

High-level Architecture Documentation

About arc42

arc42, the template for documentation of software and system architecture.

Template Version 8.2 EN. (based upon AsciiDoc version), January 2023

Created, maintained and © by Dr. Peter Hruschka, Dr. Gernot Starke and contributors. See <https://arc42.org>.

This version of the template contains some help and explanations. It is used for familiarization with arc42 and the understanding of the concepts. For documentation of your own system you use better the *plain* version.

Introduction and Goals

This project is an effort of establishing a development workflow via practical product development.

There are two main requirements for the project:

- The product should be a working and entertaining for a cat autonomous toy.
- Components, processes and libraries of the project must be developed to be reusable in other projects.

The project is based on the following principles:

1. **Architecture-oriented**

- Develop the right architecture using architectural best practices

2. **Result-oriented and minimalistic**

- Progress over perfection

3. **Integration-oriented**

- Reusing existing libraries, projects, etc. Minimum development of individual pieces

4. **Reusable in my other projects**

- It should be done automatically with a good architecture. But I want to make several special decisions to make the project easy to transfer to my another robotic project.

Quality Goals

No	Quality	Motivation
1	Cat Friendliness	The product should be interesting and attractive for a cat
2	Safety	The product should be safe for pets, it should not harm the pet in any way
3	Devellopment Reusability	The project components should be easy to reuse in other projects. Especially in my ouw projects

Stakeholders

Role/Name	Expectations
Cats (especially Bastet)	The cats should get a simple funny toy that is able to entertain them
Developers	Developers must get an polished transparent quality product that can be reused or valuable from the education point of view
Andrei Gramakov	I suppose to do two things at once: 1. improve my architect skills, 2. move my another enormously big robotic project.
Potential employers of Andrei Gramakov	Should get an allustrative entry in my portfolio to better understand my skills and experience.

Architecture Constraints

Software:

- Should follow the Clean Architecture principles
- Should be modular and reusable
- Hardware details should go to the thin last layer
- Most of the development should be done platform-independent.
- Movement logic should be easy detachable of the application logic - to be able to use it in other projects
- Programming language is C/C++

Hardware:

- Motors must be quite

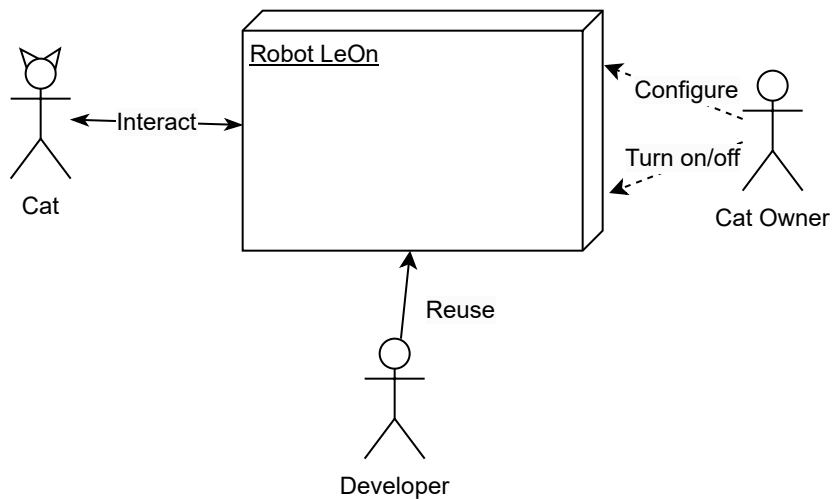
System Scope and Context

Business Context

The business context is represented by a diagram with all counterparts interacting with the system.

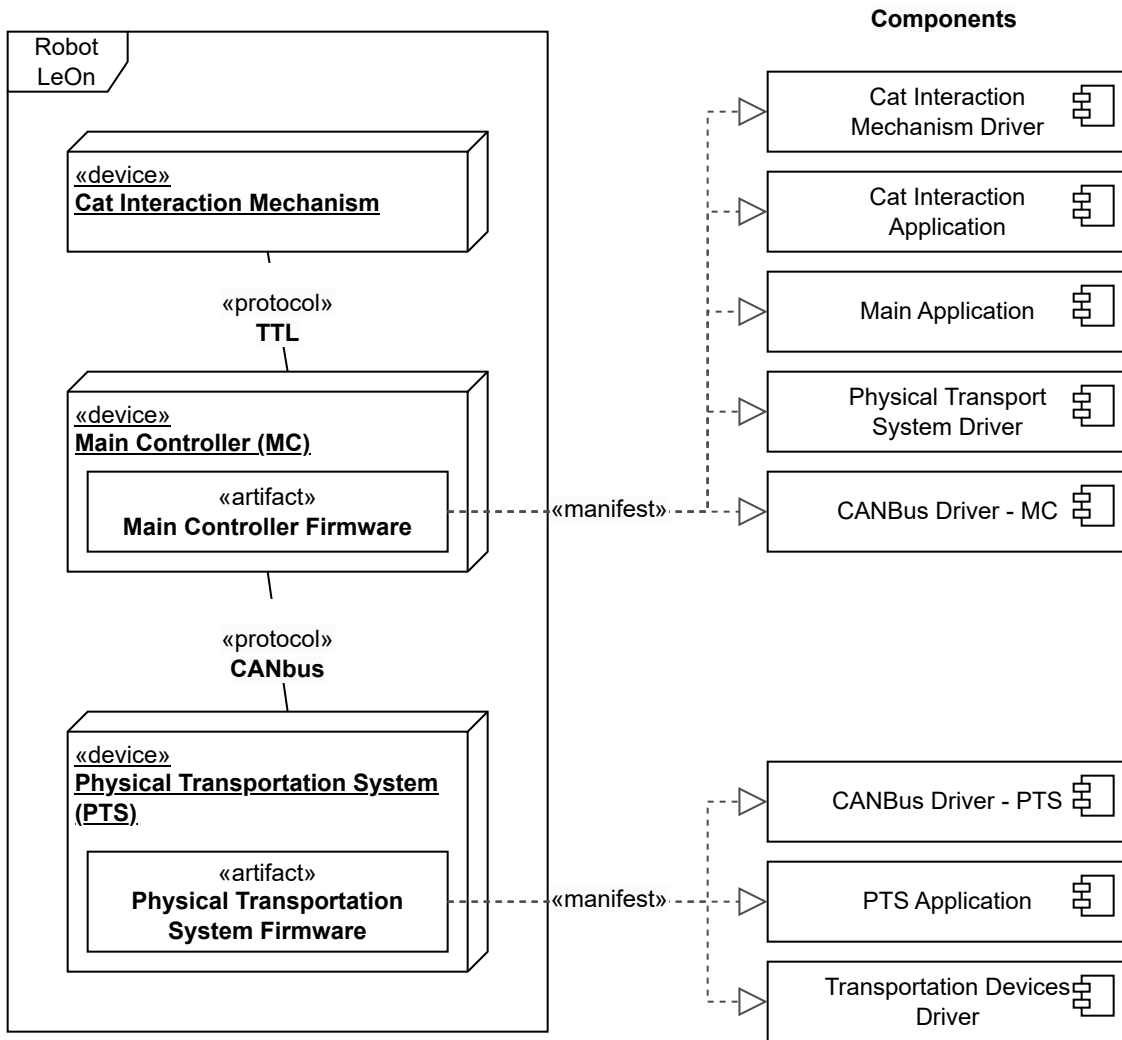
Communication partners:

- Cat
- Cat owner
- Developers



Technical Context

The technical context is represented by a UML deployment diagram.



Solution Strategy

1. Physical Transportation System is a separate physical component

Solution approach. The System can be considered as a separate device accepting commands from the main controller via a CANbus protocol.

2. Physical Transportation System is an independent and unit-tested piece of software. The rest of the system is tested via integration or black-box tests

Solution approach. It has a separate location with it's own file structure - src, include, test, etc. Unit testing is done only for the Physical Transportation System.

3. Cat Interaction Logic is isolated from the main app

Solution approach. Cat Interaction Logic interacts with the main app via an abstract interface. The logic itself is enclosed in an entity, and interacts with the physical components also via an abstract interface.

4. Besides the drivers, all components are platform-independent

Solution approach. Each components besides the applications should be a separate library.

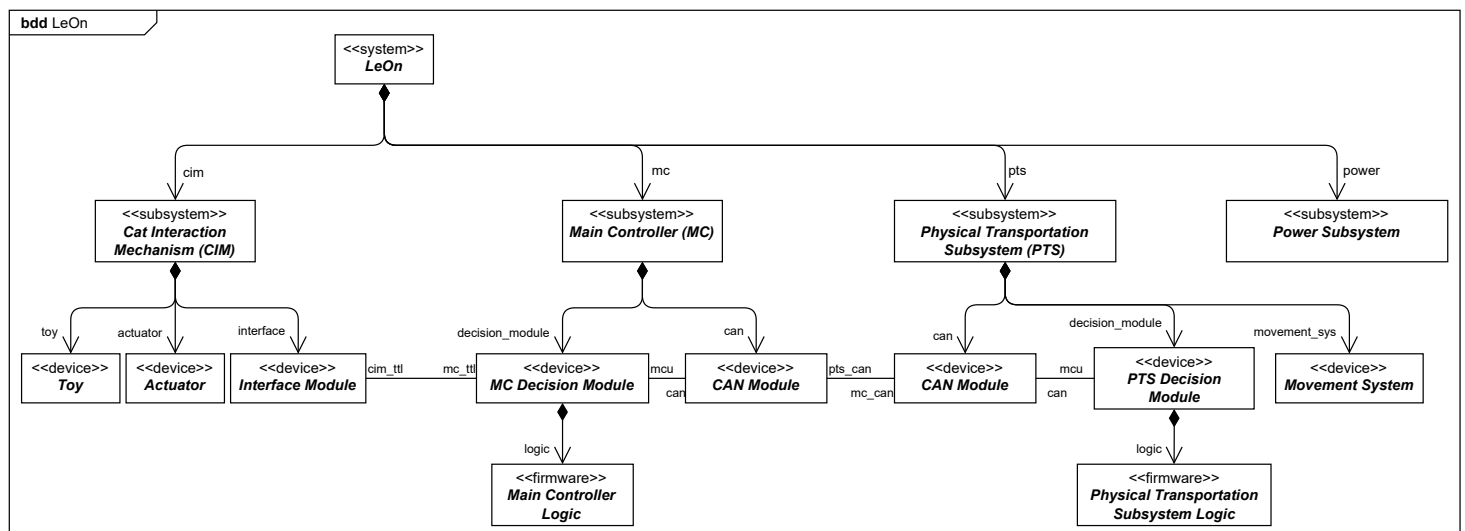
5. Most of the development is done without involving the hardware

Solution approach. Drivers should have an abstract interface, and the should have fake implementation of it for the development environment (e.g. a desktop OS).

Building Block View - System

Level 1 - White Box: Overall System

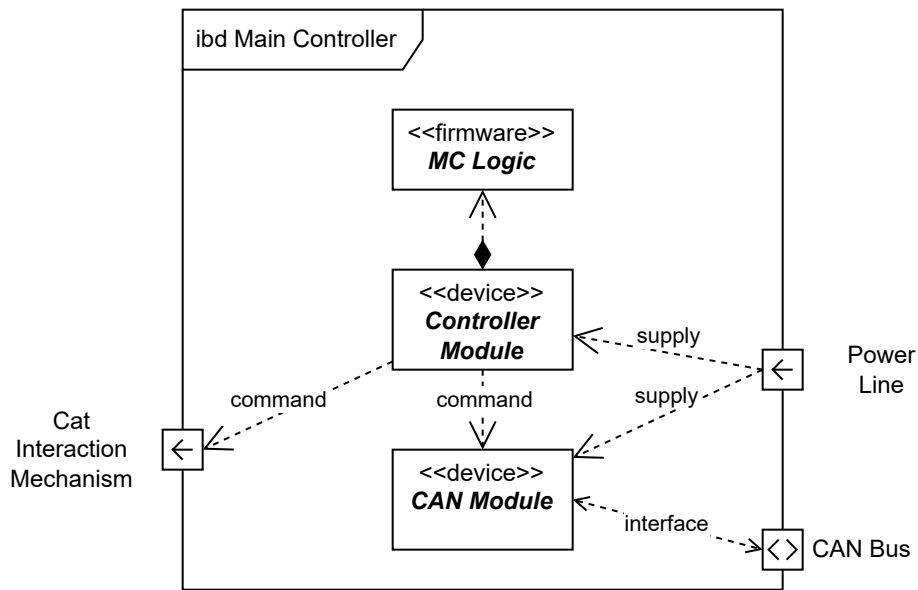
The decomposition of the system splits the system in three subsystems:



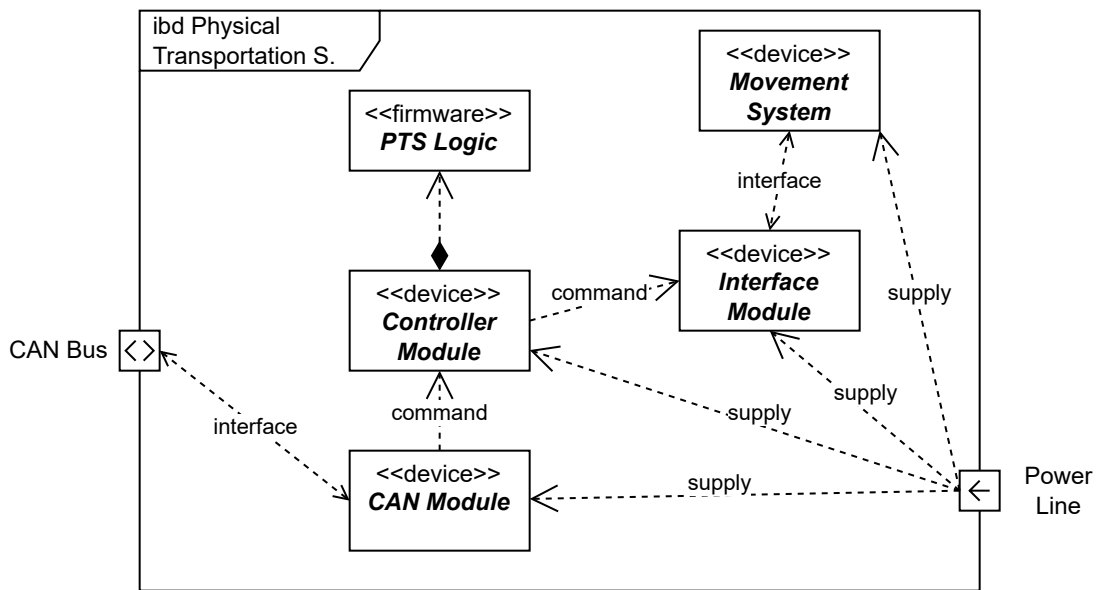
Each subsystem represents a separate physical module of the system.

Level 2

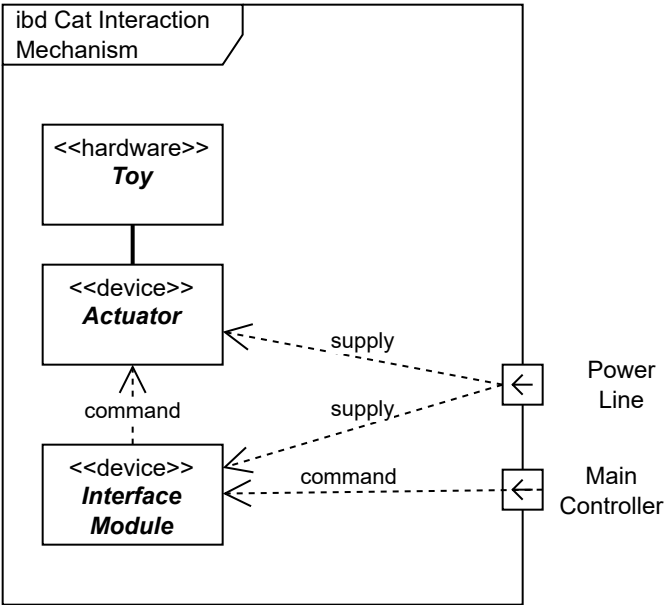
White Box: Main Controller Hardware



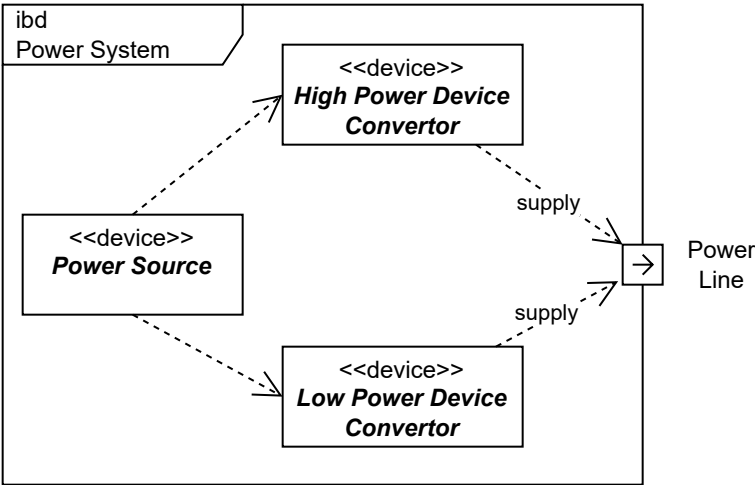
White Box: Physical Transportation System Hardware



White Box: Cat Interaction Mechanism Hardware

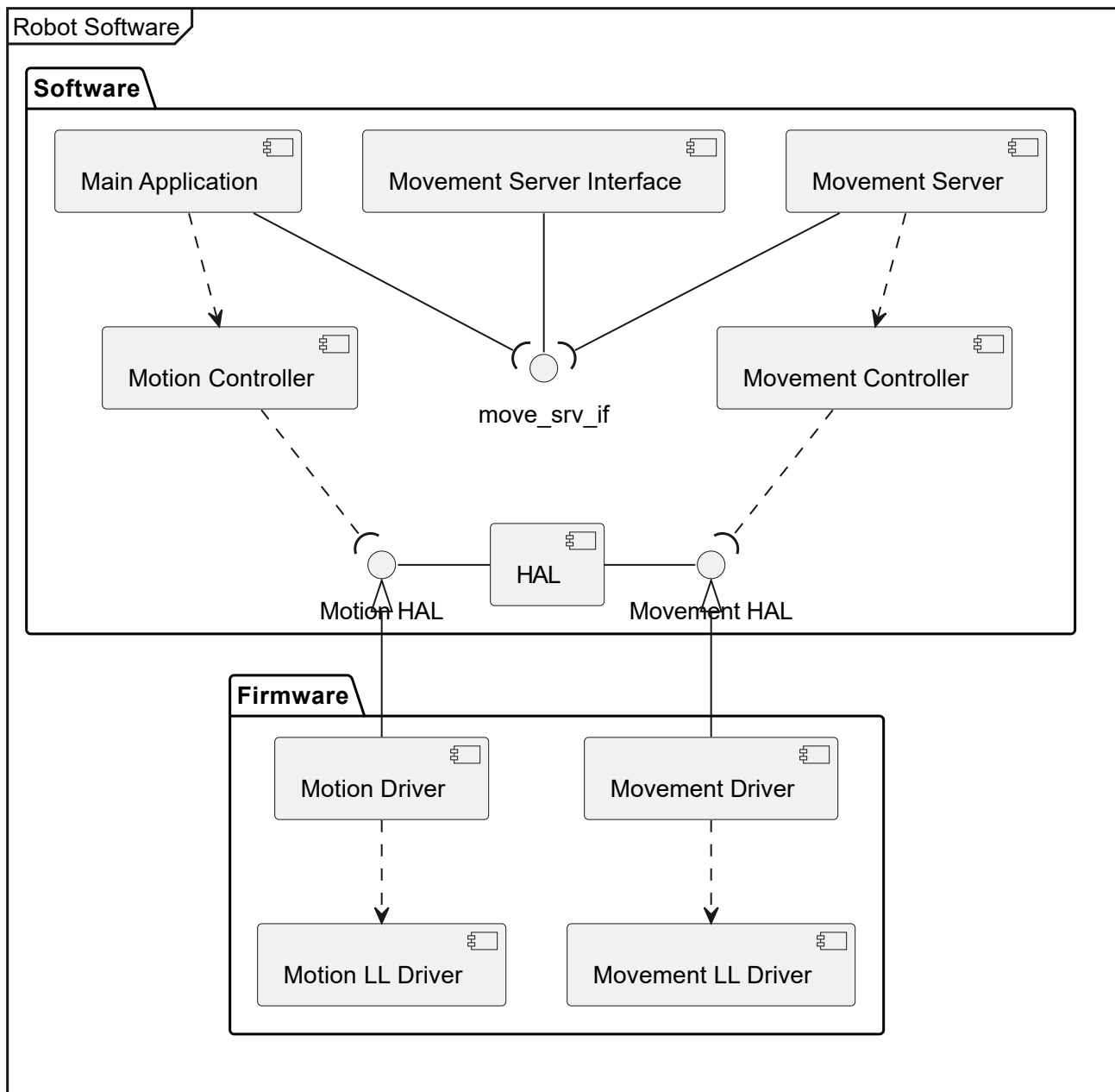


White Box: Power System Hardware



Building Block View - Software

Level 1 - White Box: Overall System



Software:

- **Main Application** - the main application of the system. It is responsible for the overall system control.
- **HAL** - the hardware abstraction layer. It is responsible for interaction with the hardware drivers
- **Motion Controller** - the controller responsible for high-level motion control logic (e.g. utilizing different motion modes to build a specific pattern)

- **Movement Controller** - the controller responsible for high-level movement control logic. E.g. utilizing different movement modes to build a specific pattern on top of the Main Application commands, like shaking, or non-linear movement, user input, supervising, fault mitigation, etc.
- **Movement Server** - the server responsible for accepting the movement commands from the Main Application and translating them to the Movement HAL commands.
- **Movement Server Interface** - the interface for communication between the Movement Server and the Main Application.

Firmware:

- **Movement Driver** - the driver utilizing the Low-Level Movement Driver to implement the Movement HAL.
- **Movement LL Driver** - the low-level driver for the movement hardware (e.g. a step-motor driver)
- **Motion Driver** - the driver utilizing the Low-Level Motion Driver to implement the Motion HAL.
- **Motion LL Driver** - the low-level driver for the motion hardware (e.g. a DC motor driver)

Interface Specification

In this perspective all interface will be described as a set of functions with partial signatures. If the type is not specified, it is up to the implementation.

Movement Server Interface

The interface is similar to ROS's `cmd_vel` topic. See [ROS cmd_vel](#).

- `SetVelocity(int X, int Y, int Z, int AngularX, int AngularY, int AngularZ)` - sets the velocity of the robot.

The origin is the center of the robot. The X axis is directed forward, the Y axis is directed to the left, the Z axis is directed up. The angular coordinates are the rotation around the X, Y and Z axes respectively.

HAL

Movement HAL:

- `MoveX(int8_t Speed)` - moves the robot forward if the speed is positive, backward if the speed is negative.
- `MoveY(int8_t Speed)` - moves the robot to the right if the speed is positive, to the left if the speed is negative.
- `MoveZ(int8_t Speed)` - moves the robot up if the speed is positive, down if the speed is negative.

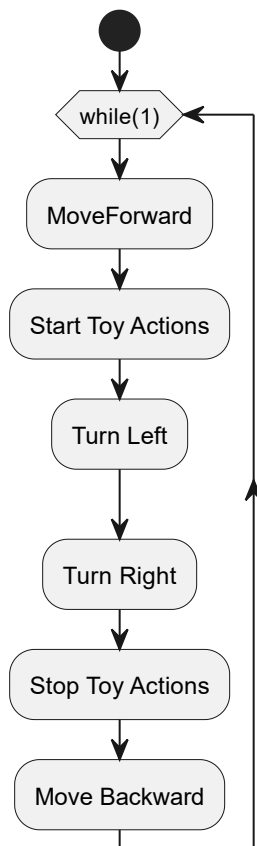
- RotateX(int8_t Speed) - rotates the robot around the X axis clockwise if the speed is positive, counterclockwise if the speed is negative.
- RotateY(int8_t Speed) - rotates the robot around the Y axis clockwise if the speed is positive, counterclockwise if the speed is negative.
- RotateZ(int8_t Speed) - rotates the robot around the Z axis clockwise if the speed is positive, counterclockwise if the speed is negative.

Motion HAL:

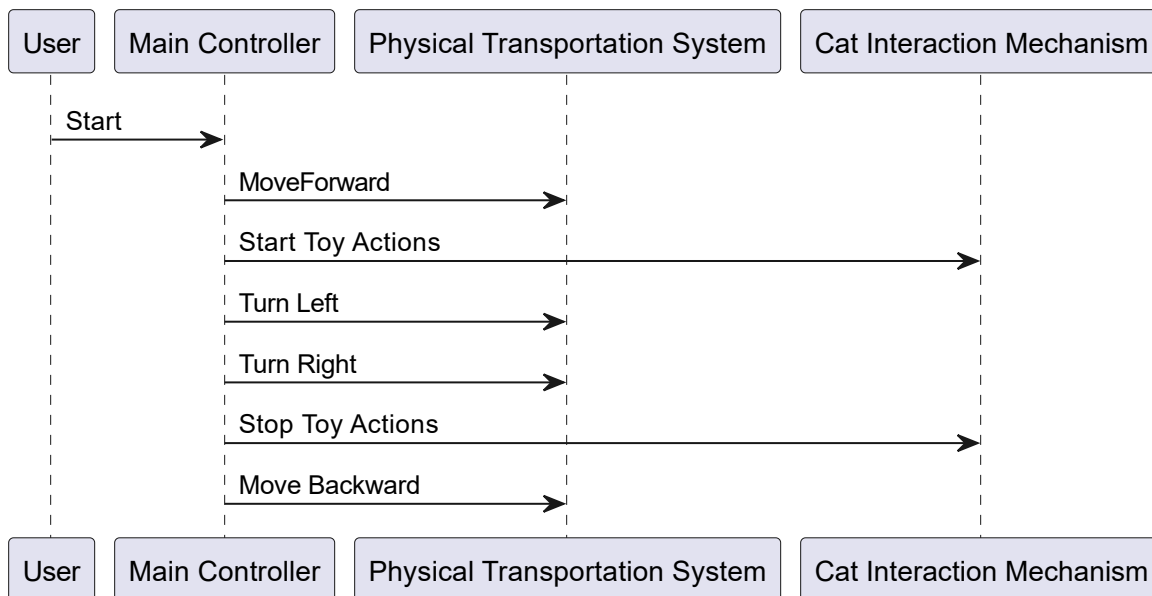
- Start() - starts the motion
- Stop() - stops the motion
- SetMode(Mode mode) - sets the mode of the motion. The modes are up to implementation. If the mode is not supported, the function should return a non-implemented error.

Runtime View

The overall robot algorithm is the following:



The interaction between the robot components is described in the following sequence diagram:



Deployment View

Content

The deployment view describes:

1. technical infrastructure used to execute your system, with infrastructure elements like geographical locations, environments, computers, processors, channels and net topologies as well as other infrastructure elements and
2. mapping of (software) building blocks to that infrastructure elements.

Often systems are executed in different environments, e.g. development environment, test environment, production environment. In such cases you should document all relevant environments.

Especially document a deployment view if your software is executed as distributed system with more than one computer, processor, server or container or when you design and construct your own hardware processors and chips.

From a software perspective it is sufficient to capture only those elements of an infrastructure that are needed to show a deployment of your building blocks. Hardware architects can go beyond that and describe an infrastructure to any level of detail they need to capture.

Motivation

Software does not run without hardware. This underlying infrastructure can and will influence a system and/or some cross-cutting concepts. Therefore, there is a need to know the infrastructure.

Maybe a highest level deployment diagram is already contained in section 3.2. as technical context with your own infrastructure as ONE black box. In this section one can zoom into this black box using additional deployment diagrams:

- UML offers deployment diagrams to express that view. Use it, probably with nested diagrams, when your infrastructure is more complex.
- When your (hardware) stakeholders prefer other kinds of diagrams rather than a deployment diagram, let them use any kind that is able to show nodes and channels of the infrastructure.

See [Deployment View](#) in the arc42 documentation.

Infrastructure Level 1

Describe (usually in a combination of diagrams, tables, and text):

- distribution of a system to multiple locations, environments, computers, processors, .., as well as physical connections between them
- important justifications or motivations for this deployment structure
- quality and/or performance features of this infrastructure
- mapping of software artifacts to elements of this infrastructure

For multiple environments or alternative deployments please copy and adapt this section of arc42 for all relevant environments.

<Overview Diagram>

Motivation

<explanation in text form>

Quality and/or Performance Features

<explanation in text form>

Mapping of Building Blocks to Infrastructure

<description of the mapping>

Infrastructure Level 2

Here you can include the internal structure of (some) infrastructure elements from level 1.

Please copy the structure from level 1 for each selected element.

<Infrastructure Element 1>

<diagram + explanation>

<Infrastructure Element 2>

<diagram + explanation>

...

<Infrastructure Element n>

<diagram + explanation>

Cross-cutting Concepts

Content

This section describes overall, principal regulations and solution ideas that are relevant in multiple parts (= cross-cutting) of your system.

Such concepts are often related to multiple building blocks. They can include many different topics, such as

- models, especially domain models
- architecture or design patterns
- rules for using specific technology

- principal, often technical decisions of an overarching (= cross-cutting) nature
- implementation rules

Motivation

Concepts form the basis for *conceptual integrity* (consistency, homogeneity) of the architecture. Thus, they are an important contribution to achieve inner qualities of your system.

Some of these concepts cannot be assigned to individual building blocks, e.g. security or safety.

Form

The form can be varied:

- concept papers with any kind of structure
- cross-cutting model excerpts or scenarios using notations of the architecture views
- sample implementations, especially for technical concepts
- reference to typical usage of standard frameworks (e.g. using Hibernate for object/relational mapping)

Structure

A potential (but not mandatory) structure for this section could be:

- Domain concepts
- User Experience concepts (UX)
- Safety and security concepts
- Architecture and design patterns
- "Under-the-hood"
- development concepts
- operational concepts

Note: it might be difficult to assign individual concepts to one specific topic on this list.



Possible topics for crosscutting concepts

See [Concepts](#) in the arc42 documentation.

<Concept 1>

<explanation>

<Concept 2>

<explanation>

...

<Concept n>

<explanation>

Architecture Decisions

Contents

Important, expensive, large scale or risky architecture decisions including rationales. With "decisions" we mean selecting one alternative based on given criteria.

Please use your judgement to decide whether an architectural decision should be documented here in this central section or whether you better document it locally (e.g. within the white box template of one building block).

Avoid redundancy. Refer to section 4, where you already captured the most important decisions of your architecture.

Motivation

Stakeholders of your system should be able to comprehend and retrace your decisions.

Form

Various options:

- ADR ([Documenting Architecture Decisions](#))
for every important decision
- List or table, ordered by importance and consequences or:
- more detailed in form of separate sections per decision

See [Architecture Decisions](#) in the arc42 documentation. There you will find links and examples about ADR.

Quality Requirements

Content

This section contains all quality requirements as quality tree with scenarios. The most important ones have already been described in section 1.2. (quality goals)

Here you can also capture quality requirements with lesser priority, which will not create high risks when they are not fully achieved.

Motivation

Since quality requirements will have a lot of influence on architectural decisions you should know for every stakeholder what is really important to them, concrete and measurable.

See [Quality Requirements](#) in the arc42 documentation.

Quality Tree

Content

The quality tree (as defined in ATAM – Architecture Tradeoff Analysis Method) with quality/evaluation scenarios as leafs.

Motivation

The tree structure with priorities provides an overview for a sometimes large number of quality requirements.

Form

The quality tree is a high-level overview of the quality goals and requirements:

- tree-like refinement of the term "quality". Use "quality" or "usefulness" as a root
- a mind map with quality categories as main branches

In any case the tree should include links to the scenarios of the following section.

Quality Scenarios

Contents

Concretization of (sometimes vague or implicit) quality requirements using (quality) scenarios.

These scenarios describe what should happen when a stimulus arrives at the system.

For architects, two kinds of scenarios are important:

- Usage scenarios (also called application scenarios or use case scenarios) describe the system's runtime reaction to a certain stimulus. This also includes scenarios that describe the system's efficiency or performance. Example: The system reacts to a user's request within one second.
- Change scenarios describe a modification of the system or of its immediate environment. Example: Additional functionality is implemented or requirements for a quality attribute change.

Motivation

Scenarios make quality requirements concrete and allow to more easily measure or decide whether they are fulfilled.

Especially when you want to assess your architecture using methods like ATAM you need to describe your quality goals (from section 1.2) more precisely down to a level of scenarios that can be discussed and evaluated.

Form

Tabular or free form text.

Risks and Technical Debts

Contents

A list of identified technical risks or technical debts, ordered by priority

Motivation

“Risk management is project management for grown-ups” (Tim Lister, Atlantic Systems Guild.)

This should be your motto for systematic detection and evaluation of risks and technical debts in the architecture, which will be needed by management stakeholders (e.g. project managers, product owners) as part of the overall risk analysis and measurement planning.

Form

List of risks and/or technical debts, probably including suggested measures to minimize, mitigate or avoid risks or reduce technical debts.

See [Risks and Technical Debt](#) in the arc42 documentation.

Glossary

Contents

The most important domain and technical terms that your stakeholders use when discussing the system.

You can also see the glossary as source for translations if you work in multi-language teams.

Motivation

You should clearly define your terms, so that all stakeholders

- have an identical understanding of these terms
- do not use synonyms and homonyms

A table with columns <Term> and <Definition>.

Potentially more columns in case you need translations.

See [Glossary](#) in the arc42 documentation.

Term	Definition
<Term-1>	<definition-1>
<Term-2>	<definition-2>