Convoy Cruise Control

Roberto Casadei

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy
roberto.casadei12@studio.unibo.it

Abstract. This document represents a snapshot of the development process of the Convoy Cruise Control.

Keywords: Convoy Cruise Control, control system, development process, requirements, analysis, project, implementation, testing.

1 Introduction

1.1 Vision

In case of environmental disasters, the technology can help.
In particular, the need of tools and goods is prominent.
A convoy of vehicles can satisfy this need. Moreover, it would be great if we were able to reckon on a control system in order to ease the management of the convoy.

1.2 Goals

We would like to develop the software support which will be installed onto the vehicles and on the chief vehicle. All this software forms what we call the *Convoy Cruise Control* system.
As it is a software project, we will apply the principles of software engineering in order to reduce the time and cost of the project.
Our aim is to apply a software process that is repeatable, defined and managed.

2 Requirements

Look at `www.cruisecontrolsystem.it/requirements.pdf`

3 Requirement analysis

3.1 Glossary

– *CONVOY CRUISE CONTROL*: It is a control system that allows to manage the convoy and the convoy's vehicles. It receives the information

from the vehicles about their status and speed, and send commands to them, allowing to monitor and control the behavior of the overall convoy and of the single vehicles.

- *CONVOY*: It is a line of vehicles with a chief at the head.

- *VEHICLE*: It is a means of conveyance which is able to move autonomously. It must keep a precise position in the line and move at the *speed* which has been set by the chief. It includes a dashboard with a display that shows the current speed (in Kms/sec and Kms/h) and the number of kilometers covered by the vehicle.

- *CHIEF VEHICLE*: It is the vehicle on which the chief (the person responsible for the convoy) stays. It has a dashboard which is composed of a display that shows the status of all the convoy's vehicles and of a display like that of the other vehicles. Moreover, the chief vehicles has a control panel that allows to set the convoy speed and to make it start and stop. Through the dashboard and the control panel, the chief can interface himself to the Cruise Control System.

- *DASHBOARD*: It is a panel that contains displays which allow to see multiple information.

- *DISPLAY*: It is something that can render some textual content.

- *CONTROL PANEL*: It is a panel which contains buttons and input fields. It allows to send commands to the convoy's vehicles upon a certain communication infrastructure.

- *KILOMETER COUNTER*: It's needed by the vehicles in order to keep the count of the kilometers covered.

3.2 Use Cases

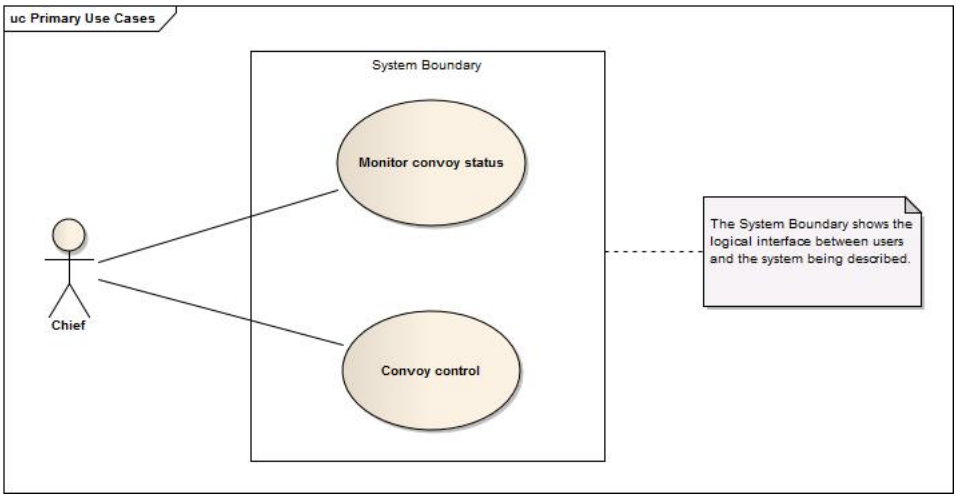

Fig. 1. Primary use cases

3.3 Scenarios

The following scenarios express the main things we want the system to do.

Table 1. Scenario 1: Monitor Convoy Status

Field	**Description**
ID(Nome)	UC1 - Monitor Convoy Status
Description	Here the chief monitors the status of all the vehicles of the convoy.
Actors	Chief.
Main scenario	The chief will look at the display in the chief vehicle's dashboard and will see one flag for each vehicle in the convoy, indicating if the vehicle is able or not to run. These flags are set by the Convoy Cruise Control according to the information gathered from the vehicles.
Preconditions	A convoy must exist and a communication infrastructure must be up and running.
Postconditions	If one vehicle is able to run, its flag must be green, red otherwise.

Table 2. Scenario 2: Convoy Control

Field	**Description**
ID(Nome)	UC2 - Convoy Control
Description	Here the chief controls the convoy's vehicles through a dashboard on the chief vehicle at the head of the convoy.
Actors	Chief.
Main scenario	The chief wants to make the convoy start. In order to do so, he will *set the speed* of the convoy (i.e. of all the convoy's vehicles) in *Km/s* (and possibly as an expression) and then will send a *start* command. When the convoy has started, the CCC applies the control logic based on the feedback received from the vehicles.
Secondary scenarios	Sometimes, when the convoy is running, the chief may decide to stop it. In order to do so, he will send to the convoy a *stop* command.
Preconditions	A convoy must exist and a communication infrastructure must be up and running. In order to send a *start* command, the convoy speed must be set.
Postconditions	All the convoy's vehicles must execute the command sent by the chief, if possible, otherwise the system must cope with the problem.

3.4 Domain Model

3.4.1 The Convoy Cruise Control

The CCC encapsulates the convoy control logic.
The following interfaces summary the operations that need to be performed by the Convoy Cruise Control.

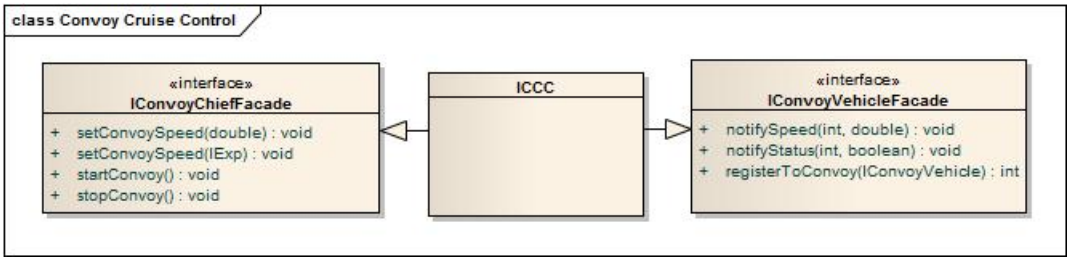

Fig. 2. Domain Model - The CCC system

Note how the notion of expression as a speed representation (*IExp*) is present only at the boundary that forms the chief interface to the CCC.

3.4.2 The Convoy

It's a line of vehicles.

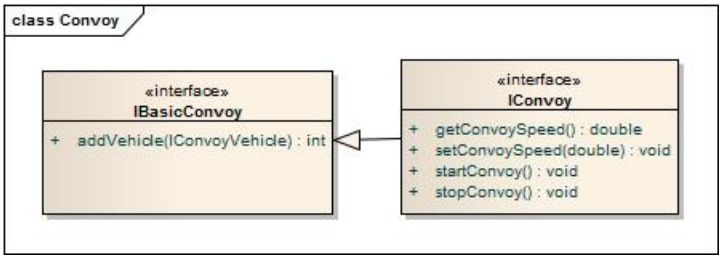

Fig. 3. Domain Model - The Convoy

For a semantic description of this entity, look at the *TestConvoy* test case.

3.4.3 The Convoy Vehicle

The vehicle is an *active* entity.

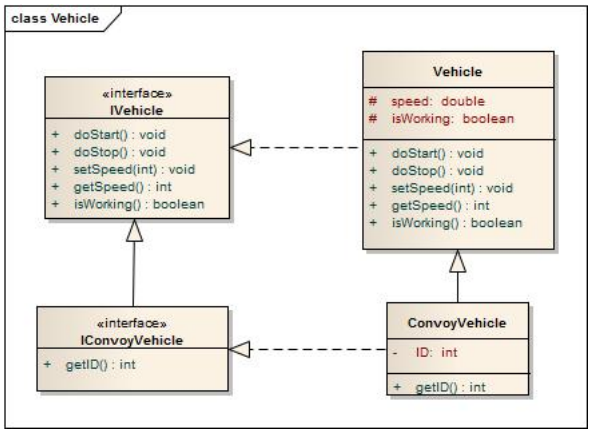

Fig. 4. Domain Model - Vehicles

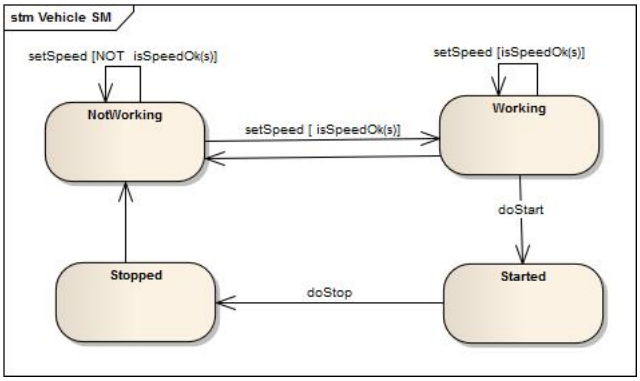

Fig. 5. Domain Model - Behavioral - Vehicles

For a semantic description of this entity, look at the *TestVehicle* test case.

3.4.4 The Chief Vehicle

The chief vehicle is just a vehicle.
In addition, it is equipped with a dashboard that allows it to interface the chief with the CCC.

3.4.5 Dashboard, Display, Command Panel, Kilometer Counter, Expression Parser and Evaluator

The Display and Kilometer Counter entities has been already modelled in a previous project (see *The Kilometer Counter*).
The same argument applies for the dashboard (which is a frame that is able to host multiple displays and buttons), the command panel, the I/O communication infrastructure, the expression parser and evaluator.

We'll leverage on these well-proved solutions in order to strike down the cost and time of the project.

4 Problem analysis

4.1 Main Issues

4.1.1 Convoy construction

A *Convoy* is composed of at least two *Vehicle*, with the first one being the chief. A subsequent issue regards *how* a *Vehicle* becomes part of the *Convoy*. It may be considered a sort of registration mechanism handled through the *CCC* system.

4.1.2 Vehicle Identification

The *Vehicle*s *and* their position need to be identified.
For example, as the speed/status information can be sent to the *CCC* from different *Vehicle*s, a way for discriminating between them is needed.

4.1.3 Convoy Management

Some issues need to be considered when it has to do with the management of the *Convoy*.

- When a *Convoy* has to be started, the vehicles must start leaving a security distance of *DD* m.
- When a *Convoy* has to be started or stopped, the *CCC* and/or the *Vehicle*s must guarantee that no crash happens.

It follows that a sort of *reliability* is needed.
In particular:

- A *Vehicle* cannot start if the next *Vehicle* hasn't started. It also has to start in order to keep the agreed security distance.
- A *Vehicle* cannot stop if the previous *Vehicle* hasn't stopped.

Some possible solutions *might* include:

- Sending the *Start* command from the first *Vehicle* to the last one, spaced with a delay that allows to guarantee the security distance;
- Sending the *Stop* command from the last *Vehicle* to the first one in order to avoid pile-up crashes;
- Delegating the security distance guarantee to *Vehicle*s;

but it turns out that, in the first two cases, we must be sure that the command will be executed.
It follows that, unless abnormal situations aren't handled by the vehicles' operator, a cooperation of the *Vehicle*s with the *CCC* is necessary.

4.2 Distribution

The *Convoy Cruise Control* is intrinsically a distributed system. Consequently, the notion of *CCC* is distributed along the system: the *Vehicle*s and the *Chief Vehicle* has to be aware of it, at two distinct levels (see *Fig. 2*).

As a consequence, a communication infrastructure must be up and running.

We can note also how the idea of a *Convoy* as a simple aggregation of *Vehicle*s is not much significant. It gets meaning when it becomes intimately related with the *CCC*.

4.3 Communication between the *Vehicle*s and the *CCC*

The *Vehicle*s and the *Convoy Cruise Control* need to communicate. The *CCC* sends commands (set speed, start, stop) to the *Vehicle*s, and the *Vehicle*s notify their status and speed, periodically.
The nature of the messages is fundamentally *asynchronous*.

In addition, as we are in a distributed context, we must consider a series of issues:

- We're not sure that a message will be correctly delivered to the destination;
- If we don't resort to an acknowledge/reply mechanism, we're also not sure if the message will be correctly executed;
- Timing issues raise;
- Security issues raise;

4.4 Logical Architecture

4.4.1 Interaction View

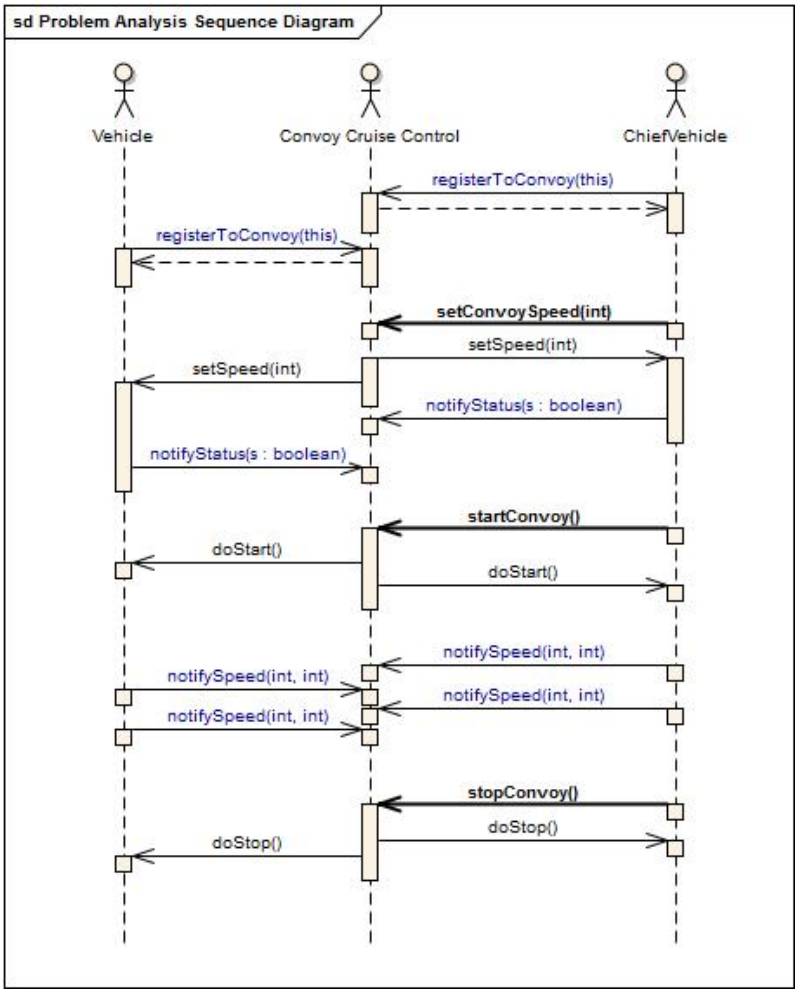

Fig. 6. Logical Architecture - Interaction View

4.4.2 Structural View

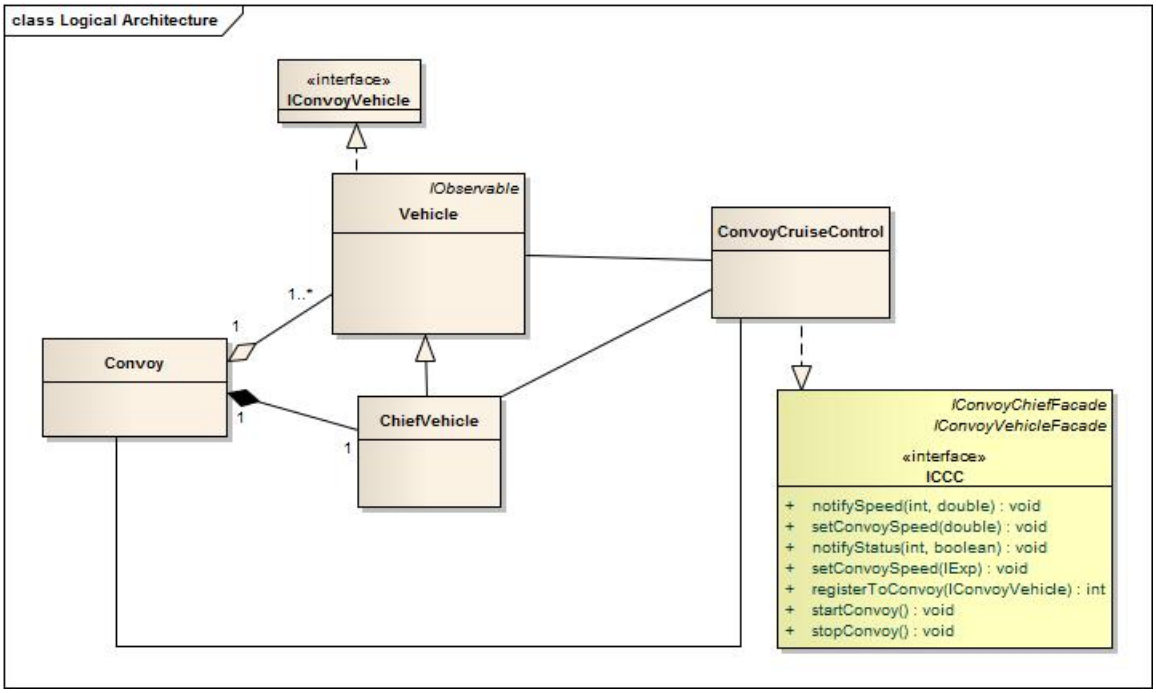

Fig. 7. Logical Architecture - Structural View

5 Project

First of all, let's underline the main project choices and characteristics of the solution.

- The *Convoy* collapsed into the notion of *Convoy Cruise Control* as a system. In fact, the *Vehicle* and the *Chief Vehicle* are more concerned with the *CCC* than with the *Convoy* (actually they're not concerned with the *Convoy* at all once that the registration to the *Convoy* is handled by the *CCC*), which is hidden by the former.

- The system-wide information is placed in a centralized knowledge (data) base.

- The communication between the *CCC* and the *Vehicle*s can be seen as an exchange of messages and commands. So, the solution (around the *IActivity* interface) is inspired to the *Command* and *Interpreter* patterns.

- The *Vehicle*s has been made *Observable* entities, so that we can attach *Observer*s (such as *Display*s).

- The *CCC* can be seen as a sort of *Observer*, so that it has to be notified for the speed and status change by the *Vehicle*s.

- The sender on the client (vehicle) side and the sender/receiver on the server (chief) side are based on the *Proxy* pattern.

5.1 Structure

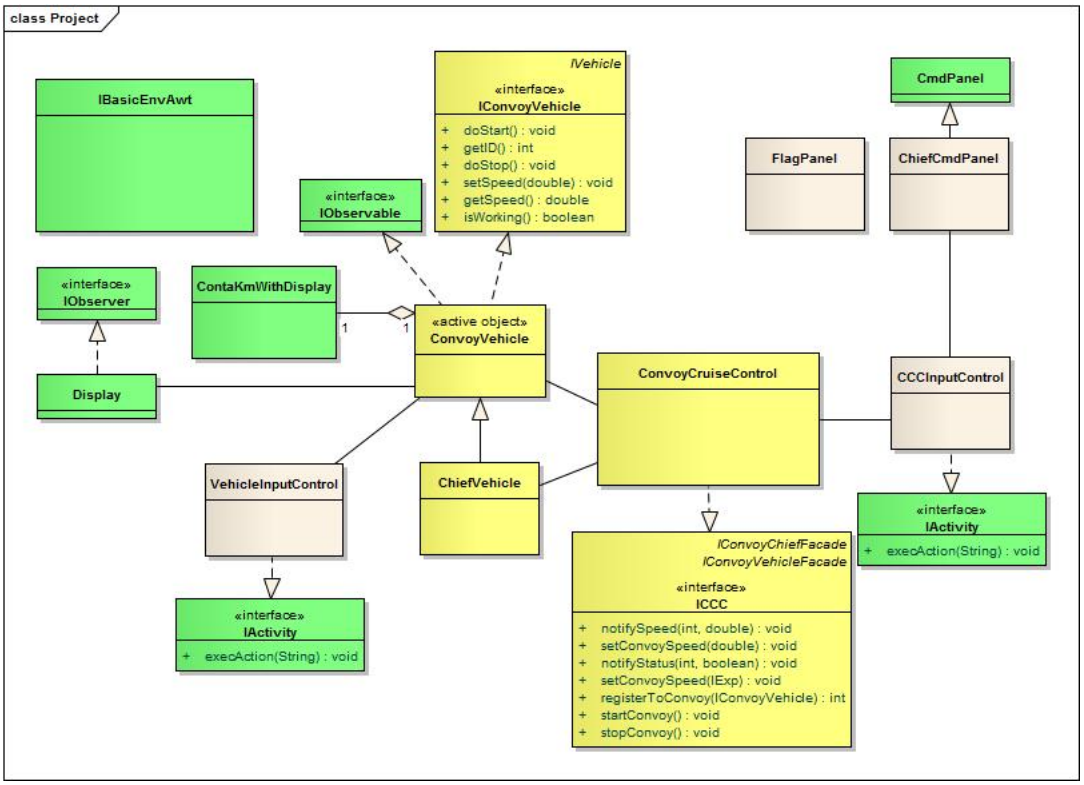

Fig. 8. Project - Structural View

5.2 Interaction

5.2.1 Vehicle to CCC

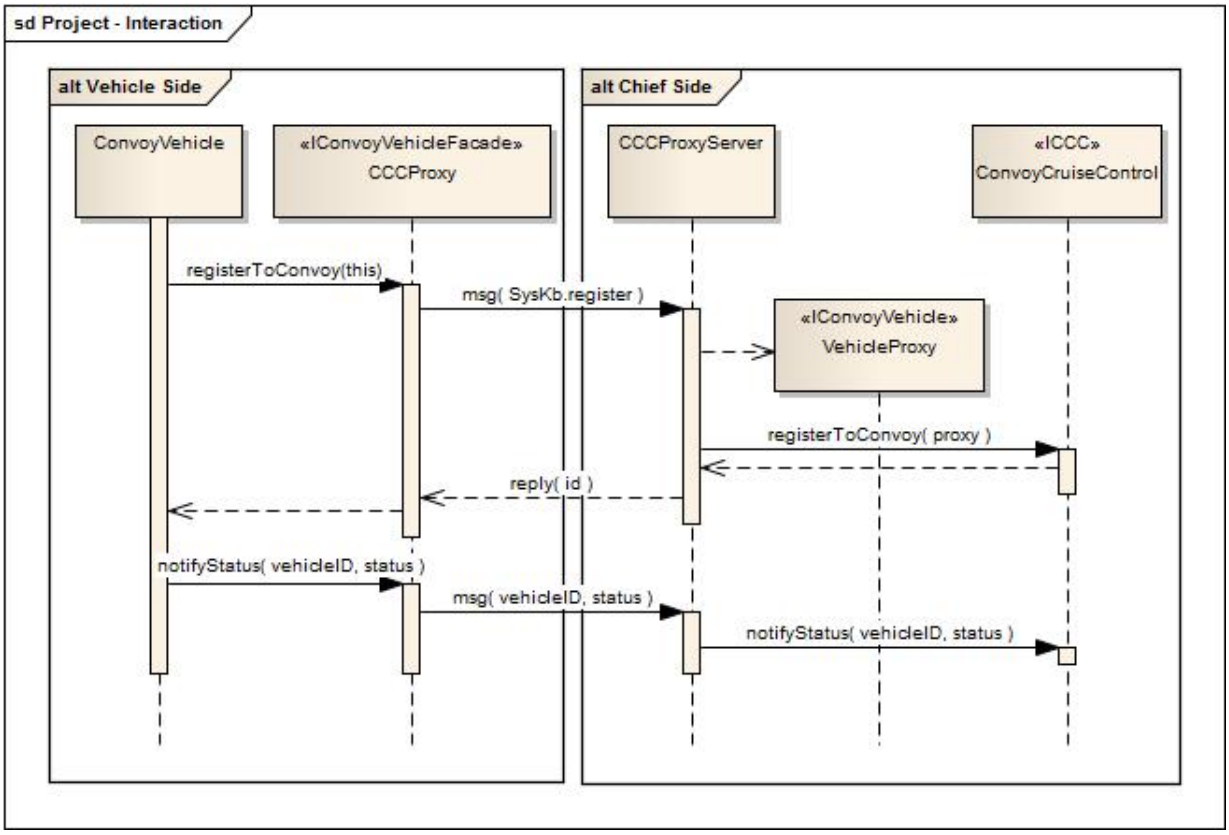

Fig. 9. Project - Interaction View (Vehicle to CCC)

5.2.2 CCC to Vehicles

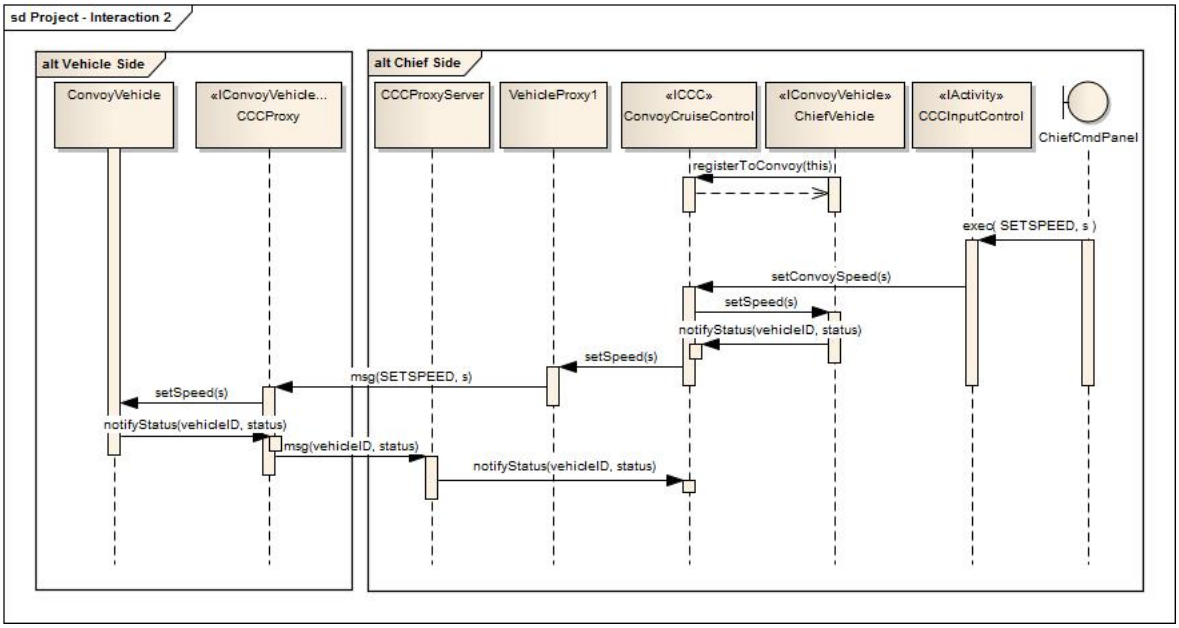

Fig. 10. Project - Interaction View (CCC to Vehicles)

6 Implementation

The project is implemented in the Java language as a set of Eclipse plugins (OSGI bundles).

6.1 Component System

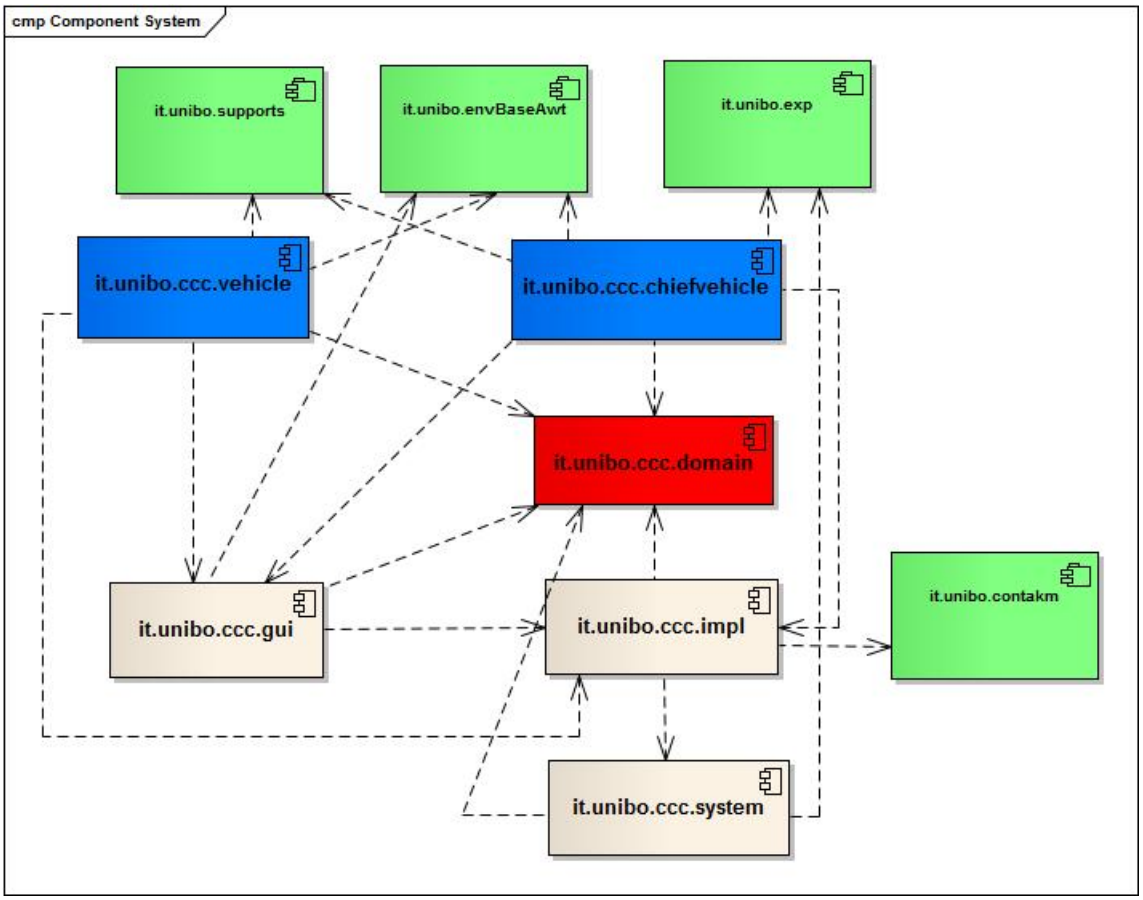

Fig. 11. Component Dependencies Structure

7 Deployment

So that the project has been developed as a set of components represented by Eclipse plugins (OSGI bundles), it can be deployed as a set of JARs to be run on an OSGI container.

Listing 1.1. OSGI bundle installation

```
> install file:plugins/it.unibo.basicInterfaces_1.4.4.jar
> install file:plugins/it.unibo.system_1.0.0.jar
> install file:plugins/it.unibo.supports_1.0.0.jar
> install file:plugins/it.unibo.envBaseAwt_1.4.4.jar

> install file:plugins/it.unibo.contaKm.displayAwt_1.0.0.jar
> install file:plugins/it.unibo.contaKm.domain_1.0.0.jar

> install file:plugins/it.unibo.exp.interfaces_1.0.0.jar
> install file:plugins/it.unibo.exp.interpreter_1.0.0.jar
> install file:plugins/it.unibo.exp.lexer_1.0.0.jar
> install file:plugins/it.unibo.exp.parser_1.0.0.jar
> install file:plugins/it.unibo.exp.token_1.0.0.jar
> install file:plugins/it.unibo.exp.expr_1.0.0.jar
> install file:plugins/it.unibo.exp.input_1.0.0.jar

> install file:plugins/it.unibo.ccc.domain_1.0.0.jar
> install file:plugins/it.unibo.ccc.system_1.0.0.jar
> install file:plugins/it.unibo.ccc.gui_1.0.0.jar
> install file:plugins/it.unibo.ccc.impl_1.0.0.jar
> install file:plugins/it.unibo.ccc.vehicle_1.0.0.jar
> install file:plugins/it.unibo.ccc.chiefvehicle_1.0.0.jar
```

Next, the bundle related to *it.unibo.ccc.chiefvehicle* (chief side) and *it.unibo.ccc.vehicle* (vehicle side) can be started.

7.1 Full Dependency Graph

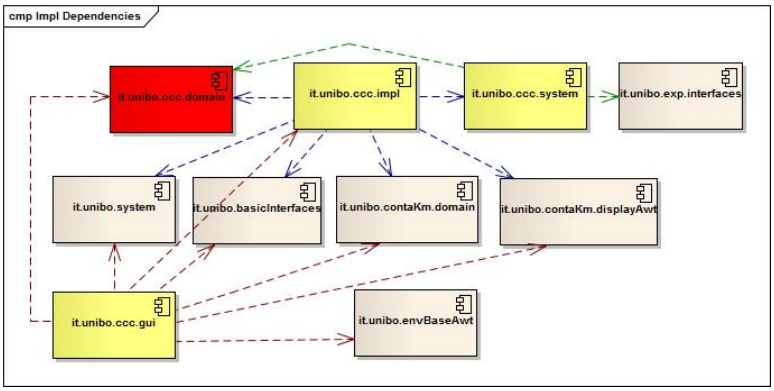

Fig. 12. Full Dependency Graph 1

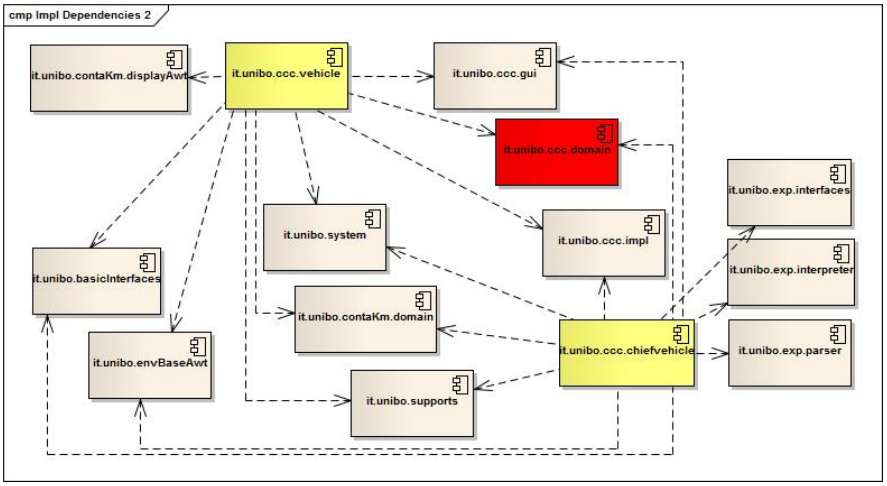

Fig. 13. Full Dependency Graph 2

8 Testing

The bundle *it.unibo.ccc.tests* contains some unit tests born from the test plans for the domain entities.

A simulation can be performed by starting the bundles *it.unibo.ccc.chiefvehicle* and *it.unibo.ccc.vehicle*s on different machines. This can tests the functional aspects of the *Convoy Cruise Control system*.

Later we'll perform an Acceptance Testing session...