Software Systems Engineering
Final project process report

Roberto Casadei Massimiliano Martella Roberto Reda

March 4, 2015

1 Introduction

This report represents a concise, informative, and annotated **snapshot** of the final project for the course "Software Systems Engineering" at Alma Mater Studiorum - University of Bologna.

2 Vision

Product and process

We believe that a good software product is the natural result of a *mature, systematic process* that, as such, should be repeatible, defined, managed and – if we make it – even optimizing (see *Capability Maturity Model*).

Why ⇒ What ⇒ How

We believe that there is no code without design, no design without analysis, no analysis without requirements. So, we recognize the causal progression from the *why* (vision/goals/requirements) to the *what* (analysis) and finally to the *how* (design/code).

Projects and knowledge

Software projects are inherently **knowledge-intensive**. As such, we think it is prominent to look for better ways to:

- *Make project knowledge **explicit*** (cf. project artifacts, their documentation, and the documentation about their craft)
- *Make project knowledge accessible, searchable, navigable, ...* (cf. project websites, wikis, issue tracking systems, ...)
- *Keep the information **consistent*** (cf. editing of different representations of the same resource, the relationship between analysis and design artifacts)
- ***Link** related information* (cf. traceability)
- *Keep track of changes* (cf. version control, continuous testing/build/delivery systems)
- ...

Make the best out of analysis

We recognize the importance of analysis for a mature process, but we don't want to waste time by just producing informal knowledge. Instead, we would like to *capitalize the effort spent in analysis by producing something that has value, that is, working software (prototype) and reusable frameworks.*

Reuse

We believe that *reuse* is prominent in what we do because it is *the key to efficiency and effectiveness*, to productivity and quality.

Reuse can happen at several levels and forms, but we are ultimately interested in *code reuse.*

Starting from problems: top-down approach

We firmly believe that the results from (requirements and problem) analysis should NOT depend on the reference paradigm and, least of all, on the technological platform.

Thus, we strive to follow a *top-down approach*, that is, from the problem to the technology.

We are aware that paradigms (and, in general, the conceptual spaces projected by a given technology) are similar to *lens* that make certain aspects clearly visible while completely hiding others. This is not bad, because **complex systems** typically consists of multiple **(orthogonal) dimensions**, and each dimension might require different (conceptual and practical) tools to be adequately investigated.

As a consequence, we think that, in order to be able to effectively tackle the problems at hand, we should

1. be aware of the limitations of adopting a single point of view
2. be aware of the current reference conceptual space in use and
3. try to find and use the (conceptual) tools that most fit the problem

so that we can reduce both the cognitive load and the abstraction gap (if a platform exists or can be built).

Abstraction gap and innovation

By following a top-down approach, we constantly **evaluate the abstraction gap** in order to gain a better vision of what we need to *effectively* build the kind of systems we are considering and also as a way to *innovate.*

> *The entire history of software engineering is that of the rise in levels of abstraction.*
> *– Grady Booch*

Systemic approach for tackling complexity

As we are dealing with (complex) software systems, we believe that a **systemic approach** fits more than an algorithmic approach and gives us access to more powerful conceptual tools.

Reflection and continuous improvement

We think that the key for effectiveness lies in a continuous, **critical analysis** of the problems we encounter, the solutions we advance, and the approach we follow for mapping the former to the latter. A question we should always ask ourselves is: *what are we doing and why?*

We explicitly focus on **relationships between the conceptual tools used in a project and the project itself** (i.e., processes and artifacts).

We think that the use of the right tools can help minimizing the *accidental complexity*, thus allowing us to **focus on the essential complexity** of our systems.

Long-term vs. short term productivity

We believe that the **trade-off** *between long-term and short-term productivity* should be guided (at least) by the potential for reuse and the lifetime of the software artifacts produced. In this view, addressing the abstraction gap is key to grow and optimize in a sustainable manner.

Human factors in software projects

We are conscious that software development is a *human activity* and many issues arise from the coordination of several human players with different character, skillset, expectations, ...

> *Most software project problems are sociological, not technological. – From 'Peopleware' [2]*

This is why we need to follow a **disciplined approach** (enforced by rules, processes and tools), while being aware that people require comfort and space for expressing themselves through creativity, responsibility, trial and error, ...

Collaborative approach

Our work approach should be **cooperative** (rather than *competitive*).

The contracts and boundaries of software components should be defined up-front, so that the work can be parallelized across many people. In this view, the analysis is crucial (and, in particular, the logic architecture).

Moreover, the use of version control tools such as *git* is paramount to effectively carry out collaborative software development efforts. Again, tools have to be used with discernment and thought (cf. Git flow [6]).

Thus...

We think that a software development process built upon the principles outlined above can maximize the value of the effort spent during analysis and development, effectively produce software architectures that are resistant to requirements change, and even turn those changes into an opportunity to increase the organizational know-how.

3 Goals

We use this final project as an opportunity to actualize the vision.

Product and process

The main goal is to focus on the **relationships between the software production process and the final product**, with particular attention on the **artifacts produced in the phases of requirements and problem analysis**.

Project agility

Another main goal is to test **how much the project is resistant to changes in requirements**. What is the impact of required monotonic extensions on a product whose production is based on 'formal', 'technology independent' artifacts?

Reuse of domain knowledge

We are aware that the system we are building is just one of the countless applications that could be built in this domain. So, we generalise/abstract from the issues we encounter in a sustainable manner, so that we can **factor domain-knowledge out of the specific system**.

Incremental prototyping

By analyzing the requirements and the problem, we would like to produce a **working prototype by following an incremental (*piecemeal growth*) approach**.

Agile vs. model-driven

We have the goal to compare and keep the best from both agile and model-based approaches.

MDSD, software factories, meta-models, DSLs, and XText

We would like to factor the effort spent in filling the abstraction gap out into reusable artifacts. We think that an effective way to do so consists in building **custom meta-models** and associated **code generators**. XText helps us in this job by establishing a *link between models and code* (through the automatic meta-model inference from the DSL grammar).

In other words, we strive to incrementally build, around a *DSL*, a **software factory** – through a continuous collaboration between application designers and system designers – that encapsulates the knowledge to pass from "what" to "how".

With such an approach

- we make the conceptual space of our framework/DSL explicit
- we create a common language (cf. *ubiquitous language* in DDD) between (business experts and) application designers and system designers
- we can express not only design models, but even analysis models (e.g., declaratively stating what emerges from the problem)
- we have a direct, explicit, and automatic link from analysis/design elements to implementation elements

A DSL represents an organizational asset, the synthesis of the organizational know-how in a domain.

Moreover, we underline that there is even more benefit in *building* a DSL rather than *using* it. In fact, it is key for rapid prototyping and for systematically pointing out the abstraction gap by fostering a top-down approach.

While correlating what we do with the following concepts

- Models ($\Rightarrow$ Structure, Interaction, Behavior) as a way to capture the essential characteristics of what we talk about

- Software reuse as the key element to deal with bounded resources
  - Pattern vs. library vs. framework vs. middleware vs. DSL
- Technologies as specific ways to actualize designs but also as conceptual spaces (i.e., "representing" a paradigm)
- Logic architecture as the main result from analysis and possibly our main specification of the problem at hand
- Traceability (from requirements to architecture to code) as a way to support "informational consistency" within the project
- Specific vs. schematic vs. generic parts of software

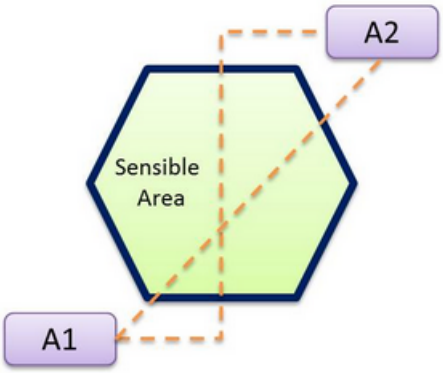
4 Requirements

The requirements are copied literally.

4.1 STEP 1

Design build a (prototype of a) software system that:

- allows a differential drive robot to move in autonomous way from a prefixed point A1 to a prefixed point A2 under a set of assumptions on the external environment (e.g. plain roof, no obstacle, etc)
- emits a signal named <RobotName>Enter when the robot enters into a SensibleArea delimited by a black line and a signal named <RobotName>Exit when the robot exits from the area
- allows the robot to react (as soon as possible) to a halt command sent by the user (via a remote console)



4.1.1 Non functional requriments at step1

.

The main goal is not only to satisfy the functional reuirements but it is mainly related to the relationships between the software production process and the final product, with particular attention on the artifacts produced in the phases of requiremt and problem analysis

4.2 STEP 2

After the development of this prototype, consider the possibility to enhance the functional capabilities of the robot, by allowing it :

- to perceive an obstacle within the SensibleArea amd, once the obstacle is detected:
  - to execute some alternative behavior (in term of moves)
  - (optionally) , to use a webcam to take a picture of the obstacle and to send it to some user command console and/or to use some device to emit sounds or vocal alerts

4.2.1 Non functional requriments at step1

The main goal is yo discuss how some change (monotonic extensions) of the requirements imact on a product whose production is based on 'formal', 'technology independent' artifacts.

5 Requirement analysis

5.1 Basic Requirements Analysis

We are aware of the **limitations of producing informal artifacts** (such as the glossary, use cases, etc). However, we consider them as intermediate steps that help us to reach a basic, high level **understanding** of the problem before we start drawing formal models from the informal requirements.

5.1.1 Elements to be understood/defined

First of all, the requirements are attentively read. As a rule of thumb, **nouns** point out the entities of the problem, **verbs** suggest what these entities should do, and **adverbs**/**adjectives** work as modifiers and suggest properties.

Step 1:

- Differential drive robot
- The robot should be able to move autonomously from prefixed point to prefixed point (in space)
- Assumptions on the external environment
- The robot emits a named signal when it enters/exits a sensible area delimited by a black line
- The robot should be able to react as soon as possible to a `halt` command sent by the user (via a remote console)

Step 2:

- The robot should be able to perceive an obstacle within the sensible area
- When the obstacle is detected, the robot should execute some alternate behavior (in term of moves)
- and use a webcam to take a picture and to send it to some user command console and/or to use some device to emit sounds or vocal alerts

The aforementioned elements have to be defined **formally** and **unambiguously**.

5.1.2 Questions & Answers

As **requirements are ambiguous, incomplete**, etc. we ask the customer some questions.

Questions to the customer:

- What means that the robot moves *autonomously*?
  $\Rightarrow$ *When the robot is turn on, it starts to move without being remotely controlled.*

- What's the starting point and the final point?
  ⇒ *The starting point is the place in which the robot is initially placed and the final point is where the robot stops.*
- What is a *signal*? What means it is *emitted – where*, *to who*?
- What is a *command*? Where do they originate from?
- What is, in general, a *sensible area*?
- . . . . . .

5.1.3 Use Cases

A very high-level, user-centered representation of functional requirements.

5.1.4 Scenarios

- **Scenario 1: Basic scenario**
  Actors
  - The robot
  - The user

  Description
  - The user monitors the robot. The robot starts to move autonomously. The user triggers the Halt signal through the console. The robot stops.

5.1.5 The system

As we are following a **systems approach**, first of all we look for the **subsystems**.

From the requirements, it follows that the system consists in the following subsystems:

- The robot
- The remote console of the user

As the console is remote, it follows that the system is **distributed**.

5.2 Domain Model

For us, a **model** is characterized by three key dimensions:

1. Structure
2. Interaction
3. Behavior

In order to express these models formally, we use (Java) interfaces and (JUnit) test plans. This is a convention used in our software house.

In our software house, we have already analyzed – in past projects – many of the concepts that emerge from requirement analysis, namely:

- Base robot
- Robot component
- Robot command
- Sensor
- Event

In other words, we exploit the domain models that have been built during the past projects in this very domain.

5.2.1 Base Robot

Structure

- A base robot is a *composed* entity
- It consists of a set of *robot components*, for example:
  - *Sensors*, which allow it to sense the world around it
  - *Actuators*, which allow it to act on the world around it

Interaction

- By perceiving and raising *events*
- An entity can ask a base robot to *execute* a *command*

Behavior

- Passive

See:

- `it.unibo.iot.executors.baseRobot.IBaseRobot`
- `it.unibo.iot.robotComponent.IRobotComponent`

5.2.2 Autonomous robot

An autonomous robot cannot be commanded/controlled externally.
With respect to a base robot, an autonomous robot cannot be requested to execute commands, that is, it has an autonomous behavior.

5.2.3 Commands

A *command* is a passive entity characterized by a (textual) representation.
A *timed command* consists in a command and a *duration*.
Note: this is **not** what, in the requirements, the term "command" refers to.

See:

- `it.unibo.iot.models.commands.ICommand`
- `it.unibo.iot.models.commands.baseRobot.IBaseRobotCommand`
- `it.unibo.robotUsage.interpreted.IRobotTimedCommand`

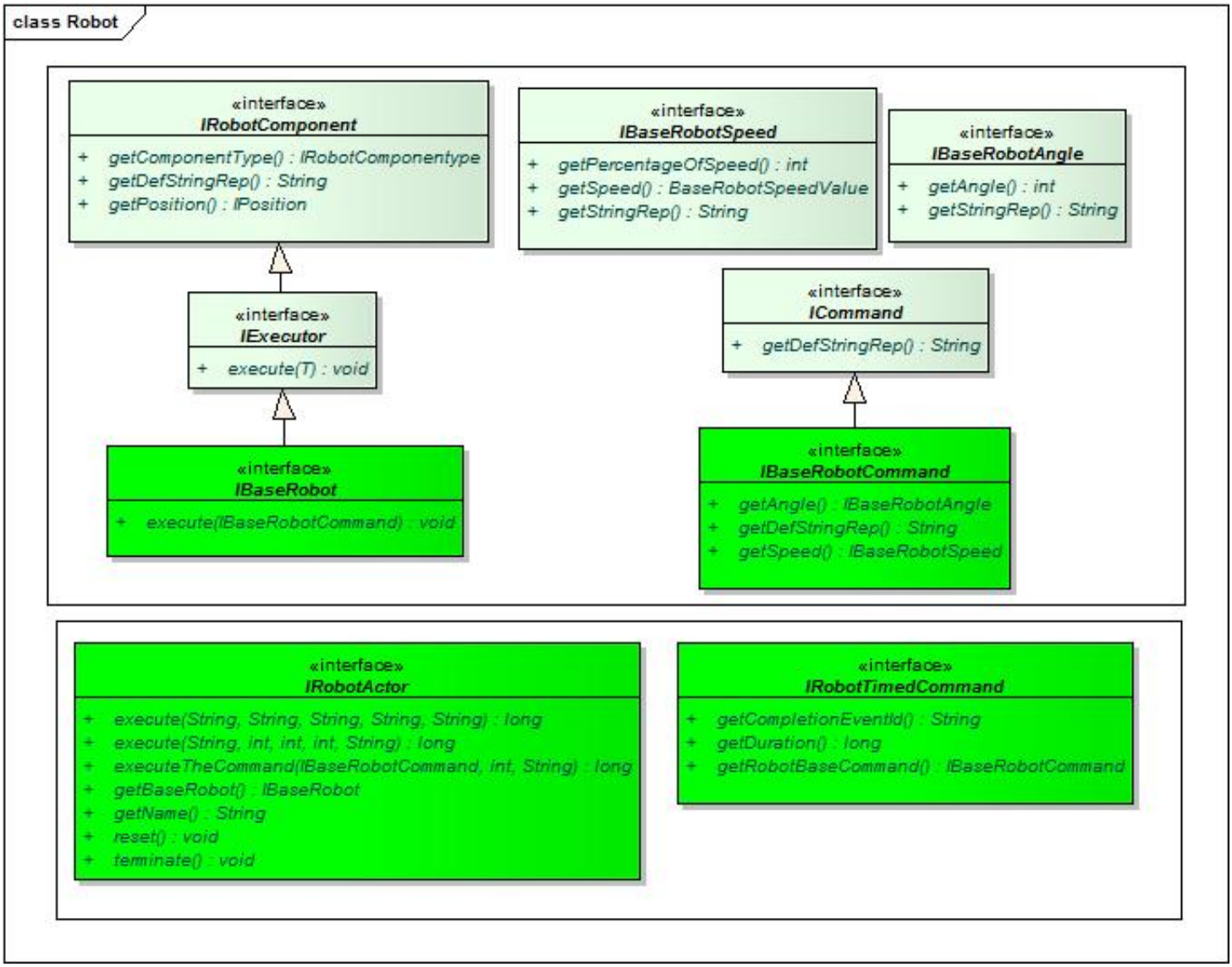
5.2.4 Sensors, Actuators

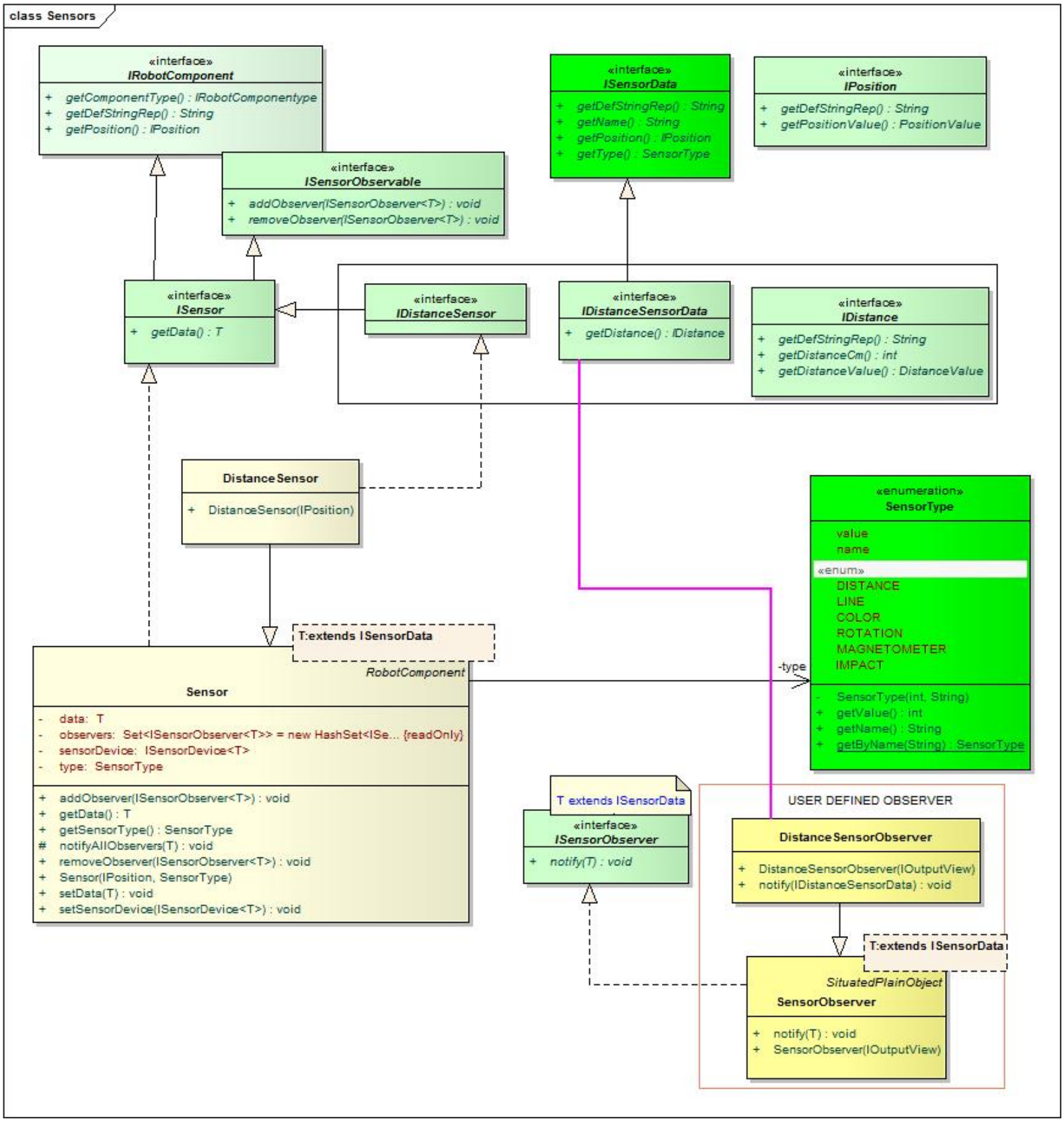
A *sensor* is an *observable* (cf. pattern Observer [3]) robot component which provides a *value* of a certain *type*.
Conversely, an *actuator* is an entity that can execute *commands* of a certain *type*.
Note that sensors and actuators have physical counterparts, but we focus on their software representatives.

- `it.unibo.iot.sensor.ISensor`
- `it.unibo.iot.executor.IExecutor`

NOTE: After problem analysis and in general after process iterations, other concepts are integrated into the domain model.

The following UML diagrams provide an excerpt, an integrated overview of the concepts that have been modelled in the past by our software house.




class Sensors
«interface»
IRobotComponent
+ getComponentType() : IRobotComponentype
+ getDefStringRep() : String
+ getPosition() : IPosition
«interface»
ISensorObservable
+ addObserver(ISensorObserver<T>) : void
+ removeObserver(ISensorObserver<T>) : void
«interface»
ISensorData
+ getDefStringRep() : String
+ getName() : String
+ getPosition() : IPosition
+ getType() : SensorType
«interface»
IPosition
+ getDefStringRep() : String
+ getPositionValue() : PositionValue
«interface»
ISensor
+ getData() : T
«interface»
IDistanceSensor
«interface»
IDistanceSensorData
+ getDistance() : IDistance
«interface»
IDistance
+ getDefStringRep() : String
+ getDistanceCm() : int
+ getDistanceValue() : DistanceValue
DistanceSensor
+ DistanceSensor(IPosition)
«enumeration»
SensorType
value
name
«enum»
DISTANCE
LINE
COLOR
ROTATION
MAGNETOMETER
IMPACT
- SensorType(int, String)
+ getValue() : int
+ getName() : String
+ getByName(String) : SensorType
T:extends ISensorData
RobotComponent
Sensor
- data: T
- observers: Set<ISensorObserver<T>> = new HashSet<ISe... {readOnly}
- sensorDevice: ISensorDevice<T>
- type: SensorType
+ addObserver(ISensorObserver<T>) : void
+ getData() : T
+ getSensorType() : SensorType
# notifyAllObservers(T) : void
+ removeObserver(ISensorObserver<T>) : void
+ Sensor(IPosition, SensorType)
+ setData(T) : void
+ setSensorDevice(ISensorDevice<T>) : void
-type
T extends ISensorData
«interface»
ISensorObserver
+ notify(T) : void
USER DEFINED OBSERVER
DistanceSensorObserver
+ DistanceSensorObserver(IOutputView)
+ notify(IDistanceSensorData) : void
T:extends ISensorData
SituatedPlainObject
SensorObserver
+ notify(T) : void
+ SensorObserver(IOutputView)

6 Problem analysis

6.1 Basic problem analysis

Here, we analyze the problem in a systematic manner. We strive to **express our ponderings through incontrovertible evidence**.
As an approach, we start off by considering the parts of the system, their behavior, and how they interact with each other. Then, we **zoom** into them, looking for problems at each level.

6.1.1 The application

From the requirement analysis, we know that the *system* is composed of **two parts**: a robot and a (remote) user console.

Usually, these two parts are on different computational nodes (i.e., the system is *distributed*) and must be able to **communicate over a network** (such as a wireless network).

The console must be able to emit information units, whereas the robot must be able to perceive them and react to them with some behavior. Note that we focus on the "information flow" but we do not want to over-commit towards a specific communication mechanism.

6.1.2 The robot

A robot consists of many *robot components* (such as sensors, actuators, ...). Each *type of sensor* (or actuator) is potentially different from one another in respect to its physical configuration, the values it provides, the rate of its working cycle, etc....

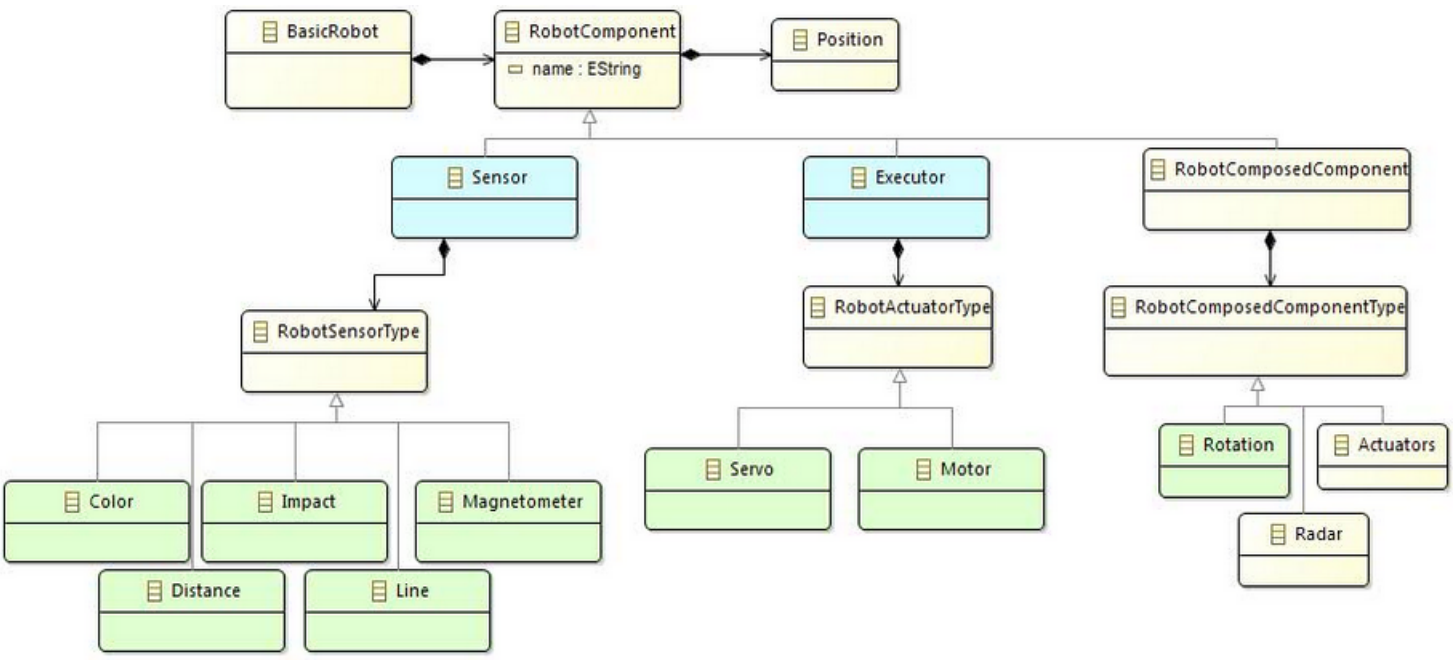


Figure 1: A base robot.

About the "*origin of the robot's behavior*", we note that a robot can be "executed" in (at least) two main *modes*:

- *avatar-mode*: the robot is controlled by an external entity (the *mind*)
- *planned-mode*: the robot is not externally controlled; instead, it executes a built-in behavior (see more about that later)

Applications require the robot to be able to carry on multiple activities (logically) at the same time, such as:

- Move in some direction (or, in general, command one or more actuators)
- Look for and react to sensor and application events
- Execute asynchronous activities (consider the case for making a video)
- ...

6.1.3 Physicality, situatedness, and environment

The **physical robot** is **situated** in the real world, can perceive it via sensors, and can act upon it via actuators. Similar considerations apply to the software robot.

We also note that the notion of **environment** is prominent in this domain.

6.1.4 Etherogeneity of robots

We also note that, in the real world, robots are usually **etherogenous** (built with different hardware/firmware/software components and configurations).

As a consequence, it's advisable to build the software system in a **technology-independent way** and to make the configuration procedure automatic.

6.1.5 Physical robot configuration/deployment

As robots can be physically built in different and etherogeneous ways, we should be able to easily deploy our robot software system to different physical embodiments.

6.1.6 Autonomy, agency, proactivity

Robots are oriented to *action*. Thus, autonomous robots are *agents*. As agents, they are *proactive* because the criterion for deciding what to do must be internal.

6.1.7 The environment is dynamically unknown, unpredictable

The sequence of elementary steps of the robot behavior *cannot be entirely defined offline*, because the environmental dynamics cannot be always anticipated.

6.1.8 Plans

We note that, in the literature, the problem of "accomplishment of prolonged, complex, and dynamically changing tasks in the real world" has been tackled by (among the others) **plan-based approaches** to robot control.

We also note that plans represent the application of the *divide-et-conquer* principle to the need of giving structure to (robot) behavior.

A *plan* is defined as a sequence of plan actions (moves).

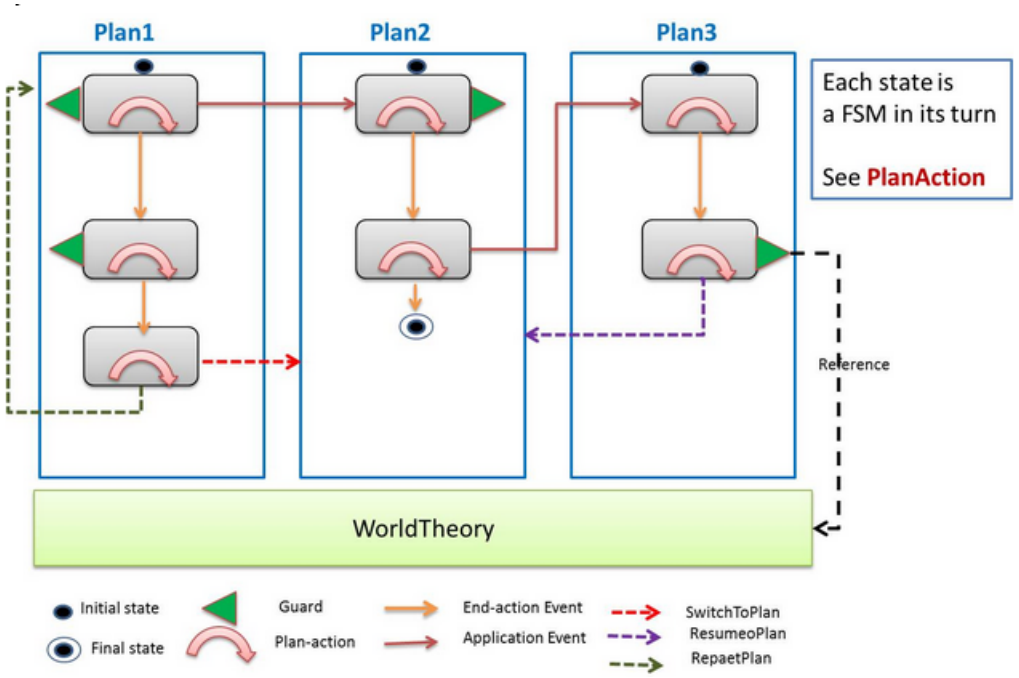


Figure 2: Plans.

6.1.9 Actions, moves

An *action* represents a (high-level) abstraction upon what a robot can do.

A *move* is a built-in action (e.g., move towards, take a photo, ...).

A *plan action* is a *guarded*, *timed* move that can be executed while also reacting to events.

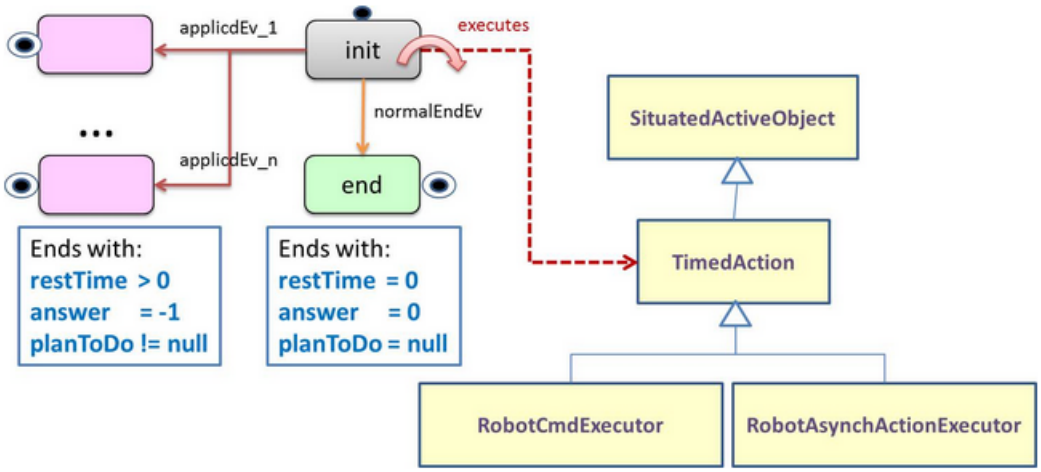


Figure 3: Plan actions.

Moreover, actions can be:

- *synchronous* – i.e., the action must be completed before executing the next move
- *asynchronous* – i.e., the action starts but the robot can immediately execute the next move

Actions could be *interruptible*.

6.1.10 Environment and representation

The importance of the environment and the dependency of the robot's behavior on the environmental context (cf. the notions of *situatedness* and *situated action*) might require the robot to (*implicitly* or *explicitly*) keep a representation of the *world* around it.

6.1.11 Reactions

The robot must be able to execute some behavior in response to events.

This must be possible even when some other behavior is being executed.

6.1.12 Reactive and proactive behavior

The robot must be able to balance *between proactivity and reactivity*.

6.1.13 State

The robot, in order to react accordingly while entering/exiting the sensible area, must have some notion of *state*.

6.1.14 Alternate behavior

Once the robot has reacted to an event by performing some alternate behavior, *what happens to the original behavior*?

Three possible options:

- The original behavior is *discarded*
- The original behavior is *resumed*
- The original behavior is *restarted*

6.1.15 Events

An *event* is a piece of information with no explicit recipient which is emitted by some *event source* and can be *perceived* by any entity that is interested to that *type of event*.

An entity may be interested to only certain *values* for an event.

An entity may be interested to many event types, with different *priorities*.

6.2 Logic architecture

The logic architecture represents the synthesis of the analysis effort; as such, it focuses on the "what" rather than on the "how". It comes from the problem and should not be influenced by the underlying technology.
Moreover, we consider the logic architecture as the key artifact (on which all the analysts agree on) for driving the next phases, namely design and implementation.
We would like it to express enough information (and so precisely, clearly, ...) to allow for work to be assigned and parallelized across many development people/units.

6.2.1 STEP 1

```
System robotSystemStep1 -java8 -http :

Context UserCtx ip[host="localhost" port=8090]{
        Actor QActorConsole;
}

Robot RobotStep1 bbb ip[host="localhost" port=8020] {

        Event haltCmd                       ;
        Event Line          when linefront  ;

        Task handleLineTask for Line        ;

        Plan alternate_plan
                println("Executing alternative plan... STOPPING");
                stop speed(0)

        Plan robot_behavior normal
                println("Let's start!!!");
                forward     speed(80) angle(0) time(6000)
                            react event haltCmd -> alternate_plan;
                right       speed(80) angle(0) time(400)
                            react event haltCmd -> alternate_plan;
                forward     speed(80) angle(0) time(4000)
                            react event haltCmd -> alternate_plan;
                backward    speed(80) angle(0) time(4000)
                            react event haltCmd -> alternate_plan

}
```

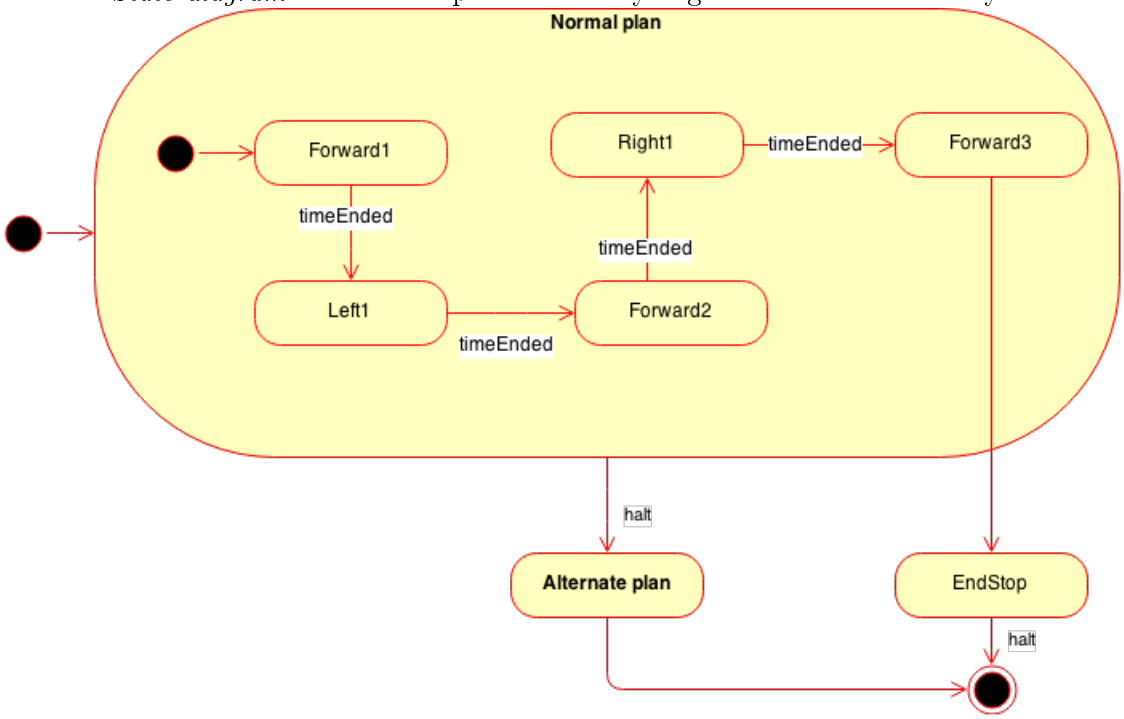
6.2.2 STEP 2

```
System robotSystemStep2 -java8 -http :

Context UserCtx ip[host="localhost" port=8090]{
        Actor QActorConsole;
}

Robot RobotStep2 bbb ip[host="localhost" port=8020] {

        Event haltCmd                               ;
        Event Line      when linefront              ;
        Event Distance  when distfront  val<10      ;
        Event endPhoto                              ;

        Task         handleLineTask for Line        ;
        EventHandler handlePhoto    for endPhoto    ;

        Action takePhoto    maxtime(2000)           ;

        Plan alternate_plan
                println("Executing alternative plan... STOPPING");
                stop speed(0)

        Plan take_picture_plan
                println("Executing ''Take picture'' plan... ");
                stop           speed(0);
                [insideSensibleArea] execAction takePhoto("group8foto.jpg")
```

```
27                                  answerEv endPhoto handledBy handlePhoto;
28              resumePlan
29
30      Plan robot_behavior normal
31              println("Let's start!!!");
32              forward     speed(80) angle(0) time(6000)
33                          react event    haltCmd -> alternate_plan
34                          or    event    Distance -> take_picture_plan;
35              right       speed(80) angle(0) time(400)
36                          react event    haltCmd -> alternate_plan
37                          or    event    Distance -> take_picture_plan;
38              forward     speed(80) angle(0) time(4000)
39                          react event    haltCmd -> alternate_plan
40                          or    event    Distance -> take_picture_plan;
41              backward    speed(80) angle(0) time(4000)
42                          react event    haltCmd -> alternate_plan
43                          or    event    Distance -> take_picture_plan
44  }
```

6.2.3 ~~Robot behavior - STEP 1~~

NOTE: the same information (actually, more detailed) has been expressed in the DDR model.
NOTE: this *State diagram* has to be kept continuously aligned with the current system.



6.3 Abstraction gap

Our **technology hypothesis** is given by

- our stack of custom frameworks,
- the Base Robot DDR DSL, and
- the High Robot DDR DSL

The problem and the practical issues encountered through the analysis phase pointed out a gap exists in many themes/areas, namely:

- **Model specification**: we have the need to build models of the system ($\Rightarrow$ S-I-C) that are *formal*, at *the right level of abstraction*, consistent, and easily modifiable.
- **General system concepts**: our systems approach should be supported by high-level concept such as "environment", "context", "situated entity", "named entity", ...
- **Rapid prototyping**: we have the goal to have analysis end up with a working prototype that allows us to receive early feedback
- **Communication and interaction**: we have the need to express how our system entities interact in a high-level, technology-independent manner
- **Infrastructure configuration and setup**: we don't want to be concerned in the details of the system setup and configuration
- **Robot configuration**: we want to be able to easily specify the logical and physical configuration of a robot
- **Robot behavior and interaction**: we want to be able to specify what a robot should do, perceive, react ... in a technology-independent manner

After repeated interactions between application designers and system designers, our frameworks/DSLs have evolved to minimize the aformentioned gap between the problem and the execution platform.

6.4 Risk analysis

In addition to

- the possibility of changes in functional requirements (i.e., what the robot system should do)
- the possibility of changes in non-functional requirements (i.e., what properties the robot system should exhibit)
- the technological evolution (both hardware and software) in this very field

which are more facts than uncertain events, we need to monitor and control the following risks.

6.4.1 Performance and efficiency

Even though we do not have real-time requirements, it's important to realize that design decisions might seriously impact on performance, possibly resulting in significant delays (e.g., in the execution of reactions to events).

6.4.2 Distribution issues

From distribution, many issues arise:

- Unreliability of the network
- Latency

- Changes in topology
- Security
- ...

These potential problems need to be taken into account if the robot is intended to be a *dependable* system (i.e., available, reliable, safe, maintainable).

6.4.3 Etherogeneity

In the real world, these kinds of systems may be implemented using many different (hardware and software) technologies.

As a consequence, integration and interoperability issues may occur.

6.4.4 Testing and debugging

Because of the many layers – from the physical layer to the application layer – and the many components involved, it might be difficult to trace and locate faults in case of errors.

7 Work plan

This section is related to the **product backlog** notion of the Scrum process framework.

The team consists in 3 team members with cross-functional skills (analysis, testing, implementation).

As we are following an iterative process, the work plan consists in iterations with the following activities (not all of them necessarily executed at each cycle):

1. Individual requirement analysis
2. Requirement analysis: discussion and results
3. Domain modelling
4. Individual problem analysis
5. Problem analysis: discussion and results
6. Logic architecture refinement
7. Risk analysis and work plan refinement
8. Abstraction gap evaluation
9. Collaboration with system designers
10. Implementation of application-specific parts
    - ... (see next)
11. System integration testing
12. User-acceptance testing

These activities can be broke down into sub-tasks or address specific parts of the artifacts involved. For example, as in this case study the requirements are split into two parts, it has been natural to split the analysis activities accordingly.

The work is parallelized when possible, however the results from analysis have to be agreed on by all the team members.

Supposing we had to design/implement the system without the DSL (possibly before building the DSL), the (high-level) work plan would have been something as:

- **Base robot**: Design/implement a software system with a base robot that merely executes commands
- **Base robot + console**: Design/implement a software system with a base robot that merely executes commands sent by a remote console
- **Get sensor data**: Design/implement a software system that shows data from the robot's sensors
- **Autonomous, planned robot**: Design/implement a software system with an autonomous robot that executes its built-in plan
- **Reactive/proactive robot**: Design/implement a software system with an autonomous robot that, while it executes its built-in plan, is able to react to sensor or application events
- ...

Note how the approach to development is incremental.

8 Design

The design of the system is encaspulated and enforced by the software factory of the DDR DSL (i.e., the code generators).

Please consult the DDR documentation (and ultimately the code generators) to have the up-to-date, authoritative description of this aspect.

At the current version of the software factory, the following **architectural style** is enforced:

- Structural organization: *contexts* populated by *qactors*
  `robotsystemstep1.pl`

```
1 context(robotstep1, "localhost",  "TCP", "8020" ).
2 context(userctx, "localhost",  "TCP", "8090" ).
3
4 qactor( qactorconsole, userctx ).
5 qactor( robotActor, robotstep1 ).
```

- Interaction through *messages*, *events*, *streams of events*

9 Implementation

The **schematic part** of the implementation of the system is generated from the logic architecture by the software factory, whereas the **specific part** has to be implemented by the application designers.

9.1 To be implemented

The following pieces have to be implemented.

9.1.1 Step1

- The event handler for `Line` events
- The QActor for the remote console

9.1.2 Step2

In addition to Step1:

- The event handler for `Distance` events

9.2 On the robot implementation

The robot subsystem is a context (`ActorContext`). Its setup consists in:

- Setup of the robot `QActor`
  - Loading of the robot configuration (`robotConfig.properties`, `hardwareConfiguration.properties`, `iotRobot.properties`)
  - Configuration of the base robot and its components
  - Launch of the robot
- Creation of the event handlers / tasks
- Loading of `WorldTheory.pl` as a TuProlog theory
- Setup of the USB connection and/or the HTTP server
- Setup of the observers for sensors and registration of the subscribers
- Setup of the Nashorn JS Engine, loading of the JavaScript robot interpreter, and execution of the plans

10 Testing

The testing activities are usually carried out in the following order:

1. **System integration testing: robot mock, concentrated system**
   ⇒ *This step allows us to assess the application logic without incurring in deployment/network/technology issues.*
2. **System integration testing: robot mock, distributed system**
   ⇒ *Let's add one big concern at a time, here distribution.*
3. **System integration testing: physical robot**
   ⇒ *Deployment testing*
4. **User acceptance testing** (simulated)
   ⇒ *Is the software right? Is what the user/customer expects?*

The **robot mock** is extremely useful because it allows to test the system without the need to deploy it into a physical robot (which demands resources to be built and may require significant effort for deployment and configuration activities).

11 Deployment

The requirements do not specify any particular deployment configuration.

An example configuration is the following one:

- ROBOT: A JAR package for the RaspberryPi subsystem that commands the physical robot
- USER CONSOLE: A JAR package for the user console

The logic architecture expressed using the HighRobot DDR DSL includes deployment details, but they could also be changed in the generated artifact `robotsystem.pl` which contains the declarations of the contexts (subsystems) of the system.

12 Discussion

12.1 UML vs. Java interfaces + test plans vs. Metamodelling

As outlined in the Section *Vision*, we are for using the right tool at the right time and for the right purpose.

When we are modelling, we want to come up with models that are unambiguous, formal, at the right level of abstraction, human-understandable. These models should also be kept consistent with related artifacts with little pain.

When using **UML**, we encounter many issues:

- UML models can be read by tools (i.e., the syntax of UML is well-defined), but the semantics of UML is **ambiguous** to some extent (it suffices to say that UML elements are described in natural language – inherently ambiguous)
- As a result, the **semantics** of UML is that given by a certain UML tool – but we are not guaranteed that the representations of our models are interoperable among different tools, nor against different versions of the same tool ($\Rightarrow$ risk of *vendor lock-in* and *version lock-in*)
- While possible, it is not (technically) easy to keep UML models *in-synch* with source code
- The **generation of code** from UML diagrams presents many issues as well. UML diagrams are useful when they abstract implementation details out of the actual code. On the other hand, a diagram which is easy to understand turns out to be not very useful for code generation; moreover, in this case, the code itself does the job nicely.
- Not everything can be *expressed* with UML. We, throughout the ISI course, have found ourselves in extending UML with ad-hoc notations. UML is quite related to the object-oriented paradigm, but we might find **expressivity** issues when we are out the OOP realm.

However, UML still represents a useful, standard communication tool in many situations.

The use of **Java interfaces** for modelling is an approach for doing analysis while actually writing code at the same time. For us, a Java interface represents a model of a domain concept. However, while formal, they are merely syntax; their semantics is just in our minds (implicit in our convention). To better specify the semantics of the concepts modelled through interfaces, we write **test plans** (which will also work as (regression) tests once implementation classes are built).

As long as the concepts can be easily expressed through the Java metamodel, this approach works well. Howerver, what can we do in case of expressivity issues, unappropriate level of abstraction, modelling mismatch, ...? We also note that these analysis artifacts are not very technology-agnostic..

Another approach consists in developing a **custom meta-model** that gives names to (domain) concepts and defines the abstract syntax of a (custom, technology-independent) modelling language. Then, it is possible to specify (custom) semantics by defining *model transformations*.

12.2 Meta-model as a living artifact

We note that the meta-model has an **evolving nature**.

In fact, as long as we get a better comprehension of the problems and the applications generate *forces* that stress the *expressive power* of the metamodel, we might found ourselves in the need of refactoring/extending/changing the metamodel.

12.3 A DSL is a project!

The DSL should be considered a software project on its own. Requirements come from applications and the feedback from application designers. The requirements and the problem have to be analyzed: what has to be expressed with the DSL? what levels of abstraction are required? are there trade-offs between expressivity and performance? ... The language abstract and concrete syntaxes have to be properly designed as well as the code generators. There are multiple (often conflicting) design dimensions (such as expressivity, domain coverage, semantics, separation of concerns, completeness for execution, language modularization, syntax [5]) to be taken into account.

12.4 Agile (Scrum, ...) vs. MDSD

We essentially agree with the principles behind the Agile Manifesto[4] and its motto

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

however we note that they are essentially "defining" principles that say little about their actualization. Agile tends to be *descriptive* rather than *prescriptive* (with exceptions, cf. XP), but this results in a gap from theory to execution. As a consequence, Agile is often misunderstood/misapplied.

Concerning the **Scrum** process framework, we recognizes that it may help the projects by forcing people to perform short iterations (cf. sprints, early feedback) and take good habits (cf. the *product backlog* as the unique list of things to do, *retrospectives*, fostering *incremental development*, avoid over-engineering by limiting the work-in-progress in the *sprint backlog*, the *scrum master* as a facilitator ...). However, there are many other degrees of freedom that might make the projects fail (c.f., self-organizing teams).

We do not see any significant contradiction between Agile and MDSD that would make them irremediably incompatible.

Historically, agile has always been for building the real system as soon as possible in order to get early feedback, while at the same time avoiding over-specification. Whereas there is the myth that model-driven approaches must be necessarily heavyweight.

We think that a fruitful link between Agile and MDSD can exist because:

- Coding is not fundamentally different from modelling
- Models can be developed iteratively and incrementally
- The distance between models and the final system can be reduced via code generators

12.5 On the relationships between the DSL and the technological infrastructure

12.5.1 Many layers

The system we have built can be seen as a *layered* system.

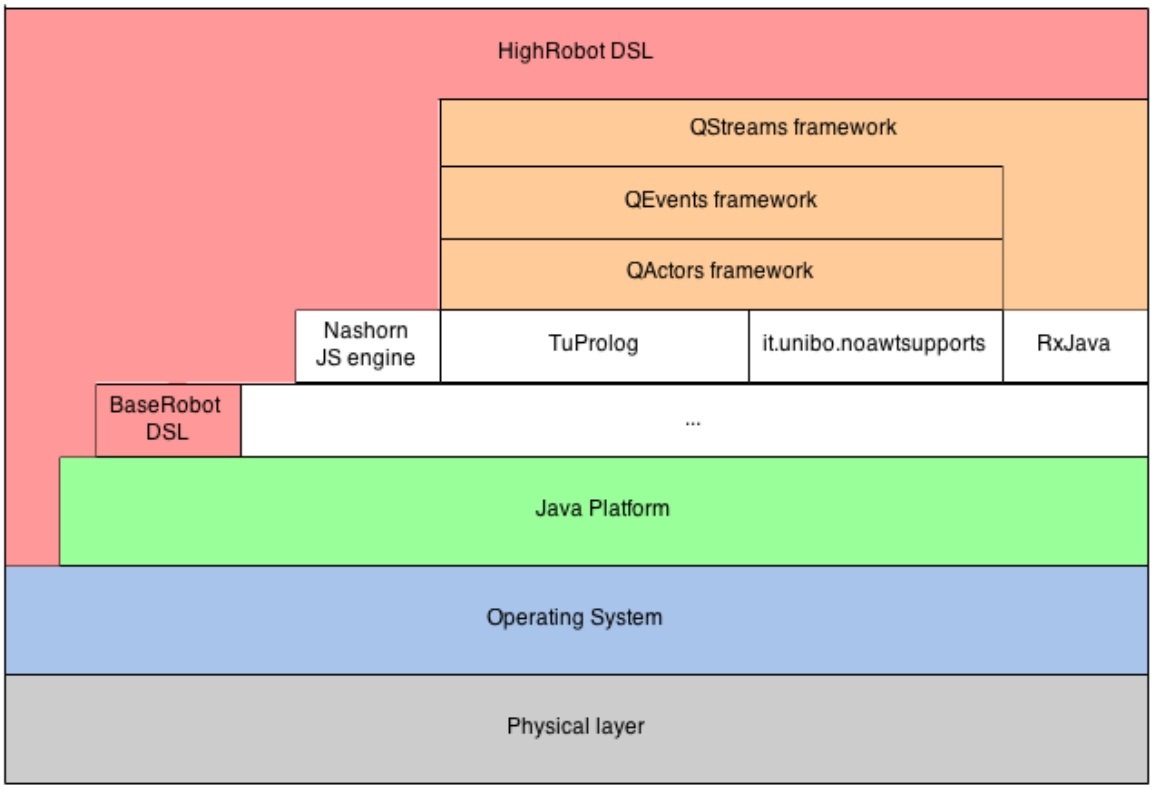


Figure 4: Layers.

The *layered* approach (or, the Layers architectural pattern [1]) is an application of the *divide-et-impera* principle and is good for reducing complexity and fostering simplicity and order. In fact, the dependencies of a single piece (layer) are reduced to just the layers directly below.

In our experience, we have seen how this approach **supports a sustainable, gradual, step-by-step reduction of the abstraction gap** and fosters a modularization that turns out to be very good for reuse.

12.5.2 Integrating top-down and bottom-up approaches: a hybrid approach

We have experienced the advantages and the *theoretical* reason behind following a top-down approach.

However, the *pragmatical* issues encountered (during implementation) often require to cope with technological details in order to make things work or to get a sense of some aspects of the problem at hand. Moreover, for example, it is possible that theoretically irreproachable solutions could result in inappropriate actualizations (e.g., because of the limitations and constraints of the available underlying technology).

Thus, **a bottom-up approach can be combined to a general top-down attitude in order to get a quicker feedback on technological issues, identify unforeseen problems**

that might emerge at higher levels, or identify relevant features not included in the models.

Our idea, when there are mutual influences among upper and lower layers, is not to be dogmatic or biased, but to be flexible and able to evaluate the pros and cons of each decision/alternative.

This discussion recalls to the relationship *between theory and practice*. They are correlated; a tension exists between them. We suggest to be balanced: to make practice with theoretical foundations, and to make theory with cognition/feedback from practice.

12.5.3 Abstraction, issues and trade-offs

Abstraction is not free. It has a cost, and when abstractions are designed, that cost has to be considered. We remember that engineering is also about making trade-offs.

Models should specify enough details to be useful. In some cases, implementation details should be provided (anticipated) in order to generate architectures that fit the concrete, specific application scenario (e.g., for performance or security).

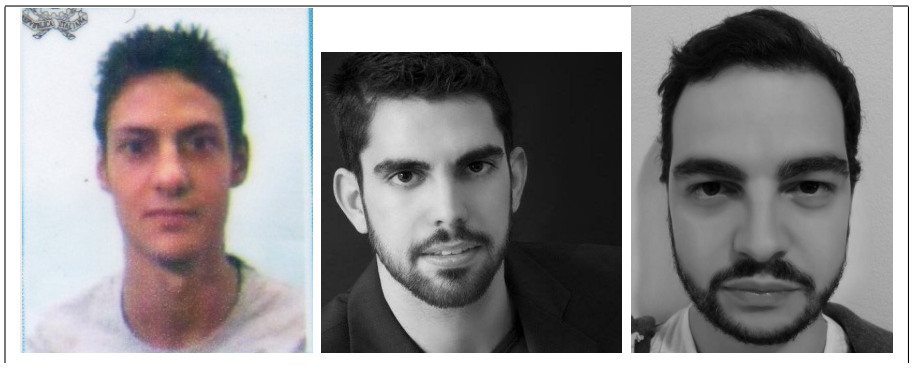
12.5.4 More on the DSL and layers

The robot is a QActor because it is active (and autonomous) and lives in a distributed setting where message-passing is a convenient communication style. Why have we chosen to make it a QActor rather than a lower-level (e.g., active objects) or higher-level (e.g., agents) abstraction? For the first case, we would have experienced a significant effort in trying to fill the abstraction gap; about the second option, we note that agent abstractions typically include higher-level characteristics that are unneccessary or too complex at the present time, not (yet) motivated by analysis.

About the contexts (our subsystem abstraction), we note that they have been included as part of the HighRobot DSL (up to version 1.1.8, with an '`ActorContext`' keyword) and directly map to QActors' `ActorContexts`. The DSL also supports the declaration of actors within the declared contexts. This is an example of lower-level concepts that are pushed up to the DSL level not only for simplicity and rapid prototyping needs but also because of information needs in the code generation phase.

Similarly, the HighRobot DSL also includes keywords such as `-http` and `-usb` to activate an HTTP web server and a USB connection, respectively. This is another example (concerning the theme of DSL and software factory design) of technology details that are anticipated for practical issues.

13 Information about the authors



References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK, 1996.

[2] T. DeMarco and L. Timothy. *peopleware - productive projects and teams.* 1987.

[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Computing Series. Addison-Wesley Professional, november 1994.

[4] A. Manifesto. Agile Manifesto home page. http://www.agilemanifesto.org/.

[5] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.

[6] G. Workflow. Gitflow Workflow. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.