoxes[routeIterator].detectedTrain(self);
    }

Physical.Detected-> Physical.NotDetected
    { timePasses [running] /
        routeSwitchBoxes[routeIterator].trainNoLongerDetected();
    }

Logic.LeavingStation -> Logic.TraversingSingleTrack
    { updatePosition(otherSb) /
        position = route[routeIterator];
        routeIterator++;

        if(routeIterator == route.length) then {
            if(repeat) then { routeIterator = 0; }
            else { self.stopIndefinitely(); };
        };
        self.timePasses();
        return
    }

Logic.TraversingSingleTrack -> Logic.BySensor
    { hailIncomingTrain(nearbySb) /
        nextSwitchBox = nearbySb;
```

```
        nextSwitchBox.tryReserve(route[routeIterator], false, self,
            position);
    }

Logic.BySensor -> Logic.BySensor
    { waitForReservation /
        running = false;
    }

Logic.BySensor -> Logic.EnteringStation
    { reservationFree /
        running = true;
    }

Logic.EnteringStation -> Logic.AtStation
    { updatePosition(otherSb) /
        //self.reservationFree(); //JDF
        //OUT.TRAIN_CONTACTING_WRONG_SB();

        nextSwitchBox = otherSb;
        running = false;

        position = route[routeIterator];
        routeIterator++;
        return;

        if(routeIterator == route.length) then {
            if(repeat) then { routeIterator = 0; }
            else { self.stopIndefinitely(); };
        };
    }

Logic.AtStation -> Logic.Ended
    { stopIndefinitely /
        running = false;
    }

end TrainControlComputer;

// object instatiations
sbNW: SwitchBox(singleTrack = 0, stationTrackInner = 2,
    stationTrackOuter = 1, tracksReserved =
    [Null,train1,Null,Null,train2,Null], trackSb = sbSW, stationSb =
    sbNE, reservingPriority = 1)
sbNE: SwitchBox(singleTrack = 3, stationTrackInner = 2,
    stationTrackOuter = 1, tracksReserved =
    [Null,train1,Null,Null,train2,Null], trackSb = sbSE, stationSb =
```

```
    sbNW, reservingPriority = 2)
sbSE: SwitchBox(singleTrack = 3, stationTrackInner = 4,
    stationTrackOuter = 5, tracksReserved =
    [Null,train1,Null,Null,train2,Null], trackSb = sbNE, stationSb =
    sbSW, reservingPriority = 3)
sbSW: SwitchBox(singleTrack = 0, stationTrackInner = 4,
    stationTrackOuter = 5, tracksReserved =
    [Null,train1,Null,Null,train2,Null], trackSb = sbNW, stationSb =
    sbSE, reservingPriority = 4)

//train1: TrainControlComputer (position=1, route=[3,5], repeat=false,
    nextSwitchBox=sbNE, routeSwitchBoxes=[sbNE,sbSE,sbSW,sbNW])
    //ClockWise
//train2: TrainControlComputer (position=4, route=[3,2], repeat=false,
    nextSwitchBox=sbSE, routeSwitchBoxes=[sbSE,sbNE,sbNW,sbSW])
    //CounterClockWise Out commented for easier graphing

train1: TrainControlComputer (position=1, route=[3,5,0,1], repeat=false,
    nextSwitchBox=sbNE, routeSwitchBoxes=[sbNE,sbSE,sbSW,sbNW])
    //ClockWise
train2: TrainControlComputer (position=4, route=[3,2,0,4], repeat=false,
    nextSwitchBox=sbSE, routeSwitchBoxes=[sbSE,sbNE,sbNW,sbSW])
    //CounterClockWise Out commented for easier graphing

//train1: TrainControlComputer (position=1, route=[3,5,0,1,3,5],
    repeat=false, nextSwitchBox=sbNE,
    routeSwitchBoxes=[sbNE,sbSE,sbSW,sbNW,sbNE,sbSE,sbSW,sbNW])
    //ClockWise
//train2: TrainControlComputer (position=4, route=[3,2,0,4,3,2],
    repeat=false, nextSwitchBox=sbSE,
    routeSwitchBoxes=[sbSE,sbNE,sbNW,sbSW,sbSE,sbNE,sbNW,sbSW])
    //CounterClockWise Out commented for easier graphing

//train1: TrainControlComputer (position=1,
    route=[3,5,0,1,3,5,0,1,3,5,0,1], repeat=false, nextSwitchBox=sbNE,
    routeSwitchBoxes=[sbNE,sbSE,sbSW,sbNW,sbNE,sbSE,sbSW,sbNW,sbNE,sbSE,sbSW,sbNW])
    //ClockWise
//train2: TrainControlComputer (position=4,
    route=[3,2,0,4,3,5,0,1,3,5,0,1], repeat=false, nextSwitchBox=sbSE,
    routeSwitchBoxes=[sbSE,sbNE,sbNW,sbSW,sbSE,sbNE,sbNW,sbSW,sbSE,sbNE,sbNW,sbSW])
    //CounterClockWise Out commented for easier graphing

Abstractions {
State:  train1.position = $1 and train2.position = $2 ->
    positions($1,$2)
//State:  train1.position = $1 -> position($1)
//State:  train1.position = $1 -> train1pos($1)//
```

```
//State:  train2.position = $1 -> train2pos($1)

//State:  train1.running = $1 and train2.running = $2 -> running($1,$2)
//State:  train1.routeIterator = $1 -> RouteIterator($1)
//State:  train1.nextSwitchBox = $1 -> nextSwitchBox($1)

//State sbSE.tracksReserved = $1 -> trackReservations($1)

/*
State: inState(sbNE.Reserver1.Syncing) -> sbNE_Syncing
State: inState(sbNE.Reserver2.Syncing) -> sbNE2_Syncing

State: inState(sbNE.Reserver1.Stopped) -> sbNE_Stopped
State: inState(sbNE.Reserver1.Checking) -> sbNE_Checking
State: inState(sbNE.Reserver1.Syncing) -> sbNE_Syncing
State: inState(sbNE.Reserver1.WaitingForSensor) -> sbNE_WaitingForSensor
State: inState(sbNE.Reserver1.TrainDetected) -> sbNE_TrainDetected

State: inState(sbNE.Reserver2.Stopped) -> sbNE2_Stopped
State: inState(sbNE.Reserver2.Checking) -> sbNE2_Checking
State: inState(sbNE.Reserver2.Syncing) -> sbNE2_Syncing
State: inState(sbNE.Reserver2.WaitingForSensor) -> sbNE2_WaitingForSensor
State: inState(sbNE.Reserver2.TrainDetected) -> sbNE2_TrainDetected

State: inState(sbSE.Reserver1.Stopped) -> sbSE_Stopped
State: inState(sbSE.Reserver1.Checking) -> sbSE_Checking
State: inState(sbSE.Reserver1.Syncing) -> sbSE_Syncing
State: inState(sbSE.Reserver1.WaitingForSensor) -> sbSE_WaitingForSensor
State: inState(sbSE.Reserver1.TrainDetected) -> sbSE_TrainDetected

State: inState(sbSW.Reserver1.Stopped) -> sbSW_Stopped
State: inState(sbSW.Reserver1.Checking) -> sbSW_Checking
State: inState(sbSW.Reserver1.Syncing) -> sbSW_Syncing
State: inState(sbSW.Reserver1.WaitingForSensor) -> sbSW_WaitingForSensor
State: inState(sbSW.Reserver1.TrainDetected) -> sbSW_TrainDetected

State: inState(train2.Logic.AtStation) -> train2_AtStation
State: inState(train2.Logic.PendingReservation) ->
    train2_PendingReservation
State: inState(train2.Logic.LeavingStation) -> train2_LeavingStation
State: inState(train2.Logic.TraversingSingleTrack) ->
    train2_TraversingSingleTrack
State: inState(train2.Logic.BySensor) -> train2_BySensor
State: inState(train2.Logic.EnteringStation) -> train2_EnteringStation
State: inState(train2.Logic.Ended) -> train1_Ended

//State: inState(train1.Physical.Detected) -> train1_Detected
```

```
State: inState(train1.Logic.AtStation) -> train1_AtStation
State: inState(train1.Logic.PendingReservation) ->
    train1_PendingReservation
State: inState(train1.Logic.LeavingStation) -> train1_LeavingStation
State: inState(train1.Logic.TraversingSingleTrack) ->
    train1_TraversingSingleTrack
State: inState(train1.Logic.BySensor) -> train1_BySensor
State: inState(train1.Logic.EnteringStation) -> train1_EnteringStation
State: inState(train1.Logic.Ended) -> train1_Ended

//Action: $obj:stopIndefinitely($*) -> stopIndefinitely($obj($*)
Action: $obj:trainNoLongerDetected($*) -> trainNoLongerDetected($obj($*)
Action: $obj:updatePosition($*) -> updatePosition($obj($*)

//Action  $obj:assign(running,$value) -> $obj(running,$value)
Action: $obj:startReserver($*) -> startReserver($obj($*)
*/

//Action: $obj:accept($1)    -> Accepted($obj,$1) -- observe dispatching
    of triggers
//Action: $obj:$obj2.$event($*) -> $event($obj=>$obj2($*)      --
    observe events with all their params

Action: TRAIN_CONTACTING_WRONG_SB($*) ->
    ERROR(TRAIN_CONTACTING_WRONG_SB,$*)
Action: BOTH_RESERVERS_BUSY($*) -> ERROR(BOTH_RESERVERS_BUSY,$*)
Action: WRONG_TRAIN_ON_SWITCH($*) -> ERROR(WRONG_TRAIN_ON_SWITCH,$*)
Action: ALREADY_RESERVED($*) -> ERROR(ALREADY_RESERVED,$*)
Action: SWITCH_RESERVED($*) -> ERROR(SWITCH_RESERVED,$*)

Action: $obj:lostevent($1) -> DISCARDED($obj,$1) -- observe discarding
    of triggers
Action: $obj:$obj2.Runtime_Error -> RUNTIME_ERROR($obj=>$obj2)

}

//MODEL CHECKING
//AG AF positions(1,4) AND not EF {ERROR} true AND not EF {DISCARDED}
    true AND AG [$1] AF {$2} true AND <true> AG not positions(0,0) AND
    <true> AG not positions(1,1) AND <true> AG not positions(2,2) AND
    <true> AG not positions(3,3) AND <true> AG not positions(4,4) AND
    <true> AG not positions(5,5)

//  EF {sig4($1)} EF {sig4(%1)} true -- action sig4 repeated twice with
    same arg
//  not EF {discarded} true         -- no signal is ever discarded
```

```
//  AG [$1] AF {$1} true            -- every signal sent is always
    accepted
//  <true> AG not initial(Obj1)     -- after the initial state, no other
    reachable state are labelled "initial(Obj)"
//  EF v1(2,2)
```

APPENDIX C

Implementation Code

C.1 ControlUnit

```
package controlSystem;

import java.util.LinkedList;

import legoAPI.Communicator.UnitId;
import signals.Signal;
import util.DebugMessage;

public abstract class ControlUnit extends Thread {

  public UnitId id;

  protected LinkedList<Signal> signals = new LinkedList<Signal>();

  public synchronized void addSignal(Signal signal) {
    signals.add(signal);
    notifyAll();
  }

  public synchronized boolean hasDuplicateSignal(Signal signal) {
    if (!signals.isEmpty() && signals.get(0).equals(signal)) {
```

```
            return true;
        }
        return false;
    }

    protected void waitForSignal() {
        try {

            while (this.signals.isEmpty())
                wait();

        } catch (InterruptedException e) {
            DebugMessage.showCatchMessage(e, "InterruptedException");
        }
    }

    protected synchronized Signal waitForSignalWithName(String
        signalName, String signalName2) {
        while (true) {
            waitForSignal();
            // Due to shadowing, this is the reservers queue, not the
                switchbox',
            // but "this" just in case
            Signal signal = this.signals.remove(0);
//          System.out.println("sig " + signal.getName());
            if (signal.getName().equals(signalName) ||
                signal.getName().equals(signalName2))
                return signal;
        }
    }

    protected synchronized Signal waitForSignalWithName(String
        signalName) {
        return waitForSignalWithName(signalName, null);
    }
}
```

C.2 SwitchBox

```
package controlSystem;

import java.util.ArrayList;
import java.util.Hashtable;
```

```java
import initialization.TrackLayout;
import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;
import legoAPI.Switch;
import signals.ClearReservation;
import signals.HailIncomingTrain;
import signals.ReservationFree;
import signals.Signal;
import signals.SyncFail;
import signals.SyncReservation;
import signals.SyncSuccesful;
import signals.TrainDetected;
import signals.TrainNoLongerDetected;
import signals.TryReserve;
import signals.UpdatePosition;
import signals.WaitForReservation;
import util.DebugMessage;

public class SwitchBox extends ControlUnit {

   private final UnitId trackSb; // The one across the single track
   private final UnitId stationSb;

   // Hashtable can have null
   private volatile ArrayList<Integer> tracks;
   private volatile Hashtable<Integer, UnitId> tracksReserved;
   private volatile UnitId switchReserved = null;

   // Arbitrarily given, just used as a "tie-breaker" for simultaneous,
   // conflicting track updates.
   // Higher = higher priority. Could be a time-stamp?
   private volatile Long syncingSince;
   private int[] switchPosition; // The tracks that the switch connects

   private Switch switcher;

   private volatile Reserver[] reservers = { null, null };

   public SwitchBox() {
      trackSb = null;
      stationSb = null;
   }

   public SwitchBox(UnitId id, ArrayList<Integer> tracks,
        Hashtable<Integer, UnitId> startReservations,
         int[] switchPosition, UnitId stationSb, UnitId trackSb, Switch
             switcher) {
```

```
      this.id = id;
      this.tracks = tracks;
      this.tracksReserved = startReservations;
      this.switchPosition = switchPosition;
      this.switcher = switcher;
      this.trackSb = trackSb;
      this.stationSb = stationSb;

      for (int i = 0; i < reservers.length; i++) {
         reservers[i] = new Reserver();
         reservers[i].id = this.id;
         reservers[i].start();
      }
      this.start();
   }

   public void run() {
      try {
         while (true) {
            receiverIdle();
         }
      } catch (Exception e) {
         DebugMessage.showCatchMessage(e, id.toString());
      }
   }

   // Idle
   private synchronized void receiverIdle() throws Exception {
      while (signals.isEmpty())
         wait();

      Signal signal = signals.remove(0);

      switch (signal.getName()) {
      case TryReserve.NAME:
         // TODO: Call methods by what they do, and sig by what happened
             (the
         // event)? Write in comments what the methods represent?
         TryReserve trySig = (TryReserve) signal;
         tryReserveIdle(trySig.track, trySig.train,
             trySig.currentPosition);
         break;
      case SyncReservation.NAME:
         SyncReservation syncSig = (SyncReservation) signal;
         SyncReservationIdle(syncSig.trackToReserve, syncSig.train,
             syncSig.otherSwitchBox, syncSig.otherSyncingSince);
         break;
```

```
        case ClearReservation.NAME:
          ClearReservation clearSig = (ClearReservation) signal;
          tracksReserved.put(clearSig.track, null);
          break;
        case TrainDetected.NAME:
          TrainDetected detectedSig = (TrainDetected) signal;
          trainDetectedIdle(detectedSig);
          break;
        default:
          for (Reserver reserver : reservers)
            reserver.addSignal(signal);
          break;
        }

        for (Reserver reserver : reservers)
          reserver.wakeUp();
      }

      private synchronized void tryReserveIdle(int track, UnitId train, int
          currentPosition) throws Exception {
        if (!tracks.contains(track))
          throw new Exception("Wrong SB " + id + " contacted! Track:" +
              track + " size" + tracks.size());

        for (int i = 0; i < reservers.length; i++) {
          if (reservers[i].reservingTrain == null) {
            reservers[i].tryReserve(track, train, currentPosition);
            break;
          } else if (i == reservers.length - 1)
            throw new Exception("All reservers busy!");
        }
      }

      private void SyncReservationIdle(int trackToReserve, UnitId train,
          UnitId otherSwitchBox, long otherSyncingSince) {
        DebugMessage.print(
            "syncRes " + trackToReserve + " t" + train + " sb" +
                otherSwitchBox + " s" + otherSyncingSince + "", this);
        if (tracksReserved.get(trackToReserve) == null && (syncingSince ==
            null || syncingSince > otherSyncingSince)) {
          tracksReserved.put(trackToReserve, train);
          Communicator.instance.sendSignal(otherSwitchBox, new
              SyncSuccesful());
        } else {
          Communicator.instance.sendSignal(otherSwitchBox, new
              SyncFail());
```

```
      }
   }

   private void trainDetectedIdle(TrainDetected signal) {
      DebugMessage.print("trainDete " + switchReserved + " " +
           tracksReserved.get(switchPosition[0]), this);
      if (!signal.sensorInStation && switchReserved == null) // &&
                                                                   //
                                                                        tracksRes
                                                                   //
                                                                        !=
                                                                        null)
         Communicator.instance.sendSignal(tracksReserved.get(switchPosition[0]),
              new HailIncomingTrain(id));
      else
         for (Reserver reserver : reservers)
            reserver.addSignal(signal);
   }

   protected class Reserver extends ControlUnit {
      // Creation and releasing of threads is quite costly, so we don't
      // release the instance for garbage collection

      protected UnitId reservingTrain = null;

      private int pendingReservation = -1;
      private boolean comingFromStation;
      private int trainPosition = -1;

      public synchronized void tryReserve(int track, UnitId train, int
           currentPosition) {
         reservingTrain = train; // pendingTrain
         pendingReservation = track;
         trainPosition = currentPosition;

         if (TrackLayout.isStationTrack(currentPosition)) {
            comingFromStation = true;
            // only if reservation not avaliable?
         } else
            comingFromStation = false;

         // StartReserver signal
         notifyAll();
      }

      public void run() {
         try {
```

```
            while (true)
               stopped();
         } catch (Exception e) {
            DebugMessage.showCatchMessage(e, "Reserver",
                  "HomemadeException");
         }

      }

      private synchronized void stopped() throws Exception {
         while (reservingTrain == null)
            wait();
         DebugMessage.print("startReserver", SwitchBox.this);
         checking();
      }

      private synchronized void checking() throws Exception {
         while (tracksReserved.get(pendingReservation) != null ||
               switchReserved != null || syncingSince != null)
            wait();

         syncingSince = System.currentTimeMillis();

         SyncReservation signal = new
               SyncReservation(pendingReservation, reservingTrain, id,
               syncingSince);
         if (comingFromStation)
            Communicator.instance.sendSignal(trackSb, signal);
         else
            Communicator.instance.sendSignal(stationSb, signal);

         syncing();
      }

      private void syncing() throws Exception {
         Signal signal = waitForSignalWithName(SyncSuccesful.NAME,
               SyncFail.NAME);

         if (signal.getName().equals(SyncSuccesful.NAME)) {
            if (tracksReserved.get(pendingReservation) != null)
               throw new Exception("Already reserved " +
                     pendingReservation);
            if (switchReserved != null)
               throw new Exception("Switch reserved " +
                     pendingReservation);

            tracksReserved.put(pendingReservation, reservingTrain);
```

```
            switchReserved = reservingTrain;

            if (!comingFromStation) {
               switchPosition[0] = trainPosition;
               switchPosition[1] = pendingReservation;
               switcher.switchTrack(switchPosition[1]);
            }
//          else {
//             switchPosition[0] = pendingReservation;
//             switchPosition[1] = trainPosition;
//             switcher.switchTrack(switchPosition[1]);
//          }

            this.signals.clear(); //Bit of a dirty hack, but it works

            Communicator.instance.sendSignal(reservingTrain, new
                ReservationFree());
            pendingReservation = -1;
            syncingSince = null;

            trainbySensor();

         } else if (signal.getName().equals(SyncFail.NAME)) {
            // Communicator.instance.sendSignal(stationSb, signal);
                //Redundant
            DebugMessage.print("syncFail back", this);
            if (!comingFromStation)
               Communicator.instance.sendSignal(reservingTrain, new
                   WaitForReservation());

            syncingSince = null;
            // Rapidly repeating recheckings can lead to stackoverflow
            // Thread.sleep(5000);
            wait();
            checking();
         }
      }

      private synchronized void trainbySensor() throws Exception {
         while (true) {
            TrainNoLongerDetected signal = (TrainNoLongerDetected)
                waitForSignalWithName(TrainNoLongerDetected.NAME);
            if (signal.sensorInStation != comingFromStation)
               break;
         }
         DebugMessage.print("trainLeft " + comingFromStation, this);
```

```
        if (!comingFromStation)
           Thread.sleep(300);
        tracksReserved.put(trainPosition, null);
        switchReserved = null;
        Communicator.instance.sendSignal(reservingTrain, new
            UpdatePosition());
        reservingTrain = null;
        if (comingFromStation)
           Communicator.instance.sendSignal(stationSb, new
               ClearReservation(trainPosition));
        else
           Communicator.instance.sendSignal(trackSb, new
               ClearReservation(trainPosition));

        for (Reserver reserver : reservers)
           reserver.wakeUp();
     }

     private synchronized void wakeUp() {
        // Recommended over notify by the internet
        notifyAll();
     }
  }
}
```

C.3 TrainControlComputer

```
package controlSystem;

import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;
import legoAPI.TrainMotor;
import signals.HailIncomingTrain;
import signals.ReservationFree;
import signals.Signal;
import signals.TryReserve;
import signals.UpdatePosition;
import signals.WaitForReservation;
import util.DebugMessage;

public class TrainControlComputer extends ControlUnit {

  private static final int STATION_STOP_TIME = 2500;
```

```
private int position;

public TrainMotor trainMotor;

private int[] route; // The tracks to go to in order
public UnitId[] routeSwitchBoxes; // The SwitchBoxes to contact in
    order
public int routeIterator = 0;

public Boolean repeatRoute = true;

public TrainControlComputer(UnitId id, int startingPosition,
    TrainMotor trainMotor, int[] route,
     UnitId[] routeSwitchBoxes, boolean repeatRoute) {
  this.id = id;
  position = startingPosition;
  this.trainMotor = trainMotor;
  this.route = route;
  this.routeSwitchBoxes = routeSwitchBoxes;
  this.repeatRoute = repeatRoute;
}

public void run() {
  try {
    while (!routeCompleted())
      atStation();
    // implicitly enters StopIndefinitely state if loop terminated,
        as the
    // thread closes
  } catch (Exception e) {
    trainMotor.stop();
    DebugMessage.showCatchMessage(e, id.toString());
  }
}

private synchronized void atStation() throws Exception {
  Communicator.instance.sendSignal(routeSwitchBoxes[routeIterator],
      new TryReserve(route[routeIterator], id, position));

  DebugMessage.print("PendingRes" + position, this);
  pendingReservation();
}

private synchronized void pendingReservation() throws Exception {
  waitForSignalOfType(ReservationFree.class);

  trainMotor.start();
```

```
        DebugMessage.print("leaving", this);
        leavingStation();
    }

    private synchronized void leavingStation() throws Exception {
        waitForSignalOfType(UpdatePosition.class);

        position = route[routeIterator];
        routeIterator++;
        if (repeatRoute)
            routeIterator %= route.length;

        DebugMessage.print("traversing " + position + " [" + routeIterator
                + "]=" + route[routeIterator], this);
        if (routeCompleted())
            return;
        else
            traversingSingleTrack();
    }

    private synchronized void traversingSingleTrack() throws Exception {
        HailIncomingTrain signal =
                waitForSignalOfType(HailIncomingTrain.class);
        DebugMessage.print("hailed" + routeSwitchBoxes[routeIterator],
                this);

        trainMotor.stop();
        if (signal.nearbySb != routeSwitchBoxes[routeIterator])
            throw new Exception("Hailed by wrong SB! Should be " +
                    routeSwitchBoxes[routeIterator] + " is " + signal.nearbySb);
        Communicator.instance.sendSignal(routeSwitchBoxes[routeIterator],
                new TryReserve(route[routeIterator], id, position));

        bySensor();
    }

    private synchronized void bySensor() throws Exception {
        while (true) {
            Signal signal = waitForSignalWithName(WaitForReservation.NAME,
                    ReservationFree.NAME);

            DebugMessage.print("bySensor " + signal.getName(), this);

            if (signal.getName().equals(WaitForReservation.NAME))
                trainMotor.stop();
            else {
```

```
                trainMotor.start();
                DebugMessage.print("entering ", this);
                enteringStation();
                break;
            }
        }
    }

    private synchronized void enteringStation() throws
          InterruptedException {
        waitForSignalOfType(UpdatePosition.class);
        position = route[routeIterator];
        routeIterator++;

        DebugMessage.print("entered " + routeIterator + " " +
              (routeIterator % route.length), this);
        if (repeatRoute)
            routeIterator %= route.length;

        trainMotor.stop();
        Thread.sleep(STATION_STOP_TIME);
        // Unwinds the stack, eventually returning to the central loop and
              calling
        // atStation again
        return;
    }

    private boolean routeCompleted() {
        return routeIterator == route.length;
    }

    @SuppressWarnings("unchecked")
    private <T extends Signal> T waitForSignalOfType(Class<T> clazz) {
        waitForSignal();

        Signal answer = signals.remove(0);
        return (T) answer;
    }
}
```

C.4 AutomaticStart

```
package initialization;
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Hashtable;

import controlSystem.ControlUnit;
import controlSystem.SwitchBox;
import controlSystem.TrainControlComputer;
import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;
import legoAPI.Detector;
import legoAPI.Switch;
import legoAPI.TrainMotor;
import lejos.nxt.Button;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.nxt.comm.Bluetooth;
import util.DebugMessage;

public class AutomaticStart {

   // @formatter:off
   /* Track shape and naming is as such:
         1
         -------
   ___/_______\___
  /  NW   2   NE  \
  |               |
 0|               | 3
  |  SW   4   SE  |
  \_______________/
      \_______/
         5
Points/SwitchBoxes are designated by cardinal direction (North West =
    NW). Stations are referred to as outer (1,5) and inner (2,4).
Train 1 starts at 1 and train 2 at 4. Switches connect to the inner
    tracks (2, 4)
*/
   // @formatter:on

   public static void main(String[] args) throws IOException,
       InterruptedException {
      // Debug.disabled = true;
      // Debug.printSwitchBox = true;
      // Debug.printAPI = true;
      // DebugMessage.printTrainControlComputer = true;
      System.out.println("Started");
```

```
if (Communicator.WestNxtName.equals(Bluetooth.getFriendlyName())) {

   // Initializing SBNW
   Hashtable<Integer, UnitId> tracksReservedSBNW = new
       Hashtable<Integer, UnitId>();
   tracksReservedSBNW.put(1, UnitId.TccW);

   ArrayList<Integer> tracksNW = new ArrayList<Integer>();
   tracksNW.add(0);
   tracksNW.add(1);
   tracksNW.add(2);

   Switch switchNW = new Switch(Motor.A, 2);

   SwitchBox switchBoxNW = new SwitchBox(UnitId.SbNW, tracksNW,
       tracksReservedSBNW, new int[] { 0, 2 }, UnitId.SbNE,
        UnitId.SbSW, switchNW);

   Detector detectorNwStation = new Detector(SensorPort.S1,
       switchBoxNW, true);
   Detector detectorNwEntrance = new Detector(SensorPort.S2,
       switchBoxNW, false);

   // Initializing SBSW
   Hashtable<Integer, UnitId> tracksReservedSBSW = new
       Hashtable<Integer, UnitId>();
   tracksReservedSBSW.put(0, null);
   tracksReservedSBSW.put(4, UnitId.TccE);
   tracksReservedSBSW.put(5, null);

   ArrayList<Integer> tracksSW = new ArrayList<Integer>();
   tracksSW.add(0);
   tracksSW.add(4);
   tracksSW.add(5);

   Switch switchSW = new Switch(Motor.C, 4);

   SwitchBox switchBoxSW = new SwitchBox(UnitId.SbSW, tracksSW,
       tracksReservedSBSW, new int[] { 0, 4 }, UnitId.SbSE,
        UnitId.SbNW, switchSW);

   Detector detectorSWentrance = new Detector(SensorPort.S3,
       switchBoxSW, false);
   Detector detectorSWstation = new Detector(SensorPort.S4,
       switchBoxSW, true);

   // Initializing train1
```

```
        int[] route = { 3, 5, 0, 1 };
        UnitId[] routeSwitchBoxes = { UnitId.SbNE, UnitId.SbSE,
            UnitId.SbSW, UnitId.SbNW };

        TrainMotor trainMotor = new TrainMotor(Motor.B, true);
        TrainControlComputer train1 = new
            TrainControlComputer(UnitId.TccW, 1, trainMotor, route,
            routeSwitchBoxes,
             true);

        ControlUnit[] units = { train1, switchBoxNW, switchBoxSW };

        Communicator communicator = new Communicator(units);
        train1.start();

        control(switchNW, switchSW, trainMotor, train1);
    } else {

        // Initializing SBNE
        Hashtable<Integer, UnitId> tracksReservedSBNE = new
            Hashtable<Integer, UnitId>();
        tracksReservedSBNE.put(1, UnitId.TccW);

        ArrayList<Integer> tracksNE = new ArrayList<Integer>();
        tracksNE.add(1);
        tracksNE.add(2);
        tracksNE.add(3);

        Switch switchNE = new Switch(Motor.C, 2);

        SwitchBox switchBoxNE = new SwitchBox(UnitId.SbNE, tracksNE,
            tracksReservedSBNE, new int[] { 3, 2 }, UnitId.SbNW,
             UnitId.SbSE, switchNE);

        Detector detectorNeEntrance = new Detector(SensorPort.S3,
            switchBoxNE, false);
        Detector detectorNeStation = new Detector(SensorPort.S4,
            switchBoxNE, true);

        // Initializing SBSE
        Hashtable<Integer, UnitId> tracksReservedSBSE = new
            Hashtable<Integer, UnitId>();
        tracksReservedSBSE.put(4, UnitId.TccE);

        ArrayList<Integer> tracksSE = new ArrayList<Integer>();
        tracksSE.add(3);
        tracksSE.add(4);
```

```
      tracksSE.add(5);

      Switch switchSE = new Switch(Motor.A, 4);

      SwitchBox switchBoxSE = new SwitchBox(UnitId.SbSE, tracksSE,
          tracksReservedSBSE, new int[] { 3, 4 }, UnitId.SbSW,
           UnitId.SbNE, switchSE);

      Detector detectorSeEntrance = new Detector(SensorPort.S2,
          switchBoxSE, false);
      Detector detectorSeStation = new Detector(SensorPort.S1,
          switchBoxSE, true);

      // Initializing train2
      int[] route = { 3, 2, 0, 4 };
      UnitId[] routeSwitchBoxes2 = { UnitId.SbSE, UnitId.SbNE,
          UnitId.SbNW, UnitId.SbSW };

      TrainMotor trainMotor = new TrainMotor(Motor.B, false);
      TrainControlComputer train2 = new
          TrainControlComputer(UnitId.TccE, 4, trainMotor, route,
          routeSwitchBoxes2,
           true);

      ControlUnit[] units = { train2, switchBoxSE, switchBoxNE };
      Communicator communicator = new Communicator(units);
      train2.start();

      control(switchNE, switchSE, trainMotor, train2);
   }
}

private static void control(Switch switch1, Switch switch2,
     TrainMotor motor1, TrainControlComputer tcc) {
   System.out.println("EscapeBtn exits");
   while (true) {
      switch (Button.waitForAnyPress()) {
      case Button.ID_ESCAPE:
         tcc.interrupt();
         motor1.stop();
         switch1.switchOff();
         switch2.switchOff();
         System.exit(0);
         break;
      case Button.ID_ENTER:
         tcc.repeatRoute = false;
         if (!tcc.isAlive()) {
```

```
            System.out.println("Started train again");
            tcc.start();
         }
         break;
      case Button.ID_RIGHT:
         tcc.repeatRoute = true;
         System.out.println("Train repeating");
         if (!tcc.isAlive()) {
            tcc.start();
         }
      case Button.ID_LEFT:
         tcc.repeatRoute = false;
         System.out.println("Train not repeating");
      }
   }
}

}
```

C.5 TrackLayout

```
package initialization;

public class TrackLayout {

   // @formatter:off
   /* Track shape and naming is as such:
        1
        -------
   ___/_______\___
 /  NW  2   NE  \
 |              |
0|              | 3
 |  SW  4   SE  |
 \_______________/
     \_______/
        5
Points/SwitchBoxes are designated by cardinal direction (North West =
    NW). Stations are referred to as outer (1,5) and inner (2,4).
Train 1 starts at 1 and train 2 at 4.
*/
   // @formatter:on

   public static boolean isStationTrack(int track) {
```

```
    if (track == 0 || track == 3) {
      return false;
    }
    return true;
  }

}
```

C.6 Communicator

```
package legoAPI;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.Hashtable;

import controlSystem.ControlUnit;
import lejos.nxt.comm.BTConnection;
import lejos.nxt.comm.Bluetooth;
import signals.ClearReservation;
import signals.HailIncomingTrain;
import signals.ReservationFree;
import signals.Signal;
import signals.SyncFail;
import signals.SyncReservation;
import signals.SyncSuccesful;
import signals.TryReserve;
import signals.UpdatePosition;
import signals.WaitForReservation;
import util.DebugMessage;

public class Communicator extends Thread {

  public enum UnitId {
    TccW, TccE, SbNW, SbNE, SbSE, SbSW,
  }

  public static Communicator instance;

  private Hashtable<UnitId, ControlUnit> localControlUnits = new
      Hashtable<UnitId, ControlUnit>();

  private static DataInputStream reader;
```

```
private static DataOutputStream writer;

// West is synonymous with left, has a pin on left side of the brick,
    and
// controls the left dial of the train remote
private boolean isWestNxt;
public static final String WestNxtName = "WestRailway";
public static final String EastNxtName = "EastRailway";

public Communicator(ControlUnit[] controlUnits) throws IOException,
    InterruptedException {

  for (ControlUnit controlUnit : controlUnits)
    localControlUnits.put(controlUnit.id, controlUnit);

  isWestNxt = checkOrSetNxtIdentity();

  DebugMessage.print("Conn NXTs", this, true);

  makeConnection();

  instance = this;

  this.start();
}

private boolean checkOrSetNxtIdentity() throws IOException {
  // TODO: Set this up to be given in the constructor
  if (WestNxtName.equals(Bluetooth.getFriendlyName()))
    return true;
  else
    return false;
}

private void makeConnection() throws IOException,
    InterruptedException {
  Bluetooth.setVisibility((byte) 1); // Set visibility on
  BTConnection connection = null;
  boolean mutualConnectionConfirmed = false;

  if (isWestNxt) {
    while (connection == null || !mutualConnectionConfirmed) {
      DebugMessage.print("Wait for conn", this, true);
      connection = Bluetooth.waitForConnection();
      if (connection == null)
        continue;
      DebugMessage.print("Got conn", this, true);
```

```
            writer = connection.openDataOutputStream();
            reader = connection.openDataInputStream();
            writer.writeBoolean(true);
            writer.flush();
            Thread.sleep(500);
            if (reader.available() != 0)
               mutualConnectionConfirmed = reader.readBoolean();
         }
      } else {
         while (connection == null || !mutualConnectionConfirmed) {
            DebugMessage.print("Try conn ", this, true);
            connection =
                Bluetooth.connect(Bluetooth.getKnownDevice(WestNxtName));
            if (connection == null) {
               Thread.sleep(1000);
               continue;
            }
            reader = connection.openDataInputStream();
            writer = connection.openDataOutputStream();
            Thread.sleep(250);

            if (reader.available() != 0) {
               mutualConnectionConfirmed = reader.readBoolean();
               writer.writeBoolean(true);
               writer.flush();
               // }
            } else {
               reader.close();
               writer.close();
               connection.close();
            }
         }
      }
   }

   public synchronized void sendSignal(UnitId recipient, Signal signal) {
      try {

         if (localControlUnits.get(recipient) != null) {
            localControlUnits.get(recipient).addSignal(signal);
            DebugMessage.print("L sent " + signal.getName() + " to " +
                recipient, this);
         } else {
            writer.writeUTF(recipient.toString());
            writer.writeUTF(signal.getName());
            signal.writeSignal(writer);
            writer.flush();
```

```
            DebugMessage.print("Sent " + signal.getName() + " to " +
                recipient, this);
         }

      } catch (IOException e) {
         DebugMessage.showCatchMessage(e, "Communicator sending " +
             signal.getName());
      }
   }

   public void run() {
      try {
         DebugMessage.print("Signal Listening", this, true);
         while (true) {
            listenForSignal();
         }
      } catch (Exception e) {
         DebugMessage.showCatchMessage(e, "Communicator");
      }
   }

   private void listenForSignal() throws Exception {
      String receiverStr = reader.readUTF();
      UnitId recipient = parseId(receiverStr);
      String signalName = reader.readUTF();

      Signal signal;
      switch (signalName) {
      case ClearReservation.NAME:
         signal = ClearReservation.readSignal(reader);
         break;
      case HailIncomingTrain.NAME:
         signal = HailIncomingTrain.readSignal(reader);
         break;
      case ReservationFree.NAME:
         signal = ReservationFree.readSignal(reader);
         break;
      case SyncFail.NAME:
         signal = SyncFail.readSignal(reader);
         break;
      case SyncReservation.NAME:
         signal = SyncReservation.readSignal(reader);
         break;
      case SyncSuccesful.NAME:
         signal = SyncSuccesful.readSignal(reader);
         break;
      case TryReserve.NAME:
```

```
            signal = TryReserve.readSignal(reader);
            break;
         case UpdatePosition.NAME:
            signal = UpdatePosition.readSignal(reader);
            break;
         case WaitForReservation.NAME:
            signal = UpdatePosition.readSignal(reader);
            break;
         default:
            throw new IOException("Unrecognized " + signalName + " signal
                received!");
         }
         DebugMessage.print("Got " + signal.getName() + " to " + recipient,
             this);

         ControlUnit controlUnit = localControlUnits.get(recipient);

         if (controlUnit == null)
            throw new Exception("Non-local " + recipient + "! Has " +
                localControlUnits.get(0));

         controlUnit.addSignal(signal);
      }

      public static UnitId parseId(String name) throws Exception {
         for (UnitId unitId : UnitId.values()) {
            if (unitId.toString().equals(name)) {
               return unitId;
            }
         }
         throw new Exception("Id not found");
      }

}
```

C.7 Detector

```
package legoAPI;

import java.io.IOException;

import controlSystem.SwitchBox;
import lejos.nxt.LightSensor;
import lejos.nxt.SensorPort;
```

```
import signals.TrainDetected;
import signals.TrainNoLongerDetected;
import util.DebugMessage;

public class Detector extends Thread {

   private final int SLEEP_BETWEEN_POLLING = 25;
   private final int MIN_TRAIN_PASS_TIME = 300;

   // 10-ish if just the rear or tip is showing, up to 200.
   private final int CLOSE_TRAIN_VALUE = 50;
   private final int FAR_TRAIN_VALUE = 15;

   // Unless ambient light is changing (such as sunrise, sunset), the
       base
   // value barely changes, only by 0-2 points, and up to 5 rarely.
   private final int BASE_VARIANCE_MARGIN = 8;

   private int trainValueDiff;

   // ~600 at cloudy noon
   // ~750 at dark cloudy noon
   // ~840 past dusk
   // ~400 sun through curtains
   private int baseValue;

   private SwitchBox parentSwitchbox;

   private boolean inStation;
   private SensorPort sensorPort;

   public Detector(SensorPort sensorPort, SwitchBox parentSwitchbox,
       boolean inStation) throws InterruptedException {
      this.sensorPort = sensorPort;
      new LightSensor(sensorPort, true);
      this.parentSwitchbox = parentSwitchbox;
      this.inStation = inStation;

      Thread.sleep(400); // Necessary to let floodlight take effect

      baseValue = sensorPort.readRawValue();

      trainValueDiff = inStation ? FAR_TRAIN_VALUE : CLOSE_TRAIN_VALUE;

      this.start();
   }
```

```
public void run() {
   try {
      TrainDetected signalDetected = new TrainDetected(inStation);
      TrainNoLongerDetected signalNotDetected = new
          TrainNoLongerDetected(inStation);

      String debugString = (inStation ? "In" : "Out") +
          parentSwitchbox.id.toString().substring(2);

      DebugMessage.print(debugString + " base " + baseValue, this,
          true);

      while (true) {

         while (!trainDetected())
            Thread.sleep(SLEEP_BETWEEN_POLLING);

         DebugMessage.print(debugString + " detec " +
             normalizedValue(), this);

         parentSwitchbox.addSignal(signalDetected);

         Thread.sleep(MIN_TRAIN_PASS_TIME);

         // Wait for train to pass before restarting
         while (true) {
            Thread.sleep(SLEEP_BETWEEN_POLLING);
            if (trainNotDetected()) {
            //double check, there can be dark spots on the train
               Thread.sleep(SLEEP_BETWEEN_POLLING * 2);
               if (trainNotDetected())
                  break;
            }
         }

         parentSwitchbox.addSignal(signalNotDetected);
         DebugMessage.print(debugString + " lost " +
             normalizedValue(), this);

         Thread.sleep(MIN_TRAIN_PASS_TIME);
      }

   } catch (
   InterruptedException e) {
      e.printStackTrace();
   } catch (IOException e) {
      e.printStackTrace();
```

```
        }
    }

    // No abs, train white
    private boolean trainDetected() {
        return normalizedValue() >= trainValueDiff;
    }

    // Negative
    private boolean trainNotDetected() {
        return normalizedValue() < BASE_VARIANCE_MARGIN;
    }

    private int normalizedValue() {
        return baseValue - sensorPort.readRawValue();
    }

}
```

C.8 Switch

```
package legoAPI;

import lejos.nxt.NXTRegulatedMotor;
import util.DebugMessage;

public class Switch {

    //The "position" being the number/id of the track switched too
    public int switchPosition;
    private int initialSwitchPosition;

    private final int degrees = 90;
    private final int speed = 300;

    private NXTRegulatedMotor motor;

    public void switchTrack(int newSwitchPosition) throws
         InterruptedException {
        DebugMessage.print("Switch to" + newSwitchPosition, this);
        if(switchPosition!=newSwitchPosition){
        if (initialSwitchPosition == newSwitchPosition)
            motor.rotateTo(0);
        else
```

```
            motor.rotateTo(-degrees);
        }
    }

    public void switchOff() {
        motor.rotateTo(0);
    }

    public Switch(NXTRegulatedMotor motorPort, int initialPosition) {
        motor = motorPort;
        this.initialSwitchPosition = initialPosition;
        motor.setSpeed(speed);
        this.switchPosition = initialPosition;
    }
}
```

C.9 TrainMotor

```
package legoAPI;

import lejos.nxt.NXTRegulatedMotor;
import util.DebugMessage;

public class TrainMotor {

    public boolean running;

    public int defaultSpeedDegrees = 45;

    private NXTRegulatedMotor motor;

    private boolean leftTrain;

    private int directionModifier = -1;

    public void stop() {
        DebugMessage.print("Stop motor", this);
        motor.rotateTo(directionModifier * -4);
        running = false;
        motor.flt(); // Allows manual reconfiguring/turning of the motor
    }

    public void start() {
        DebugMessage.print("Start motor", this);
```

```
        if (running)
            DebugMessage.showCatchMessage(new Exception((leftTrain ? "Left"
                : "Right") + " motor already started!"), "");

        try {
        // Remote cannot handle when one train has to be started/stopped
             at the same time
            if(leftTrain)
            Thread.sleep(500);
            else
                Thread.sleep(1500);
        } catch (InterruptedException e) {
        }
        int limitAngle = directionModifier * defaultSpeedDegrees;
        motor.rotateTo(limitAngle);
        running = true;
        motor.flt(); // Allows manual reconfiguring of the motor
    }

    public TrainMotor(NXTRegulatedMotor motorPort, boolean leftTrain) {
        this.motor = motorPort;
        this.leftTrain = leftTrain;
        motor.setAcceleration(1500);// 2000
        motor.setSpeed(150);// 200
        // The right train is for whatever reasons (likely the battery
             being
        // different) faster than the other
        // if (!leftTrain)
        // defaultSpeedDegrees -= 10;
        // directionModifier = (leftTrain ? 1 : 1);
    }

}
```

C.10 RemoteCalibrator

```
package nxtUtil;

import legoAPI.TrainMotor;
import lejos.nxt.Button;
import lejos.nxt.Motor;

public class RemoteCalibrator {
```

```
   public static void main(String[] args) throws InterruptedException {
      TrainMotor trainMotor = new TrainMotor(Motor.B, true);
      System.out.println("Wait for calibration.");
      while (true) {
         for (int i = 0; i < 3; i++) {
            trainMotor.start();
            Thread.sleep(100);
            trainMotor.stop();
         }

         System.out.println("If train stopped, calibr success, press
             back. Else, stop train and left/right btn");
         Motor.B.flt(); //Allows user to turn the dial themselves
         int btnPressed = Button.waitForAnyPress();
         if (btnPressed == Button.ID_ESCAPE)
            System.exit(0);
         else if (btnPressed == Button.ID_LEFT){
            Motor.B.resetTachoCount();
         }
         else if (btnPressed == Button.ID_RIGHT){
            Motor.B.rotate(-2);
            Motor.B.resetTachoCount();
         }
         //Else, if it's enter, try again, as the user might have turned
             the dial
      }
   }
}
```

C.11 SetNameEast

```
package nxtUtil;

import legoAPI.Communicator;
import lejos.nxt.comm.Bluetooth;

public class SetNameEast {
   public static void main(String[] args) {
      Bluetooth.setFriendlyName(Communicator.EastNxtName);
   }
}
```

C.12 SetNameWest

```
package nxtUtil;

import legoAPI.Communicator;
import lejos.nxt.comm.Bluetooth;

public class SetNameWest {
   public static void main(String[] args) {
      Bluetooth.setFriendlyName(Communicator.WestNxtName);
   }
}
```

C.13 ClearReservation

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class ClearReservation extends SwitchBoxSignal {

   public final static String NAME = "ClearReservation";

   public int track;

   public ClearReservation(int track) throws IOException {
      this.track = track;
   }

   public static Signal readSignal(DataInputStream reader) throws
       Exception {
      return new ClearReservation(reader.readInt());
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
      writer.writeInt(track);
   }

   public String getName() {
      return NAME;
   }
```

```
}
```

C.14 HailIncomingTrain

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;

public class HailIncomingTrain extends TrainControlComputerSignal {

   public final static String NAME = "HailIncomingTrain";

   public UnitId nearbySb;

   public HailIncomingTrain(UnitId nearbySb) {
      this.nearbySb = nearbySb;
   }

   public static Signal readSignal(DataInputStream reader) throws
        Exception {
      return new
           HailIncomingTrain(Communicator.parseId(reader.readUTF()));
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
      writer.writeUTF(nearbySb.toString());
   }

   public String getName() {
      return NAME;
   }

}
```

C.15 ReservationFree

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class ReservationFree extends TrainControlComputerSignal {

   public final static String NAME = "ReservationFree";

   public ReservationFree() {
   }

   public static Signal readSignal(DataInputStream reader) throws
       Exception {
      return new ReservationFree();
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
   }

   public String getName() {
      return NAME;
   }

}
```

C.16 Signal

```
package signals;

import java.io.DataOutputStream;
import java.io.IOException;

import legoAPI.Communicator.UnitId;

public interface Signal {

   // Should always have a field like this:
   // public static final String NAME = [Name of the signal, AKA
       classname in
```

```
    // lowercase]; (cannot inherit/enforce static final strings in
        subclasses)

    public void writeSignal(DataOutputStream writer) throws IOException;

    public boolean unitMatchSignal(UnitId recepient) throws IOException;

    // Used for accessing the static name from an instance
    public String getName();

    // Should also have a readSignal with an DataInputStream, that reads
        and instantiates a signal from the read values
    // (cannot enforce static methods in subclasses)
}
```

C.17 SwitchBoxSignal

```
package signals;

import java.io.IOException;

import legoAPI.Communicator.UnitId;

public abstract class SwitchBoxSignal implements Signal {

    public boolean unitMatchSignal(UnitId recepient) throws IOException {
        if (recepient == UnitId.TccW || recepient == UnitId.TccE)
            throw new IOException("Switchbox signal sent to train!");
        return true;
    }
}
```

C.18 SyncFail

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
```

```
public class SyncFail extends SwitchBoxSignal {

   public final static String NAME = "SyncFail";

   public SyncFail() {
   }

   public static Signal readSignal(DataInputStream reader) throws
        Exception {
      return new SyncFail();
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
   }

   public String getName() {
      return NAME;
   }

}
```

C.19 SyncReservation

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;

public class SyncReservation extends SwitchBoxSignal {

   public final static String NAME = "SyncReservation";

   public int trackToReserve;
   public UnitId train;
   public UnitId otherSwitchBox;
   public long otherSyncingSince;

   public SyncReservation(int trackToReserve, UnitId train, UnitId
        otherSwitchBox, long syncingSince) {
      this.trackToReserve = trackToReserve;
```

```
      this.train = train;
      this.otherSwitchBox = otherSwitchBox;
      this.otherSyncingSince = syncingSince;
   }

   public static Signal readSignal(DataInputStream reader) throws
        Exception {
      return new SyncReservation(reader.readInt(),
           Communicator.parseId(reader.readUTF()),
            Communicator.parseId(reader.readUTF()), reader.readLong());
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
      writer.writeInt(trackToReserve);
      writer.writeUTF(train.toString());
      writer.writeUTF(otherSwitchBox.toString());
      writer.writeLong(otherSyncingSince);
   }

   public String getName() {
      return NAME;
   }

}
```

C.20 SyncSuccesful

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class SyncSuccesful extends SwitchBoxSignal {

   public final static String NAME = "SyncSuccesful";

   public SyncSuccesful() {
   }

   public static Signal readSignal(DataInputStream reader) throws
        Exception {
      return new SyncSuccesful();
   }
```

```
   public void writeSignal(DataOutputStream writer) throws IOException {
   }

   public String getName() {
      return NAME;
   }

}
```

C.21 TrainControlComputerSignal

```
package signals;

import java.io.IOException;

import legoAPI.Communicator.UnitId;

public abstract class TrainControlComputerSignal implements Signal {

   public boolean unitMatchSignal(UnitId recepient) throws IOException {
      if (recepient != UnitId.TccW && recepient != UnitId.TccE)
         throw new IOException("Switchbox signal sent to train!");
      return true;
   }
}
```

C.22 TrainDetected

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class TrainDetected extends SwitchBoxSignal {

   public final static String NAME = "TrainDetected";

   public boolean sensorInStation;
```

```
  public TrainDetected(boolean sensorInStation) throws IOException {
    this.sensorInStation = sensorInStation;
  }

  public static Signal readSignal(DataInputStream reader) throws
      Exception {
    return new TrainDetected(reader.readBoolean());
  }

  public void writeSignal(DataOutputStream writer) throws IOException {
    writer.writeBoolean(sensorInStation);
  }

  public String getName() {
    return NAME;
  }

}
```

C.23 TrainNoLongerDetected

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class TrainNoLongerDetected extends SwitchBoxSignal {

  public final static String NAME = "TrainNoLongerDetected";

  public boolean sensorInStation;

  public TrainNoLongerDetected(boolean sensorInStation) throws
      IOException {
    this.sensorInStation = sensorInStation;
  }

  public static Signal readSignal(DataInputStream reader) throws
      Exception {
    return new TrainNoLongerDetected(reader.readBoolean());
  }
```

```
  public void writeSignal(DataOutputStream writer) throws IOException {
    writer.writeBoolean(sensorInStation);
  }

  public String getName() {
    return NAME;
  }

}
```

C.24 TryReserve

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;

public class TryReserve extends SwitchBoxSignal {

  public final static String NAME = "TryReserve";

  public int track;
  public UnitId train;
  public int currentPosition;

  public TryReserve(int track, UnitId train, int currentPosition)
      throws IOException {
    this.track = track;
    this.train = train;
    this.currentPosition = currentPosition;
  }

  public static Signal readSignal(DataInputStream reader) throws
      Exception {
    return new TryReserve(reader.readInt(),
        Communicator.parseId(reader.readUTF()), reader.readInt());
  }

  public void writeSignal(DataOutputStream writer) throws IOException {
    writer.writeInt(track);
```

```
      writer.writeUTF(train.toString());
      writer.writeInt(currentPosition);
   }

   public String getName() {
      return NAME;
   }

}
```

C.25 UpdatePosition

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;

public class UpdatePosition extends TrainControlComputerSignal {

   public final static String NAME = "UpdatePosition";

   public UpdatePosition() throws IOException {
   }

   public static Signal readSignal(DataInputStream reader) throws
        Exception {
      return new UpdatePosition();
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
   }

   public String getName() {
      return NAME;
   }

}
```

C.26 WaitForReservation

```
package signals;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class WaitForReservation extends TrainControlComputerSignal {

   public final static String NAME = "WaitForReservation";

   public WaitForReservation() {
   }

   static Signal readSignal(DataInputStream reader) throws Exception {
      return new WaitForReservation();
   }

   public void writeSignal(DataOutputStream writer) throws IOException {
   }

   public String getName() {
      return NAME;
   }

}
```

C.27 CommTest

```
package tests;

import java.io.IOException;
import java.util.Hashtable;

import controlSystem.ControlUnit;
import controlSystem.SwitchBox;
import legoAPI.Communicator;
import legoAPI.Communicator.UnitId;
import lejos.nxt.comm.Bluetooth;
import signals.TryReserve;
import util.DebugMessage;
```

```
public class CommTest {

   public static void main(String[] args) throws IOException,
       InterruptedException {
      DebugMessage.printStartUp = false;

      Hashtable<Integer, UnitId> tracksReservedSBNW = new
          Hashtable<Integer, UnitId>();
      tracksReservedSBNW.put(1, UnitId.TccW);

//    SwitchBox switchBox = new SwitchBox(UnitId.SbNW, UnitId.SbSW,
    UnitId.SbNE, null);
//    ControlUnit[] localControlUnits = { switchBox };
//    if (Communicator.WestNxtName.equals(Bluetooth.getFriendlyName())) {
//    } else {
//    }
//    Communicator communicator = new Communicator(localControlUnits);
//    communicator.instance.sendSignal(UnitId.SbNW, new TryReserve(2,
    UnitId.Tcc2, 0));
   }

}
```

C.28 DetectorTest

```
package tests;

import controlSystem.SwitchBox;
import legoAPI.Communicator.UnitId;
import legoAPI.Communicator;
import legoAPI.Detector;
import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.SensorPort;
import lejos.nxt.comm.Bluetooth;
import util.DebugMessage;

public class DetectorTest {

   public static void main(String[] args) throws InterruptedException {
      DebugMessage.printAPI = true;
      DebugMessage.printStartUp = true;
```

```
    boolean isWest =
        Communicator.WestNxtName.equals(Bluetooth.getFriendlyName());

    SwitchBox switchBoxNW = new SwitchBox();
    switchBoxNW.id = isWest ? UnitId.SbNW : UnitId.SbSE;
    new Detector(SensorPort.S1, switchBoxNW, true);
    new Detector(SensorPort.S2, switchBoxNW, false);

    SwitchBox switchBoxSW = new SwitchBox();
    switchBoxSW.id = isWest ? UnitId.SbSW : UnitId.SbNE;
    new Detector(SensorPort.S3, switchBoxSW, false);
    new Detector(SensorPort.S4, switchBoxSW, true);
    while (true) {
      Button.waitForAnyPress();
      LCD.clear();
    }

  }

}
```

C.29 DetectSensor

```
package tests;

import lejos.nxt.Button;
import lejos.nxt.LightSensor;
import lejos.nxt.SensorPort;
import lejos.nxt.UltrasonicSensor;

public class DetectSensor {

  public static void main(String[] args) {
    System.out.println(SensorPort.S2.getMode());
    System.out.println(SensorPort.S3.getType());
    System.out.println(SensorPort.S4.getSensorPin(4));

    UltrasonicSensor ultrasonicSensor = new
        UltrasonicSensor(SensorPort.S1);
    UltrasonicSensor ultrasonicSensor2 = new
        UltrasonicSensor(SensorPort.S2);
    LightSensor light1 = new LightSensor(SensorPort.S1, true);
    LightSensor light2 = new LightSensor(SensorPort.S2, true);
```

```
      //For ultrasonicSensor S1 and light S2
      System.out.println(ultrasonicSensor.getDistance()); // Just
          returns 255
      System.out.println(ultrasonicSensor2.getDistance()); // Just
          returns 255
      System.out.println(light1.getLightValue()); // Returns 0
      System.out.println(light2.getLightValue()); // Just returns 255

      Button.waitForAnyPress();
   }
}
```

C.30 Ev3onNxt

```
package tests;

import lejos.nxt.LightSensor;
import lejos.nxt.Motor;
import lejos.nxt.NXTRegulatedMotor;
import lejos.nxt.SensorPort;

public class Ev3onNxt {
   public static void main(String[] args) throws InterruptedException {
      System.out.println("Ev3 test:");

      NXTRegulatedMotor motorA = Motor.A;
      NXTRegulatedMotor motorB = Motor.B;

      motorA.forward();
      motorB.forward();

      //Not
      LightSensor colorSensor = new LightSensor(SensorPort.S1,true);

      while (true) {
         System.out.println(colorSensor.getLightValue());
         System.out.println(colorSensor.readValue());
      }
   }
}
```

C.31 FullAPITest

```
package tests;

public class FullAPITest {
   public static void main(String[] args) throws InterruptedException {
      System.out.println("API test: Press to start!");
      while (true) {
//       int buttonPressed = Button.waitForAnyPress();
//       if (Button.ID_ESCAPE == buttonPressed)
//          break;
//       System.out.println("Trains are a go!");
//       TrainMotor train1 = new TrainMotor(Motor.B, 2);
//       train1.start();
//
//       Detector trainDetector = new Detector(SensorPort.S1);
//       boolean noTrainSpotted = true;
//       while (noTrainSpotted) {
//          noTrainSpotted = !trainDetector.trainSpotted();
//       }
//       Switch switch1 = new Switch(Motor.A);
//
//       switch1.switchTrack();
//
//       Thread.sleep(3000);
//
//       train1.stopTrain();
      }
   }
}
```

C.32 HelloWorld

```
package tests;
import lejos.nxt.Button;

public class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello World");
    Button.waitForAnyPress();
  }
}
```

C.33 MotorTest

```
package tests;

import legoAPI.TrainMotor;
import lejos.nxt.Button;
import lejos.nxt.Motor;

public class MotorTest {

   public static void main(String[] args) {
      TrainMotor trainMotor = new TrainMotor(Motor.B, true);
      TrainMotor trainMotor2 = new TrainMotor(Motor.A, false);
      boolean started1 = false;
      boolean started2 = false;
      System.out.println("Press enter to start/stop main train1, any
          other for train2");
      while (true) {
         System.out.println(Motor.B.getTachoCount());
         if (Button.waitForAnyPress() == Button.ID_ENTER) {
            if (started1) {
               trainMotor.stop();
               started1 = false;
            } else {
               trainMotor.start();
               started1 = true;

            }
         } else {
            if (started2) {
               trainMotor2.stop();
               started2 = false;
            } else {
               trainMotor2.start();
               started2 = true;
            }
         }
      }
   }

}
```

C.34 SensorTest

```
package tests;

import lejos.nxt.LCD;
import lejos.nxt.LightSensor;
import lejos.nxt.SensorPort;

public class SensorTest {
   public static void main(String[] args) throws InterruptedException {
      LightSensor lightSensor1 = new LightSensor(SensorPort.S1, true);
      new LightSensor(SensorPort.S2, true);
      new LightSensor(SensorPort.S3, true);
      new LightSensor(SensorPort.S4, true);

      Thread.sleep(400);

      lightSensor1.calibrateLow();
      int baseValue1 = SensorPort.S1.readRawValue();
      int baseValue2 = SensorPort.S2.readRawValue();
      int baseValue3 = SensorPort.S3.readRawValue();
      int baseValue4 = SensorPort.S4.readRawValue();
      while (true) {
         LCD.drawString("Port1:", 0, 0);
         LCD.drawInt(SensorPort.S1.readRawValue(), 4, 0, 1);
         LCD.drawInt(baseValue1 - SensorPort.S1.readRawValue(), 4, 0, 2);
//       LCD.drawInt(lightSensor1.readValue(),4, 0, 3);

         LCD.drawString("Port2:", 0, 5);
         LCD.drawInt(SensorPort.S2.readRawValue(), 4, 0, 6);
         LCD.drawInt(baseValue2 - SensorPort.S2.readRawValue(), 4, 0, 7);

         LCD.drawString("Port3:", 10, 0);
         LCD.drawInt(SensorPort.S3.readRawValue(), 4, 10, 1);
         LCD.drawInt(baseValue3 - SensorPort.S3.readRawValue(), 4, 10,
             2);

         LCD.drawString("Port4:", 10, 5);
         LCD.drawInt(SensorPort.S4.readRawValue(), 4, 10, 6);
         LCD.drawInt(baseValue4 - SensorPort.S4.readRawValue(), 4, 10,
             7);


      }
   }
}
```

C.35 SimpleTrainRun

```
package tests;

import legoAPI.TrainMotor;
import lejos.nxt.Button;
import lejos.nxt.Motor;

public class SimpleTrainRun {
   public static void main(String[] args) throws InterruptedException {
      System.out.println("Press any button to start, escape to leave.");
      while (true) {
         int buttonPressed = Button.waitForAnyPress();
         if(Button.ID_ESCAPE == buttonPressed)
            break;
         System.out.println("Trains are a go!");
         TrainMotor train1 = new TrainMotor(Motor.B, true);
         train1.start();

         Thread.sleep(2000);

         train1.stop();
      }
   }
}
```

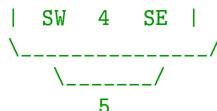
C.36 SwitchTest

```
package tests;

import legoAPI.Switch;
import lejos.nxt.Button;
import lejos.nxt.Motor;

public class SwitchTest {
   public static void main(String[] args) throws InterruptedException {
      Switch switch1 = new Switch(Motor.A, 0);
      int position = 1;
      System.out.println("Press to switch");
      while(true){
         Button.waitForAnyPress();
         switch1.switchTrack(position);
         if(position == 0)
```

```
            position = 1;
        else
            position = 0;
      }
   }
}
```

C.37 SystemTest

```
package tests;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.LinkedList;

import controlSystem.ControlUnit;
import controlSystem.SwitchBox;
import controlSystem.TrainControlComputer;
import legoAPI.Communicator;
import legoAPI.Detector;
import legoAPI.Switch;
import legoAPI.Communicator.UnitId;
import legoAPI.TrainMotor;
import lejos.nxt.Button;
import lejos.nxt.Motor;
import lejos.nxt.SensorPort;
import lejos.nxt.comm.Bluetooth;
import signals.TrainDetected;
import signals.TrainNoLongerDetected;
import util.DebugMessage;

public class SystemTest {

   private static TrainControlComputer train;

   // @formatter:off
   /* Track shape and naming is as such:
        1
        _______
   ___/_______\___
 /  NW   2   NE  \
 |               |
0|               | 3
```



```
 |  SW   4   SE  |
 \_______________/
    \_______/
        5
Points/SwitchBoxes are designated by cardinal direction (North West =
    NW). Stations are referred to as outer (1,5) and inner (2,4).
Train 1 starts at 1 and train 2 at 4. Switches connect to the inner
    tracks (2, 4)
*/
  // @formatter:on

  public static void main(String[] args) throws IOException,
      InterruptedException {
    // Debug.disabled = true;
    // Debug.printCommunicator = true;
    // Debug.printSwitchBox = true;
    DebugMessage.printTrainControlComputer = true;
    DebugMessage.attachControlUnitId = true;
    // Debug.allEnabled = true;

    Hashtable<UnitId, ControlUnit> localControlUnits = new
        Hashtable<UnitId, ControlUnit>();

    if (Communicator.WestNxtName.equals(Bluetooth.getFriendlyName())) {

      // Initializing SBNW
      Hashtable<Integer, UnitId> tracksReservedSBNW = new
          Hashtable<Integer, UnitId>();
      tracksReservedSBNW.put(1, UnitId.TccW);

      ArrayList<Integer> tracksNW = new ArrayList<Integer>();
      tracksNW.add(0);
      tracksNW.add(1);
      tracksNW.add(2);

      Switch switchNW = new Switch(Motor.A, 2);

      SwitchBox switchBoxNW = new SwitchBox(UnitId.SbNW, tracksNW,
          tracksReservedSBNW, new int[] { 0, 2 }, UnitId.SbNE,
           UnitId.SbSE, switchNW);

      // Detector detectorNwEntrance = new Detector(SensorPort.S2,
          switchBoxNW,
      // false);
      // Detector detectorNwStation = new Detector(SensorPort.S1,
          switchBoxNW,
      // true);
```

```
        // Initializing SBSW
        Hashtable<Integer, UnitId> tracksReservedSBSW = new
            Hashtable<Integer, UnitId>();
        tracksReservedSBSW.put(0, null);
        tracksReservedSBSW.put(4, UnitId.TccE);
        tracksReservedSBSW.put(5, null);

        ArrayList<Integer> tracksNE = new ArrayList<Integer>();
        tracksNE.add(0);
        tracksNE.add(4);
        tracksNE.add(5);

        Switch switchSW = new Switch(Motor.C, 4);

        SwitchBox switchBoxSW = new SwitchBox(UnitId.SbSW, tracksNE,
            tracksReservedSBSW, new int[] { 0, 4 }, UnitId.SbSE,
             UnitId.SbNW, switchSW);

        // Detector detectorSWentrance = new Detector(SensorPort.S3,
            switchBoxSW,
        // false);
        // Detector detectorSWstation = new Detector(SensorPort.S4,
            switchBoxSW,
        // true);

        // Initializing train1
        int[] route = { 3, 5, 0, 1 };
        UnitId[] routeSwitchBoxes = { UnitId.SbNE, UnitId.SbSE,
            UnitId.SbSW, UnitId.SbNW };

        TrainMotor trainMotor = new TrainMotor(Motor.B, true);
        train = new TrainControlComputer(UnitId.TccW, 1, trainMotor,
            route, routeSwitchBoxes, false);

        ControlUnit[] units = { train, switchBoxNW, switchBoxSW };

        Communicator communicator = new Communicator(units);
        train.start();
    } else {

        // Initializing SBNE
        Hashtable<Integer, UnitId> tracksReservedSBNE = new
            Hashtable<Integer, UnitId>();
        tracksReservedSBNE.put(1, UnitId.TccW);

        ArrayList<Integer> tracksNE = new ArrayList<Integer>();
```

```
tracksNE.add(1);
tracksNE.add(2);
tracksNE.add(3);

Switch switchNE = new Switch(Motor.A, 2);

SwitchBox switchBoxNE = new SwitchBox(UnitId.SbNE, tracksNE,
    tracksReservedSBNE, new int[] { 3, 2 }, UnitId.SbNW,
     UnitId.SbSE, switchNE);

// Detector detectorNeEntrance = new Detector(SensorPort.S3,
    switchBoxNE,
// false);
// Detector detectorNeStation = new Detector(SensorPort.S4,
    switchBoxNE,
// true);

// Initializing SBSE
Hashtable<Integer, UnitId> tracksReservedSBSE = new
    Hashtable<Integer, UnitId>();
tracksReservedSBSE.put(4, UnitId.TccE);

ArrayList<Integer> tracksSE = new ArrayList<Integer>();
tracksSE.add(3);
tracksSE.add(4);
tracksSE.add(5);

Switch switchSE = new Switch(Motor.A, 2);

SwitchBox switchBoxSE = new SwitchBox(UnitId.SbSE, tracksSE,
    tracksReservedSBSE, new int[] { 3, 4 }, UnitId.SbSW,
     UnitId.SbNE, switchSE);

// Detector detectorSeEntrance = new Detector(SensorPort.S2,
    switchBoxSE,
// false);
// Detector detectorSeStation = new Detector(SensorPort.S1,
    switchBoxSE,
// true);

// Initializing train2
int[] route = { 3, 2, 0, 4 };
UnitId[] routeSwitchBoxes2 = { UnitId.SbSE, UnitId.SbNE,
    UnitId.SbNW, UnitId.SbSW };

TrainMotor trainMotor = new TrainMotor(Motor.C, false);
```

```
    train = new TrainControlComputer(UnitId.TccE, 4, trainMotor,
        route, routeSwitchBoxes2, false);

    ControlUnit[] units = { train, switchBoxSE, switchBoxNE };
    Communicator communicator = new Communicator(units);
    train.start();
  }
  do {
    Button.waitForAnyPress();
  } while (!train.trainMotor.running);

  UnitId nextSb = train.routeSwitchBoxes[train.routeIterator];
  Communicator.instance.sendSignal(nextSb, new TrainDetected(true));
  Communicator.instance.sendSignal(nextSb, new TrainDetected(false));
  Communicator.instance.sendSignal(nextSb, new
      TrainNoLongerDetected(true));
  Communicator.instance.sendSignal(nextSb, new
      TrainNoLongerDetected(false));

  do {
    Button.waitForAnyPress();
  } while (!train.trainMotor.running);

  nextSb = train.routeSwitchBoxes[train.routeIterator];

  Communicator.instance.sendSignal(nextSb, (new
      TrainDetected(false)));

  do {
    Button.waitForAnyPress();
  } while (train.trainMotor.running);
  Communicator.instance.sendSignal(nextSb, new TrainDetected(true));
  Communicator.instance.sendSignal(nextSb, new
      TrainNoLongerDetected(false));
  Communicator.instance.sendSignal(nextSb, new
      TrainNoLongerDetected(true));

  do {
    Button.waitForAnyPress();
  } while (!train.trainMotor.running);

  nextSb = train.routeSwitchBoxes[train.routeIterator];
  Communicator.instance.sendSignal(nextSb, new TrainDetected(true));
  Communicator.instance.sendSignal(nextSb, new TrainDetected(false));
  Communicator.instance.sendSignal(nextSb, new
      TrainNoLongerDetected(true));
```

```
        Communicator.instance.sendSignal(nextSb, new
            TrainNoLongerDetected(false));

    }

}
```

C.38 TestMultipleConnection

```
package tests;

import java.io.DataInputStream;
import java.io.IOException;

import lejos.nxt.LCD;
import lejos.nxt.comm.Bluetooth;
import lejos.nxt.comm.NXTConnection;

public class TestMultipleConnection {

    //Nope
    public static void main(String[] args) throws IOException {
        NXTConnection connection = Bluetooth.waitForConnection();

        System.out.println("wait 2");
        NXTConnection connection2 = Bluetooth.waitForConnection();

        System.out.println("wait 3");
        NXTConnection connection3 = Bluetooth.waitForConnection();

        System.out.println("wait 4");
        NXTConnection connection4 = Bluetooth.waitForConnection();

        System.out.println("wait 5");
        NXTConnection connection5 = Bluetooth.waitForConnection();

        System.out.println("wait 6");
        NXTConnection connection6 = Bluetooth.waitForConnection();

        System.out.println("dis1");
        DataInputStream dis = connection.openDataInputStream();

        System.out.println("dis2");
        DataInputStream dis2 = connection2.openDataInputStream();
```

```
      System.out.println("dis3");
      DataInputStream dis3 = connection3.openDataInputStream();

      System.out.println("dis4");
      DataInputStream dis4 = connection4.openDataInputStream();

      System.out.println("di s5");
      DataInputStream dis5 = connection5.openDataInputStream();

      System.out.println("dis6");
      DataInputStream dis6 = connection6.openDataInputStream();

      LCD.drawInt(6, 1, 1);
      boolean readBoolean = dis6.readBoolean();

      LCD.drawString(readBoolean + " " + 1, 1, 1);
      boolean readBoolean2 = dis.readBoolean();
      LCD.drawString(readBoolean2 + " " + 3, 1, 1);
      int readInt = dis3.readInt();
      LCD.drawString(readInt + " " + 3, 1, 1);
      byte readByte = dis6.readByte();
      LCD.drawString(readByte + " " + 2, 1, 1);
      int readInt2 = dis2.readInt();
      LCD.drawString(readInt2 + " " + 6, 1, 1);
      int readInt3 = dis6.readInt();
      LCD.drawString(readInt3 + " end", 1, 1);

   }

}
```

C.39 TrainSpeed

```
package tests;

import legoAPI.TrainMotor;
import lejos.nxt.Button;
import lejos.nxt.Motor;

public class TrainSpeed {
   public static void main(String[] args) throws InterruptedException {
      System.out.println("Press any button to start, escape to leave.");
      while (true) {
```

```
        int buttonPressed = Button.waitForAnyPress();
        if (Button.ID_ESCAPE == buttonPressed)
          break;
        else if (buttonPressed == Button.ID_RIGHT) {
          Motor.B.rotate(50);
        } else if (buttonPressed == Button.ID_LEFT) {
          Motor.B.rotate(-50);
        } else if (buttonPressed == Button.ID_ENTER) {
          Motor.B.rotateTo(0);
        }
      }
    }
}
```

C.40 UltraSonic

```
package tests;

import java.awt.peer.ButtonPeer;

import legoAPI.Detector;
import lejos.nxt.Button;
import lejos.nxt.LCD;
import lejos.nxt.SensorPort;
import lejos.nxt.UltrasonicSensor;

public class UltraSonic {

  //Too unreliable, value often shifting
  public static void main(String[] args) throws InterruptedException {
    UltrasonicSensor ultra = new UltrasonicSensor(SensorPort.S4);
    ultra.setMode(UltrasonicSensor.MODE_CONTINUOUS);

    Thread.sleep(250);

    int baseValue = ultra.getDistance();
    while (true) {
      LCD.drawInt(ultra.getDistance(), 4, 0, 0);
      LCD.drawInt((int) (ultra.getRange()), 4, 0, 2);

      LCD.drawInt((int) (ultra.getDistance() -baseValue), 4, 0, 4);

//      Doesn't work
//      LCD.drawInt(SensorPort.S4.readRawValue(), 4, 0, 3);
```

```
//      LCD.drawInt(SensorPort.S4.readValue(), 4, 0, 4);
//
//      LCD.drawInt(baseValue - SensorPort.S4.readRawValue(), 4, 0, 6);
      }
   }
}
```

C.41 DebugMessage

```
package util;

import javax.microedition.lcdui.Display;

import controlSystem.ControlUnit;
import controlSystem.SwitchBox;
import controlSystem.TrainControlComputer;
import legoAPI.Communicator;
import lejos.nxt.Button;
import lejos.nxt.LCD;

public class DebugMessage {

   public static boolean disabled;
   public static boolean allEnabled;

   public static boolean printStartUp;

   public static boolean printCommunicator;
   public static boolean printSwitchBox;
   public static boolean printTrainControlComputer;

   public static boolean attachControlUnitId;
   public static boolean attachObjectAbbreviation;

   public static boolean printAPI;

   public static void print(String message, Object printingClass,
       boolean override, boolean isStartUp) {
      if (disabled)
         return;

      if (isStartUp && !printStartUp && !allEnabled)
         return;
```

```
    String prefix = "";
    if (printingClass instanceof Communicator) {
      // isStatus overrrides / has higher priority than the rest
      if (!printCommunicator && !override && !allEnabled)
        return;
      prefix = "C";
    } else if (printingClass instanceof SwitchBox) {
      if (!printSwitchBox && !override && !allEnabled)
        return;
      prefix = "SB";
    } else if (printingClass instanceof TrainControlComputer) {
      if (!printTrainControlComputer && !override && !allEnabled)
        return;
      prefix = "TCC";
    } else {
      if (!printAPI && !override && !allEnabled)
        return;
      prefix = "API";
    }

    if (attachControlUnitId && (printingClass instanceof ControlUnit))
        {
      String id = ((ControlUnit) printingClass).id.toString();
      if (printingClass instanceof SwitchBox)
        id = id.substring(id.length() - 2);
      else
        id = id.substring(id.length() - 4);
      message = id + " " + message;

    } else if (attachObjectAbbreviation)
      message = prefix + " " + message;

    if (message.length() == (Display.SCREEN_CHAR_WIDTH))
      System.out.print(message);
    else
      System.out.println(message);
  }

  public static void print(String message, Object printingFrom, boolean
       isStartUp) {
    print(message, printingFrom, false, isStartUp);
  }

  public static void print(String message, Object printingFrom) {
    print(message, printingFrom, false, false);
  }
```

```
  public static void showCatchMessage(Exception e, String className,
       String exceptionType) {
     disabled = true;
     if (exceptionType != null)
        System.out.println(className + " threw " + exceptionType + "!");
     else
        System.out.println(className + " threw exception!");
     if (e.getMessage() != null)
        System.out.println(e.getMessage());
     System.out.println("Press for trace");
     Button.waitForAnyPress();
     LCD.clear();

     e.printStackTrace();

     Button.waitForAnyPress();
  }

  public static void showCatchMessage(Exception e, String className) {
     showCatchMessage(e, className, null);
  }

}
```

Bibliography

[1] Anne E. Haxthausen, Jan Peleska, *Formal development and verification of a distributed railway control system*, IEEE, 2000

[2] Andrew Hunt, David Thomas, *The Pragmatic Programmer: From Journeyman to Master*, The Pragmatic Bookshelf, 2nd edition, 1999.

[3] Dave Thomas, Andy Hunt, *Orthogonality and the DRY Principle*, interview, 2003,
`http://www.artima.com/intv/dry.html`