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

1.2 Goals

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

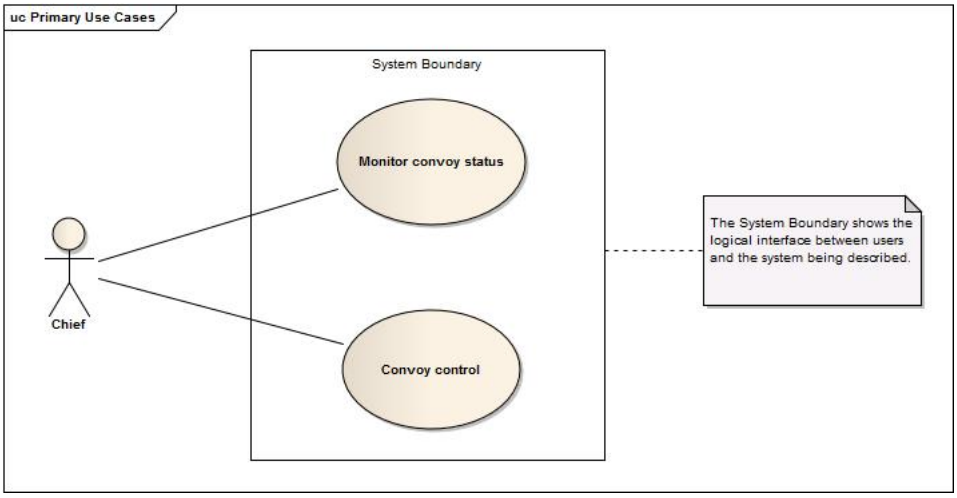

Fig. 1. Primary use cases

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) and then will send a *start* command.
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.

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.

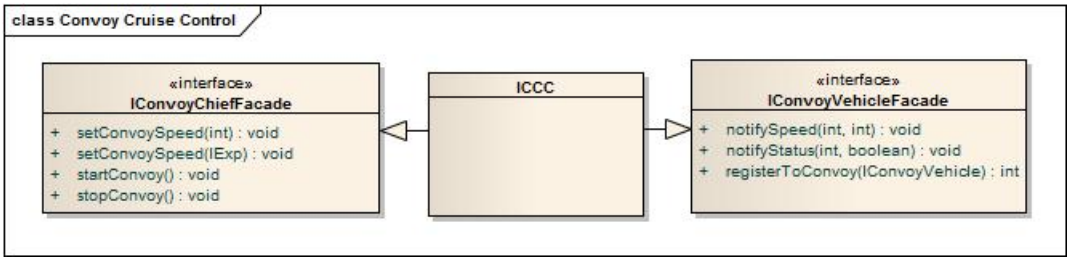

Fig. 2. Domain Model - The CCC system

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.

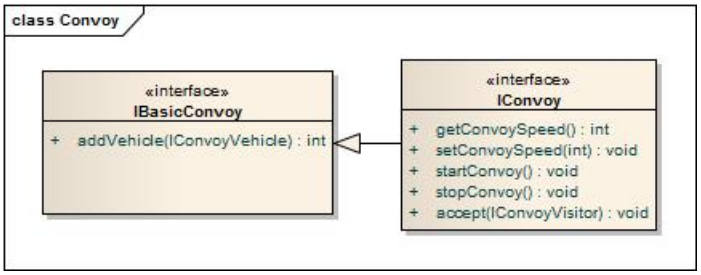

Fig. 3. Domain Model - The Convoy

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.

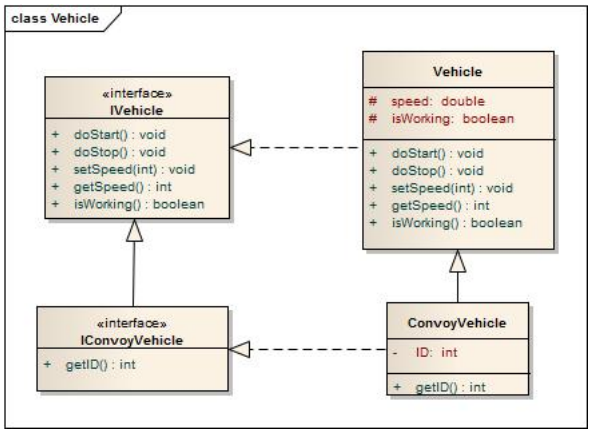

Fig. 4. Domain Model - Vehicles

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 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 *start* commands have to be sent from the first *Vehicle* to the last one; moreover, they have to be sent spaced with a delay that allows to guarantee a security distance of *DD* m.
- When a *Convoy* has to be stopped, the *stop* commands have to be sent from the last *Vehicle* to the first one; otherwise, a pile-up crash is possible.

Moreover, a sort of *reliability* is needed.
For example, a *Vehicle* cannot be sent a *stop* command if the previous *Vehicle* in the line hasn't correctly executed the *stop* command, otherwise a crash may occur.

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*).

4.3 Logical Architecture

4.3.1 Interaction View

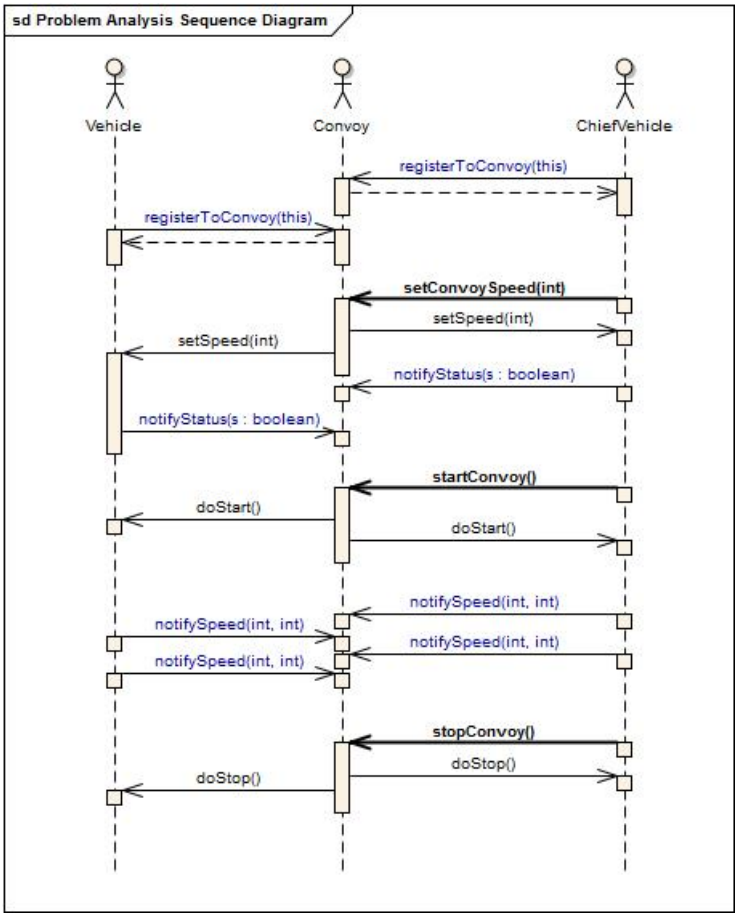

Fig. 5. Logical Architecture - Interaction View

4.3.2 Structural View

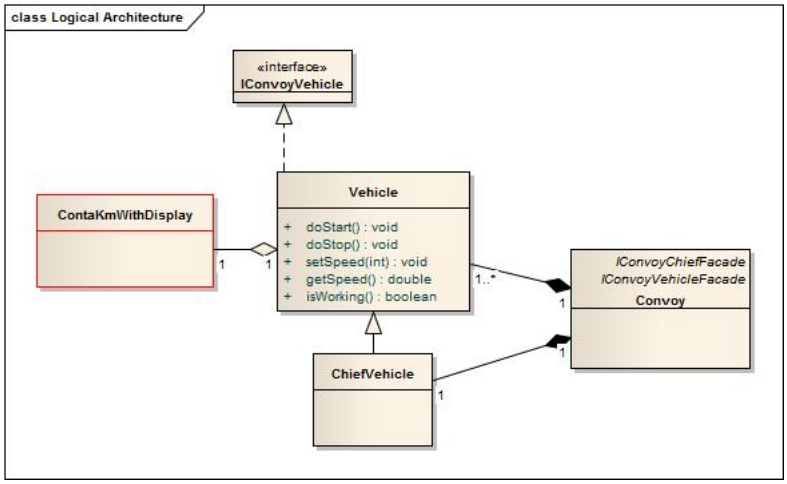

Fig. 6. Logical Architecture - Structural View

8 Notes (outside the template)

We report here some general consideration for the next lab:

- The solution to a problem defines ***how*** we achieve some goal (how we design and implement a software system that satisfies the requirements).
- Before presenting *how* we achieve a goal, we must explicitly define ***what*** must be achieved.
- If we want to verify that our code works as it is expected, we must define *what is expected* (e.g. by means of a set of *testing plan*).
- To be sure that all the tests are performed, all the test should be done in *automatic* way each time we modify the code.

9 Testing

- *Black box testing*: testing on the target public API without knowledge of the target source code
- *White box testing*: testing with knowledge of the target source code

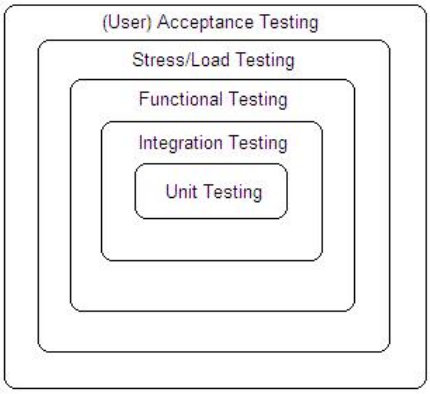

Fig. 7. Test types

- *Unit Testing*: testing single units of work
- *Integration Testing*: testing how different units of work interact
- *Functional Testing*: testing subsystems (usually on a boundary API)
- *Stress/Load Testing*: testing the system performance
- (*User) Acceptance Testing*: testing the system as a user

10 Next steps overview

1. Requirement analysis: from names to entities and from verbs to actions.
2. The concept of *application domain*.
3. "What to do" : `IContaKm`, an interface that sets a standard.
4. "What is expected" : black-box *unit testing plans*.
5. Problem analysis: a fundamental step to face the risks and to find critical aspects. The interaction problem: methods vs. messages.
6. Problem analysis: a fundamental step (for a company or a project team) to define a *working plan*.
7. The (`POJO`) `ContaKm` as a organism: the concept of invariant.
8. A first main question: is platform-independent (technology-independent) design possible?
9. The project of the structure and behavior of the (`POJO`) `ContaKm`.
10. A logic classification of the operations: *primitives, selectors,...*
11. The problem of a `display` for the `ContaKm`.
12. The implementation of the (`POJO`) type `ContaKm`. Reusing available resources: the case of a (cyclic) `Counter`.
13. Automatic testing of the final unit via `JUnit`.

References

1. A. Natali and A. Molesini. *Costruire sistemi software: dai modelli al codice*. Esculapio, 2009.