IUBH INTERNATIONALE HOCHSCHULE FERNSTUDIUM

Project: Software Engineering

Course DLMCSPSE

Portfolio Project

SENOUCI BRIKSI Djelloul

Registration number: 9225022

23.06.2024

Fribourg, Switzerland

Table of contents

Image directory

Glossary

Action	A visual or acoustic information that is shown to the passengers during a line run. Typical actions are for example announcing the next station on the loudspeakers or showing some special messages on the displays.
CAB	Complex Action Browser: The name of the product (tool) being realized in our project.
Complex Action	An action which can contain other complex action(s) or simple action(s). See ”Appendix A - Complex Actions” for more details.
Database	An SQLite database which contains all train numbers and actions used by the PIS software.
Line	In the context of this project, a Line is equivalent to a Train Number or Route. See ‘Train Number’.
PIS	Passenger Information System: The whole system handling the information that are supplied to passengers on public transport vehicles (trains, buses, ...)
Route	In the context of this project, a Route is equivalent to a Train Number or Line. See ‘Train Number’.
Simple Action	Relates to an 'Action' as defined above. The wording 'Simple' is added to distinguish a ‘Simple’ from a ‘Complex’ action.

Train Number	A Route or Line which is ran/traveled by a train from a start station to an end station via some intermediate stations. A Train Number is identified in the database by a Name and a TrainNumberID.
Trigger	A point in the route where the train must execute some actions. Typical triggers are for example opening the doors, closing the doors, or driving a certain distance to or from a station.

Introduction

This document describes the realization of our project for the course ‘Project: Software Engineering, DLMCSPSE’.

The topic of our project is the design and development of a ‘Complex Action Browser’ (CAB).

The CAB is a web application that will enable browsing complex actions contained in a Passenger Information System (PIS) database. The PIS database contains among others the actions that are executed during a train run. Such actions can be very complex and complicated to develop and debug. The CAB would enable developers of our company better develop and debug complex actions trees.

This document contains all the steps realized during the project: requirements definition, software design, project plan, testing and lessons learned.

The document will be continuously updated during the three phases of the project.

Requirements

This section contains the functional and non-functional requirements of our project. The requirements below are ordered following a top-down view. This order does *not* define the priority of the requirement. It is intended to implement all requirements listed in this section within the realization of this project.

Functional Requirements

The CAB is web-based application that allows browsing complex action trees.

The CAB shall be able to load (access) a PIS database from the user's machine. The PIS database consists of an SQLite file.

The CAB shall load complex actions that are defined in train numbers (lines) contained in the database.

The CAB shall show the corresponding list of train numbers contained in the database.

Train Numbers are usually valid on certain dates. The CAB shall give a possibility to choose a date to filter with.

Furthermore, the CAB shall give a possibility to filter (search) a certain Train Number by its identifier (Name and/or TrainNumberID).

Train Numbers are usually defined with Triggers. The CAB shall show the list of Triggers that are attached to a selected Train Number.

Triggers are usually defined with Actions. The CAB shall show the list of Actions that are defined within a selected Trigger. Simple and Complex Actions shall be distinctly identified (for example with a specific color).

The CAB shall allow browsing the content (tree) of a root Complex Action, by collapsing and expanding the tree nodes.

How this tree of complex actions is visually presented will be defined during the implementation phase.

The CAB shall show the most important artefacts of a complex action tree.

The following artefacts shall be supported by the CAB:

- Complex Action Type, such as 'Serial' or 'Parallel'. See "Appendix A - Complex Actions" for a complete list of supported complex action types.

- Complex Action Attributes, such as ‘ReportStart’ or ‘ReportEnd’. See “Appendix A - Complex Actions” for a complete list of complex action attributes.
- Action Attributes, such as ‘ActionID’ or ‘ActionType’. See “Appendix A - Complex Actions” for a complete list of action attributes.

The following diagram shows the main features of the Complex Action Browser:

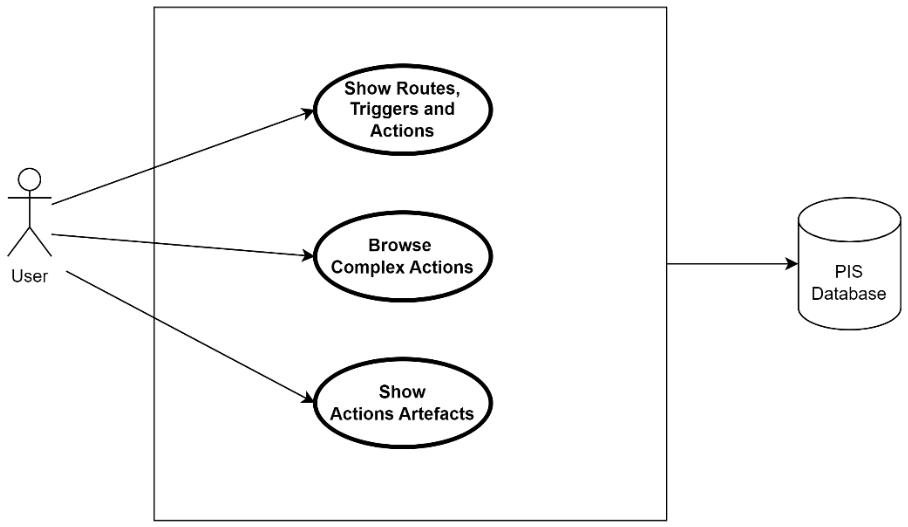


Figure 1 – CAB: Main Features

Non-Functional Requirements

The CAB shall operate on Google Chrome browser. It can optionally operate on Mozilla Firefox and Microsoft Edge browsers.

The CAB shall be able to load databases up to 1 GB.

When collapsing or expanding a complex action tree, the CAB shall render with a maximum of 300ms response time.

If the database cannot be read (eg. corrupt or wrong/unknown structure), the CAB shall show a corresponding error message.

Design

Scope and Context

The Complex Action Browser (orange box in Figure 2) defines the scope of our project.

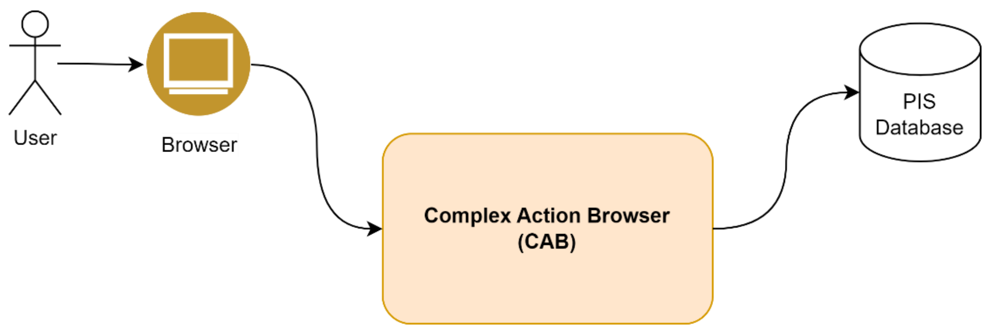


Figure 2 – CAB: Context Diagram

It is mainly accessed via a standard browser. It loads data from a PIS database.

The CAB does not provide nor use any other interface.

Components

The CAB will be implemented as a backend and a frontend application.

The backend application is written in python and uses the 'django' framework (https://www.djangoproject.com/) to access the database.

We have chosen python and django because we have some experience with programming in python. Furthermore, django already supports SQLite databases. Moreover, we found many examples in the net concerning the integration of django into Google cloud. Indeed, it is required that our application must be hosted in a cloud.

The frontend application will be implemented in a classical html/css/js solution. We will use the 'vue js' framework (https://vuejs.org/) for the general web user interface, and the special library 'd3 js' (https://d3js.org/) to visualize and browse trees of elements.

We have chosen vue js because this library is already in use in our company. So, we can get support from experienced colleagues in this matter. The choice of the d3 js library is motivated by the fact that this library is great help when trying to visualize complex graphs (such as trees) in Javascript.

In the backend application, the ‘Data Reader’ component will access the database to retrieve the train numbers (routes) with the corresponding triggers and the attached actions. The ‘Data Handler’ component converts then and prepares these data in a Json-structure, that is delivered to the frontend application.

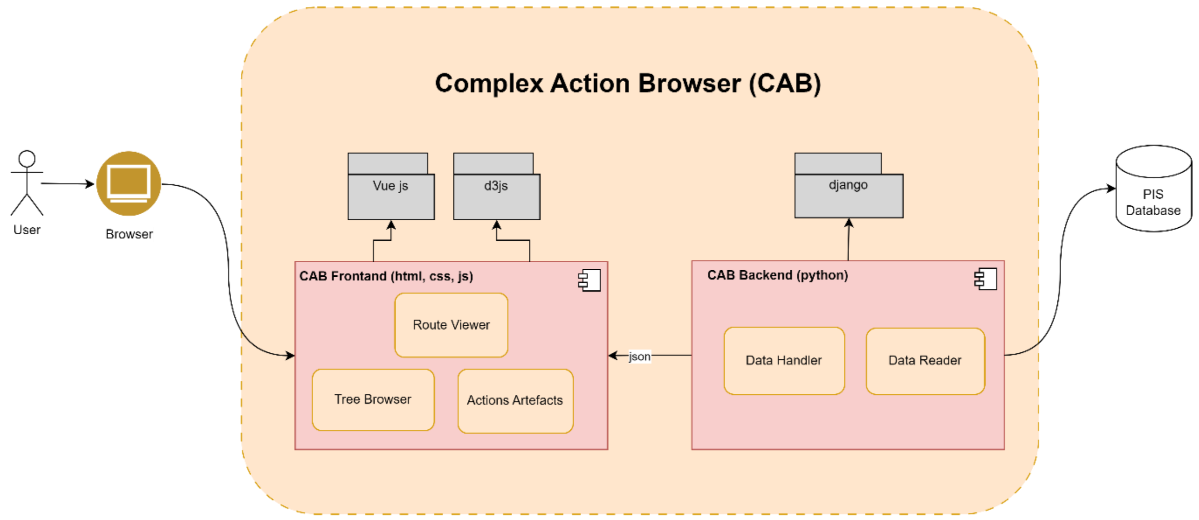


Figure 3 – CAB: Component Diagram

In the frontend application, the ‘Route Viewer’ component parses these json-data and presents the Routes (Train Numbers) to the user. Routes, Triggers and Actions are viewed through this component. Once selecting a Complex Action, component ‘Tree Browser’ enables browsing into this tree (collapsing and expanding the tree nodes). Component ‘Action Artefacts’ gives then the possibility to view actions artefacts (attributes, etc).

Realization and Project Plan

Software Development Methodology

We will realize our project following Scrum agile methodology. There are many reasons for that:

- The main feature of our product is how to best visualize and browse complex action trees. The exact way on how to visualize such trees is not defined as the beginning of the project. We need to make several tries and iterations, to find the best way to achieve it. Thus, an agile approach is the best choice to do it.
- Our customers are software developers (of our company). We will be in close contact with them everyday to better match their needs. We need to quickly react to their needs and wishes (as long as within the scope of the requirements and allocated time). The agile approach will help us keep this flexibility.
- The customers (developers) will be able to use the very first deliveries of the product, which shall mainly contain the browsing of complex action trees (see sprints below). This means that the product will be used even if not yet finalized. Developers will always get the latest – thus not yet final – version of the product, to play with. The agile approach is ideal for this.

We will be assuming all the following roles during the realization of the product:

- Scrum Product Owner
- Scrum Master
- Developer
- Tester (though some developer colleagues will also test delivered versions regularly, but not as formally as we will do).

As we are the only person in the team, no daily Scrums (standups) will be conducted.

Tools

The following tools will be used during our project:

- Visual Studio Code: IDE to develop the frontend and backend applications.
- Jira: project plan, bug tracking
- GitHub: version control

General Plan

As module ‘Project: Software Engineering DLMCSPSE’ is allocated with 5 ECTS, and one ECTS corresponds to 25 to 30 study hours, we have defined a maximum of 150 hours for our project.

Moreover, we are taking our master studies in part-time, in parallel to our work. We are planning to allocate about 16 hours / calendar week for the realization of the project.

The general plan to realize the three phases of the project is the following:

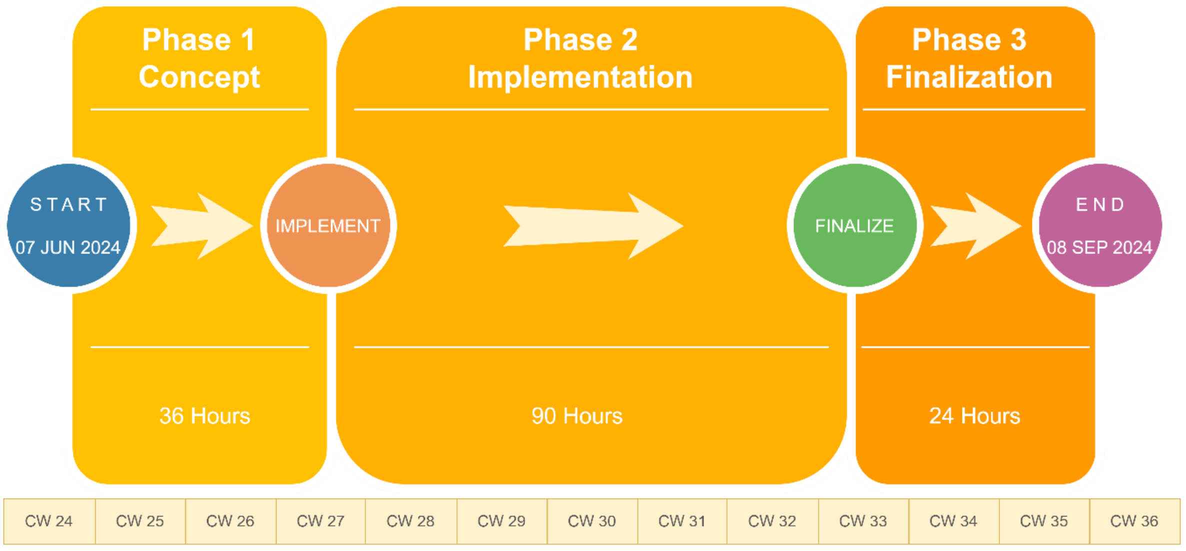


Figure 4 – General Project Plan

We intend to realize the project within 13 weeks, starting (already started at the time of this writing) from calendar week 24. We have foreseen about two weeks as buffer, for unexpected delays or to take some days off.

Backlog and Sprints

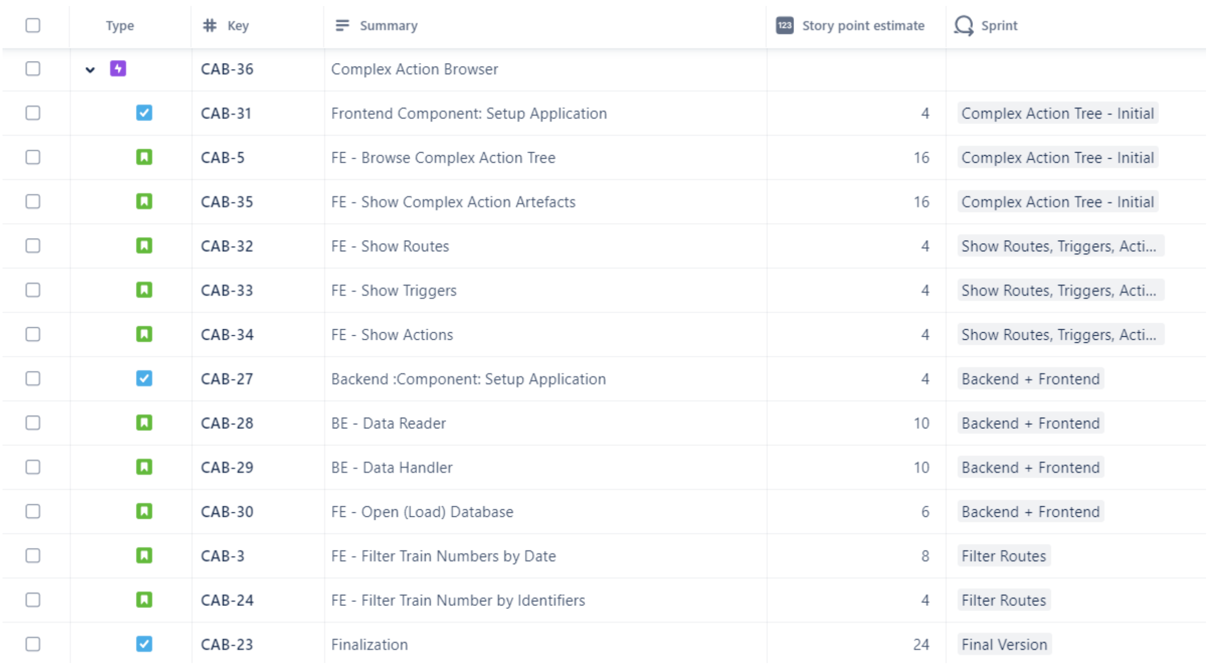
As our project is relatively small, we will have only one Epic: the ‘Complex Action Browser’.

We have planned to realize the project in five Sprints. Figure 5 shows the backlog as entered in Jira, with the corresponding Stories and Tasks:

The sprint length is different for each sprint. Please note that the story points correspond to estimated effort in hours.

Some Stories and Tasks have prefix ‘BE’ or ‘FE’: this corresponds to the ‘Backend’ or ‘Frontend’ application, respectively.

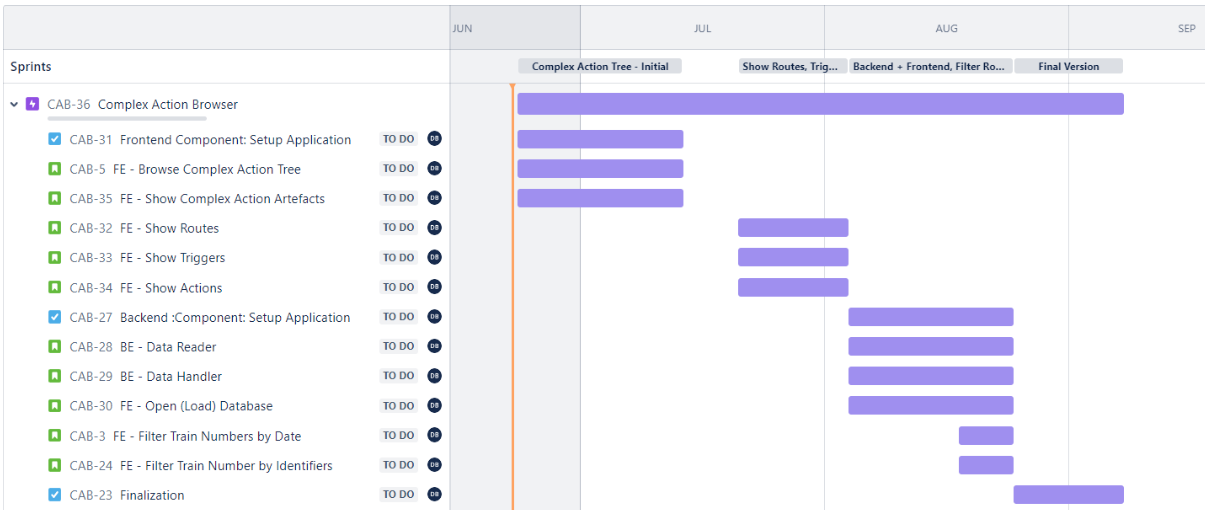
Testing activities (see chap. Testing) will be performed during the realization of the corresponding sprints.



Type	Key	Summary	Story point estimate	Sprint
	CAB-36	Complex Action Browser		
	CAB-31	Frontend Component: Setup Application	4	Complex Action Tree - Initial
	CAB-5	FE - Browse Complex Action Tree	16	Complex Action Tree - Initial
	CAB-35	FE - Show Complex Action Artefacts	16	Complex Action Tree - Initial
	CAB-32	FE - Show Routes	4	Show Routes, Triggers, Acti...
	CAB-33	FE - Show Triggers	4	Show Routes, Triggers, Acti...
	CAB-34	FE - Show Actions	4	Show Routes, Triggers, Acti...
	CAB-27	Backend :Component: Setup Application	4	Backend + Frontend
	CAB-28	BE - Data Reader	10	Backend + Frontend
	CAB-29	BE - Data Handler	10	Backend + Frontend
	CAB-30	FE - Open (Load) Database	6	Backend + Frontend
	CAB-3	FE - Filter Train Numbers by Date	8	Filter Routes
	CAB-24	FE - Filter Train Number by Identifiers	4	Filter Routes
	CAB-23	Finalization	24	Final Version

Figure 5 – Backlog and Sprints (snapshot Jira)

Figure 6 shows the sprints timeline, which is aligned with the general project plan presented above.



CAB-36 Complex Action Browser
CAB-31 Frontend Component: Setup Application TO DO
CAB-5 FE - Browse Complex Action Tree TO DO
CAB-35 FE - Show Complex Action Artefacts TO DO
CAB-32 FE - Show Routes TO DO
CAB-33 FE - Show Triggers TO DO
CAB-34 FE - Show Actions TO DO
CAB-27 Backend :Component: Setup Application TO DO
CAB-28 BE - Data Reader TO DO
CAB-29 BE - Data Handler TO DO
CAB-30 FE - Open (Load) Database TO DO
CAB-3 FE - Filter Train Numbers by Date TO DO
CAB-24 FE - Filter Train Number by Identifiers TO DO
CAB-23 Finalization TO DO

Figure 6 – Sprints Timeline (snapshot Jira)

Here is now a description of the sprints goal and content:

Sprint 1: 'Complex Action Tree – Initial'

The heart of our project is to be able to browse complex action trees. So the goal of this first sprint is to implement a very first 'shot' of browsing a complex action tree. The idea is to only implement the frontend application, with some hard-coded data (real data, but not coming from backend) and try to find out and choose what is the best d3js library that fits our need. We will have the possibility to show this first shot to our customers (developers) and adapt it accordingly.

Once again, the idea of this first sprint is to implement as quick as possible the very most important feature of our product and get early feedback from customers and a first ‘feeling’ about the relevancy and correctness of our design and plan.

Sprint 2: ‘Show Routes, Triggers, Actions’

Here again, we will use hard-coded (though real) data to continue building our frontend application, by adding the features of showing the routes (train numbers), triggers and actions. The user can then select a route, trigger and complex action and browse through the selected complex action tree.

Sprint 3: ‘Backend + Frontend’

This is the first end-to-end version. The backend application will be implemented and linked to the frontend application. We will then show and *give* this version (though not yet final) to customers (developers), to play with.

Sprint 4: ‘Filter Routes’

The last missing feature (to filter a route by date or identifiers) is added in this sprint. This is thus the first complete version of the CAB.

Sprint 5: ‘Finalization’

The product is here finalized after getting feedback from customers and from the tutor. This is the final delivery.

Risks Analysis

We have identified the following business and technical risks for our project, and thought about corresponding mitigation measures:

Risk	Mitigation
Business Risks	
Lack of User Acceptance: The CAB may not be used in practice by customers (developers).	Work in close with developers (agile); get early feedback; adapt accordingly.

No more Use of Complex Actions in PIS: The PIS software people may decide to change the concept of Complex Actions to something else. The whole CAB would then be obsolete.	PIS projects are usually long-term projects (more than 5 years). Complex Actions are heavily used in current projects. The CAB – if accepted by users- would probably still be in use in the next years.
Technical Risks	
Dependency on 3rd-party library: The main feature (browsing trees) is based on the external d3js library. If this latter is no more available, the CAB would be affected.	A survey on the internet shows that the d3js is actively maintained and used by the community. There should be no risk in this matter.
Lack of experience with external libraries: We will be using d3js and vuejs external libraries. We have no previous experience with these libraries.	There are enough examples on the internet on how to use and integrate d3js and vuejs in projects. Furthermore, the vuejs library is already in use in our company, where we can have access to experienced people who are using this library.

Testing

End-To-End Testing

The main testing method for our project will be end-to-end testing. Effective tests will start by Sprint 3, although the tool will already be tested – informally – in Sprints 1 and 2.

We will provide test databases to test all functional and non-functional requirements.

Unit Testing

We intend to make *one* unit testing: to test the Json-Interface between the backend and frontend application. Indeed, the Json-Data (transmitted through the Json-Interface) are the basis data our tool rely upon. So, it's important to always check the integrity of the data going through this interface.

Unit testing of the Json-Interface will be performed during Sprint 3.

Conclusion - Lessons Learned

(Phases 2 and 3)

Appendix A - Complex Actions

Example

Figure 7 shows a simple example of a Complex Action:

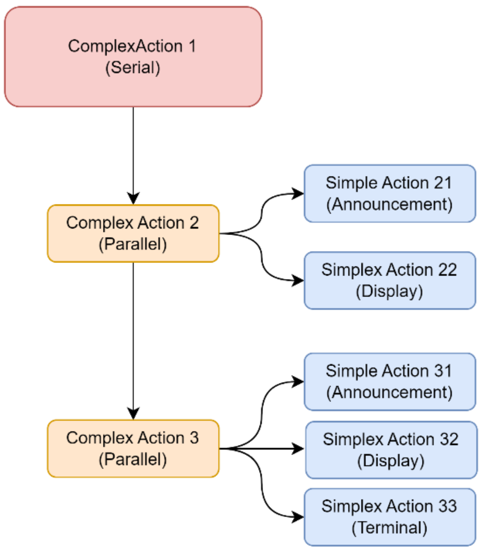


Figure 7 – Example of a Complex Action

'Complex Action 1' is the root action: the very first action which is executed on a certain trigger (for example, door opening).

This root complex action is of type 'Serial' and has two children actions, which are also complex actions: 'Complex Action 2' and 'Complex Action 3'.

The 'Serial' type means that the two children actions ('Complex Action 2' and 'Complex Action 3') will be executed one after the other (in Serie).

'Complex Action 2' is of type 'Parallel' and has two children actions, of type simple actions: 'Simple Action 21' will announce something to the passengers (on the loudspeakers), and "Simple Action 22' will show something to the passengers (on the displays).

The type 'Parallel' means that the two simple actions will be played simultaneously (in Parallel).

'Complex Action 3' is also of type 'Parallel' and has three children actions: 'Simple Action 31' (announces something to passengers), 'Simple Action 32' (shows something to passengers) and 'Simple Action 33' (shows something on the terminal of the driver). These three simple actions will be played in Parallel.

Complex Action Types

The following complex action types are supported:

Serial, Parallel, MinTime, MaxTime, ExactTime, Repeat, ExecuteOnce, EnableByExtSource, CancelByExtSource, EnableOnceByExtSource, BlockByExtSource.

Depending on the type, a complex action can have a *main* attribute which defines the behavior of the complex action. For example, a complex action of type ‘Repeat’ has an attribute ‘Count’, which defines in how many iterations shall the complex action be repeated.

The following complex action main attributes are supported:

- MinTime: Time in [s]
- MaxTime: Time in [s]
- ExactTime: Time in [s]
- Repeat: Count (Number of Repetitions)
- EnableByExtSource: Source Name
- CancelByExtSource: Source Name
- EnableOnceByExtSource: Source Name
- BlockByExtSource: Source Name

Complex Action Attributes

Furthermore, Complex Actions can also have other attributes, such as ‘ReportStart’ and ‘ReportEnd’ (reporting a message when a complex action starts or ends).

The following complex action attributes are supported:

- ReportStart: Message Name
- ReportEnd: Message Name
- Categories: List of Category Names

Action Attributes

Moreover, Actions (complex or simple) are identified by specific identifiers such as ActionID or ActionType. These identifiers help identifying a specific action in the whole database.

The following Action Identifiers are supported:

- ActionID: unique identifier of the action in the database
- ActionDetailID: second identifier (foreign key to other tables)
- ActionType: ‘Announcement’, ‘Display’, etc
- MediaType: On which device (media) shall the action be played (for example ‘Internal Loudspeakers’, ‘External Displays’, etc).

References

- Django Framework
  https://www.djangoproject.com/
- Vue js
  https://vuejs.org/
- D3 js
  https://d3js.org/