UNIVERSITÀ DEGLI STUDI DI BRESCIA
FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
TESI DI LAUREA SPECIALISTICA

ANALISI DI FATTIBILITÀ ED IMPLEMENTAZIONE DI UN SISTEMA DI ROBOTICA COGNITIVA PER COMPITI DI NAVIGAZIONE

(Analysis of preconditions and implementation of a Cognitive Robotic System for navigation tasks)

Relatore:

Ch.mo Prof. Riccardo Cassinis

Correlatore:

Ch.mo Prof. Marco Ragni

Laureando:
Francesco Bonfadelli
Matricola 83174

Anno Accademico 2012-2013

Contents

List of Figures

1. Introduction

1.1. Organization of the document

1.2. The Objective

2. State Of The Art

This chapter describes the main working instrument used for this work: the cognitive architecture *ACT-R* and the computer vision library *OpenCV*.

2.1. ACT-R

ACT-R, that stands for *Adaptive Control of Thought-Rational*, is a cognitive architecture, i.e. a computer implementation of a theory about human cognition. As such, it models the structure and behavior of the human brain trying to explain how all the components of the brain work together and form the human mind.

The following sections describe the theory on which ACT-R relies, how the theory is implemented in the framework and the concept of *model*.

2.1.1. The Theory Behind ACT-R

The following section describes two fundamental assumptions on which ACT-R relies, the *Unified Theory About Cognition* and the classification of the human memory in *declarative* and *procedural*. Both of these assumptions are important because they determinate the structure of the framework.

Unified Theory About Cognition

ACT-R implements the homonym theory developed by John Robert Anderson, professor of psychology and computer science at Carnegie Mellon University. Such theory tries to explain the overall behavior of the human mind through connections between its well-defined components that, combined together, form an integrated system. The following quote, that explains the meaning of integrated system, comes from Allen Newell, the man who inspired Anderson in creating ACT-R theory [ABB+04].

> A single system (mind) produces all aspects of behavior. It is one mind that minds them all. Even if the mind has parts, modules, components, or whatever, they all mesh together to produce behavior. Any bit of behavior has

> causal tendrils that extend back through large parts of the total cognitive system before grounding in the environmental situation of some earlier times. If a theory covers only one part or component, it flirts with trouble from the start. It goes without saying that there are dissociations, independencies, impenetrabilities, and modularities. These all help to break the web of each bit of behavior being shaped by an unlimited set of antecedents. So they are important to understand and help to make that theory simple enough to use. But they don't remove the necessity of a theory that provides the total picture and explains the role of the parts and why they exist [New94, p.17-18].

Declarative memory and procedural memory

In psychology, *memory* is defined as the processes by which information is encoded, stored and retrieved [BEAA09].

ACT-R's most important assumption about knowledge is based on Anderson's theory about memory. Anderson divides memory into *declarative* and *procedural* [ABB+04].

Declarative memory refers to all the information that can be consciously recalled. This kind of knowledge comprehends facts and notions that human beings explicitly know. To call back this kind of information, there must be a conscious process by the human being. For this reason, this kind of memory is also called *explicit*.

In contrast, procedural memory refers to all that notions or skills that human beings have but which they learnt in an implicit way. Examples of this knowledge are driving, reading and writing. In this case, in order to call back this kind of information, the human being does not need a conscious process. That is why this kind of memory is also called *implicit* [And76].

When a person starts learning typewriting, for example, an attempt he can make in the beginning is trying to memorize the layout of the keyboard. The aware knowledge of all the positions of the keys is the declarative memory. After having become a skilled typewriter, the same person will type faster putting his fingers on the right keys and pushing them in the correct order, without thinking anymore about the positions of the keys on the keyboard. Moreover, if someone asks him where the position of a certain character is on the keyboard, he will probably answer that he can not say it without looking at it. This is because, now, for this task he is using his procedural memory [And93].

2.1.2. The Architecture

This section, after having defined *chunks* and *productions*, the building blocks of ACT-R structure, presents the architecture of the framework.

Chunks and productions

In ACT-R, declarative memory is represented by structures, called *chunks*, and procedural memory by rules, called *productions*. Chunks and productions are the basic building blocks of ACT-R *models*. The concept of model will be described in 2.1.3.

The *chunks* are data structures which are defined by their *type* and their *attribute list*. This is a tuple of pairs, each of which is made up by a fixed part and a variable part. The fixed part is the *name* of the attribute and is called *slot*. The variable part is the *value* of the attribute. Each chunk has also a *name* but it is not considered to be a part of the chunk itself, as it does not exist in ACT-R theory. It is used only for convenience to reference the specific chunk when writing models. The chunk-types can be organized into hierarchies.

The *productions* are ACT-R equivalent of functions. They define sequences of actions and can be fired only if a set of preconditions is satisfied. They can be represented as *if-then* rules, where the *if-part* is a set of conditions that must be true for the production to apply and the *then-part* is the action of the production and consists of the operations the model should perform when the production is selected and used.

In general there could be some conflicts between productions. This happens when preconditions of two or more productions are satisfied at the same time. In these cases the production to be fired is the one with the highest *utility value*, a numeric quantity which gives a priority measure. It can be set a priori by the modeler or learnt while the model is running. The latter, together with the fact that in the calculation of this value the probability of reaching the goal and the time estimate are included, constitutes the basis of the learning mechanism embedded in ACT-R: the more successfully a production is, the more its estimated probability and utility grow, increasing the probability for that production to be selected again [Bota].

Organization of the Modules

All the activities carried out by the human brain, like talking or moving, are performed by neurons located close together in a well defined and limited area of the cortex. Trying to imitate this "architecture", ACT-R's framework is structured in different *modules*, each of which represents one specific function of the human brain.

Figure 2.1 shows the modular structure of ACT-R. In the picture you can see two groups of modules, separated by the *procedural module*.

The first group comprehends *visual*, *aural*, *manual* and *vocal modules*. These let the model interact with the environment. The *visual module* is responsible for recognizing objects in the visual scene and shifting the focus to them. Similarly, the *aural module* identifies sounds and moves the attention to them. The *manual module* can move the virtual hands and perform actions like pressing the key on a keyboard or moving the mouse while the *vocal module* controls the virtual voice.

The other group comprehends *goal module*, *imaginal module* and *declarative module*. These represent the internal information of the model. The *goal module* provides the system with the structure of the goal of the task, defined as a chunk. The *imaginal module* has to contain and update the current context relevant to the current task. The *declarative module* provides the model with a declarative memory, thus it stores the declarative chunks generated by the model and provides a mechanism for retrieving them.

Finally, the *procedural module* is responsible of the communication and the coordination of all the other modules.

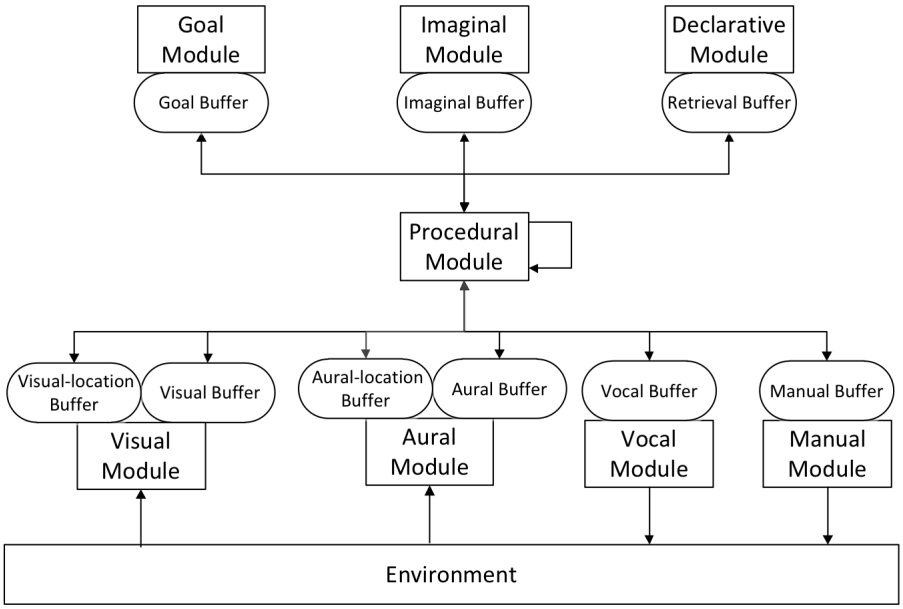

Figure 2.1.: Structure of ACT-R.

All the modules are independent of each other, they do not share variables or information. They can communicate with each other through *buffers*, which represent the interface of a module towards the others. A module can have no buffers as well as one or more than one. The communication consists in exchanging chunks. Each module can

read chunks from every buffer but it can make changes only to the chunks in its own buffers. Moreover each buffer can hold one chunk at a time.

Although modules usually work in a parallel way, their interactions can be only serial. There are two reasons for this limitation: the first one is that the structure of the buffers can hold only one chunk at a time and the second one is that only one production can be fired at a time [Bota].

2.1.3. ACT-R Models

ACT-R is able to perform several task-independent reasoning. To adapt these reasoning to a single task it is necessary an additional layer called *model*.

Each model contains the specific modelers' assumptions about the task within ACT-R view of cognition. That knowledge is expressed through productions that, when the model is running, interact with the modules, querying them and reading their buffers.

Running a model produces a series of atomic cognitive operations which step-by-step leads to the solution of the task. Each operation is associated with quantitative and qualitative measures like the correctness of the goal and the time necessary to complete the operation. This fact gives the model the possibility to predict the sequence of cognitive actions produced by human beings when they try to solve the same task. Comparisons with human performances can be useful to evaluate the quality of the models [Sea02].

ACT-R is written in Lisp and provides its own lisp-like language for the modelers to write the models.

2.2. OpenCV

OpenCV, an abbreviation that stands for *Open Source Computer Vision*, is a computer vision library that was originally developed by Intel and, later on, by Willow Garage. It is a cross-platform library and can run under Linux, Windows and Mac OS X. It is released under a BSD license, thus it is free and open source. In the beginning it was developed in C and C++ and afterwards it was expanded by the addition of interfaces for other languages as, for example, Java, Python, Ruby and Matlab.

OpenCV is designed for computational efficiency and with a strong focus on real-time applications. On Intel architectures there is the possibility to further optimize the library thanks to Intel's *IPP* library. IPP library, that stands for *Integrated Performance Primitives*, contains a series of low-level optimized routines in many different algorithmic areas. If this library is installed, OpenCV uses it automatically at runtime.

OpenCV offers a simple to use infrastructure that helps programmers create quite complicated vision applications in an acceptable time. The version 2.4 has more than 2500 algorithms that cover many areas of vision. The library has been used in many applications as, for example, mine inspection and robotics. It also contains *MLL*, a full and general purpose *Machine Learning Library*, which provides functions for pattern recognition and clustering [BK08, Ope12c].

The following sections contain an introduction to computer vision, a brief history of OpenCV library, a description of its main features and an overview of its architecture. For more information about computer vision see texts by Trucco [TV98] for a simple introduction, Forsyth [FP11] as a comprehensive reference, and Hartley [HZ03] and Faugeras [Fau93] for how 3D vision really works.

2.2.1. Introduction to Computer Vision

Computer vision is the transformation of image or video data into a new representation. Such representation can be either another image, for example a gray-scale version of the original one, or even a decision based on the information extracted from the input data. Often computer vision algorithms use contextual information in order to simplify the problem.

People not expert in computer vision may mislead the complexity of its tasks. Often, in fact, even a very simple operation for the human brain, like recognizing an animal in a picture, represents a very complex task for a machine. Human visual system is very complex. It divides the visual signals into many channels, each of which brings a different kind of information to the brain. The brain uses an attention system which identifies important parts of the image and focuses on them, excluding the analysis of the remaining parts. Moreover, further information coming from all the other senses and the experience acquired during the whole life of the person completes the analysis of the image, in a way still not completely understood. All these factors create the visual perception of human beings.

In a machine vision system, instead, when the computer receives an image or a video from a camera, it receives simply a grid of numbers. In most of cases, there are no mechanisms for pattern recognition, automatic control of aperture and focus of the camera nor cross-associations with other senses or experience. Figure 2.2 explains this concept. The machine does not recognize automatically the side mirror of the car, it sees just a grid of numbers. A goal of computer vision is to turn the grid that describes this particular object into the object itself.

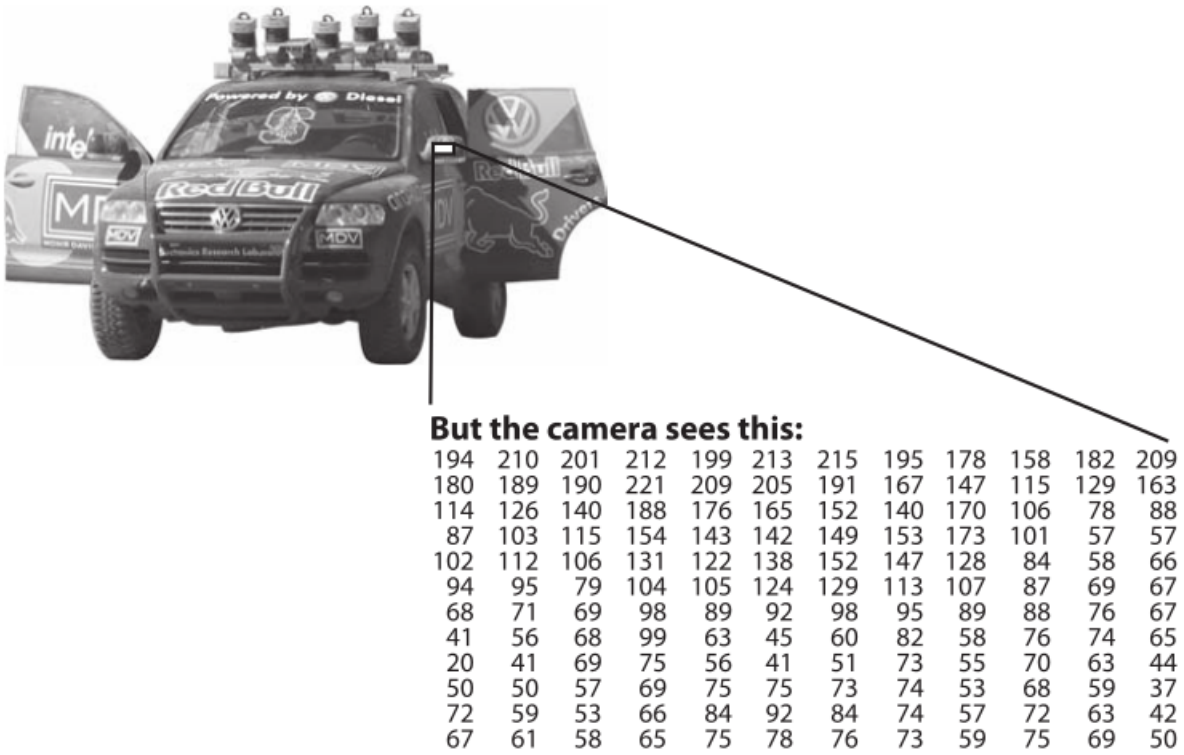

Figure 2.2.: Representation of an image in a computer vision system.

Figure 2.3 shows another aspect that makes computer vision tasks so difficult to solve. An image is a 2D description of a 3D world. The problem in the way it is posed is impossible to solve. There are infinite possible ways to reconstruct a 3D world from a 2D image. As shown in the picture, the solution changes with the point of view of the camera. This is formally an ill-posed problem.

Moreover, generally speaking, data is affected by noise. The word noise comprehends all the possible variations in the world, like weather, lightning, reflections and movements, and all the imperfections affecting the input mechanisms, like errors of the lens or of the sensors.

Some causes of noise are easy to remove. The errors caused by the lens, for example, can be mathematically described by polynomial functions, thus it is quite simple to correct them. The environment variations, instead, are not well known a priori, hence are more difficult to delete. Typically, this category of noise is dealt with statistical methods, that generally do not remove it completely but attenuate it. Say, for example, to have many different pictures of the same visual scene, every one taken in different conditions of weather, lightning, reflections and so on. In this case the noise is considered, statistically

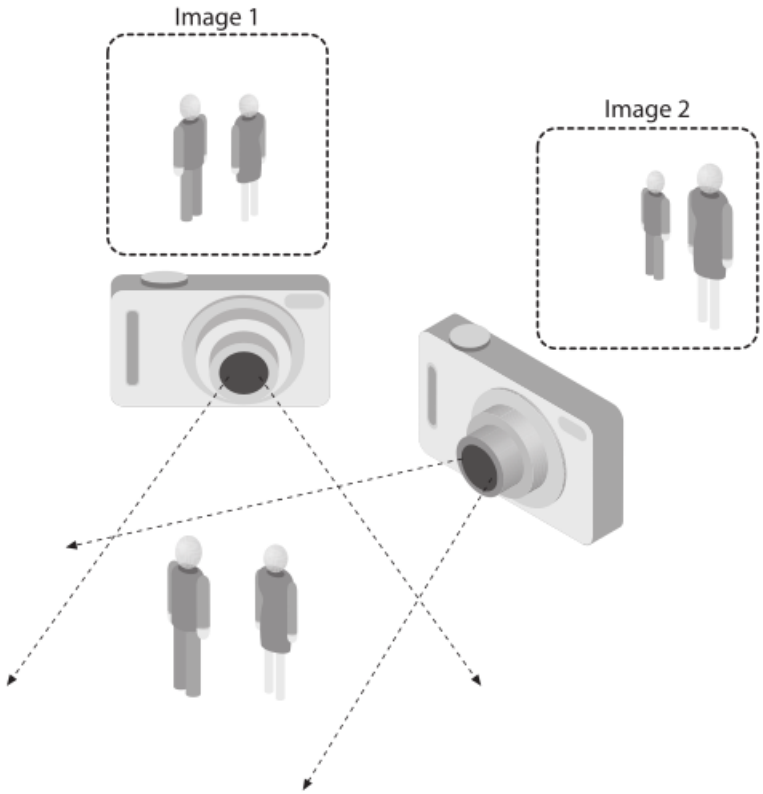

Figure 2.3.: How the 2d representation changes with the viewpoint

speaking, a white noise. A statistical approach suggests to make the average of all the pictures. In this way, the features of the image remain, while the noise is reduced by the average operation.

A common way to simplify the problem of computer vision is to add contextual knowledge to the solution. Consider the example of a mobile robot which has to find and pick up staplers in a building. The robot can use the fact that the staplers are always on desks. This fact gives the robot an implicit information about the size and the position of the object, that brings it to exclude false "recognitions". Moreover, if all the desks have rectangular shapes, are made of wood and have the same color, the computer vision system can use this additional information to exclude other false staplers, increasing the accuracy of the research. As general rule: *"the more constrained a computer vision context is, the more we can rely on those constraints to simplify the problem and the more reliable our final solution will be"* [BK08, 5]. Of course, in this way, the solution is not general purpose any more.

Machine learning techniques allow to extract variables that model contextual information from a limited number of training cases and using them to solve other instances of

the modeled problem. Differently from the previous example, such variables are not set a priory by the modeler but are learnt from a series of trials. For this reason they can be hidden parameters or may even not represent physical quantities and remain mathematical variables [BK08].

2.2.2. OpenCV History

The OpenCV Project starts in 1999 as an Intel Research initiative aimed to improve CPU intensive applications as a part of projects including real-time ray tracing, which in computer graphics is a particular technique for generating images, and 3D display walls. The early goals of the project are: developing optimized code for basic vision infrastructure, spreading this infrastructure to developers and making it portable and available for free, using a license that allows the developers to create both commercial and free applications.

The first alpha version is released to the public in 2000, followed by five beta versions between 2001 and 2005, which lead to version 1.0 in 2006. In 2008, the technology incubator Willow Garage begins supporting the project and, in the same year, version 1.1 is released.

In October 2009, OpenCV 2.0 is released. It includes many improvements, such as a better C++ interface, more programming patterns, new functions and an optimization for multi-core architectures. According to the current OpenCV release plan, a new version of the library is delivered on a six-months basis. [Ope12a].

2.2.3. Main Features

OpenCV offers a wide range of possibilities. First of all, it provides an easy way to manage image and video data types. It also offers functions to load, copy, edit, convert and store images and a basic graphical user interface that allows the developers to handle keyboard and mouse and display images and videos. The library permits to manipulate images even with matrix and vector algebra routines. It supports the most common dynamic data structures and offers many different basic image processing functions: filtering, edge and corner detection, color conversion, sampling and interpolation, morphological operations, histograms and image pyramids. Beyond this, it integrates many functions for structural analysis of the image, camera calibration, motion analysis, object recognition and machine learning. [Aga06].

2.2.4. Architecture

Since version 2.2, the OpenCV library is divided into several modules, built in library files. Each module serves a specific purpose and includes functions in order to accomplish it.

The *core module* contains the core functionalities of the library, i.e. the basic data structures, both in C and C++ languages, and a variety of functions used by all the other modules. Such functions have many purposes: operating on arrays and matrices, drawing shapes on the images, storing and restoring library data structures and primitive data types in XML and YAML formats and other.

The *imgproc module* contains the main image processing functions. Processing an image means *"using higher-level operators that are defined on image structures in order to accomplish tasks whose meaning is naturally defined in the context of graphical, visual images"* [BK08, 128]. Such functions are, for example, operations to perform various linear or non-linear filtering on 2D images, geometrical transformations of 2D images (resize, affine and perspective warping, generic table-based remapping), color space conversions, histogram analysis, feature detection and object detection.

The *highgui module* allows the library to interact with the operating system, the file system and hardware such as cameras. Its interface permits to open windows, display images, read and write graphics-related files, both images and video, and handle simple mouse, pointer, and keyboard events. It also allows to add simple user interface elements to the displayed windows.

The *features2d module* allows to use different algorithms to implement the *interest points processing*. This technique relies on the idea that instead of analyzing the complete image it is better to work on a limited number of particular pixels. Such points must be distinguishable features in all the images in which they must be detected. To do this, the module offers functions for feature detection, feature description and descriptor matching.

The *calib3d module* focuses on camera calibration and stereo vision. Camera calibration permits to modify images and videos by correcting the errors derived by the use of lenses. Stereo vision allows to have benefits from all the advantages given by using two equal cameras instead of one, like for example measuring distances with high precision.

The *video module* implements some techniques for video processing, in particular *motion estimation*, *feature tracking*, and *foreground extraction* algorithms. Motion estimation techniques define motion vectors that describe the transformation from consequent frames. Feature tracking algorithms improve motion estimation by working only with some features instead of sets of pixels. Foreground extraction algorithms try to recognize

foreground objects of interest comparing an observed image with an estimate of the same image as if it contained no objects of interest.

The *objdetect module* allows the detection of particular categories of objects such as faces or cars. To do so, it uses functions for training the software in recognizing a particular class of objects and verifying if an input image contains such kind of object once the learning is terminated.

OpenCV also includes other utility modules: the *ml module* contains machine learning functions, the *flann module* computational geometry algorithms, the *contrib module* some contributed code, the *legacy module* some obsolete code and the *gpu module* support for gpu acceleration [Lag11, Ope12b, BK08].

3. Objective

In the first section the chapter introduces the purposes of the working team and motivates the choices of the objectives of the software. Then, in the second one, it presents the goals of the software and, in the third, the requirements.

3.1. The Context

The work presented in the following chapters is a subset of the work of a research group, which aims to provide ACT-R of a visual system which is as similar as possible to the human one. As this objective is very broad, this work focuses only on a particular subset of tasks.

To reach such goal, the researchers continuously monitor the progresses in the computer vision research field in order to find useful innovative techniques to combine with consolidated solutions. Such search, anyway, it is not easy. In fact, although computer vision studies are very active and find every year new algorithms able to solve different problems, often such algorithms introduce contextual information that makes them tasks specific: they are suitable to solve only the tasks for which they are designed. For this reason, it is very difficult to find a general purpose solution which is able to make machine vision system behave like the human one.

Like most of research works, this project has to deal with a certain degree of uncertainty that makes difficult to set any time constraints. The author of this work, however, has been inserted for a limited time interval in a team composed by three people, whose aim is to implement particular solutions in order to solve specific tasks. For this reason the following parts of this document describe only the features developed during this amount of time by the author, explaining them in the context of the objectives of the team.

3.1.1. The Goal of the Team

The goal of the team is to create a general purpose software, which receives as input a video stream or an image and performs an object recognition process on them. ACT-R

uses the information obtained by this tool to orientate inside a building. In particular, the software analyzes a video looking for objects and transmits all the recognized items to a model developed in ACT-R. The model has to distinguish different categories of rooms basing on the information about the objects they contain.

As a starting point for such software, the team has to implement a tool for recognizing geometrical shapes in the input images. Such software has to accomplish two goals: the first one is to extract a subset of all the features in the image, in order to simplify the object recognition process; the second one is to make easier the definition of models for the modelers. This tool, which represents the main topic of this work, is described better in the next sections.

3.2. The Goals of the Software

The software to be designed and developed is a visual module for the cognitive architecture ACT-R, whose aim is to improve ACT-R visual perception in order to make it more similar to the human one.

For developing a model in ACT-R, the modelers need a series of parameters on which basing their assumptions about the model itself and its performances. The collection of such data is accomplished thanks to experiments in which human beings take part. For every experiment a specific program is created in order to measure some variables as, for example, the time needed to complete a task and the accuracy of the result. The parameters obtained by these experiments are then used as a reference for the development of ACT-R model.

Image 3.1 shows a test case of an experiment in which the goal is to solve some levels of the game *Rush Hour*. As shown, the grid contains some colored rectangles, each of which represents a car. The player must free the red one making it go out of the grid through the exit on the right. The cars can be moved only in the horizontal and vertical directions, according to their orientation. The lower the number of moves, the better is such solution.

In order to conduct an experiment, the modelers prepare a set of instances of the problem and every person who is going to participate to the experiment has to face a fixed number of them. For each instance, the ad-hoc program shows the image with the initial configuration (figure 3.1) and the person has to give the solution of the game by clicking on the rectangles in the order he thinks to be correct. As the program never updates the image, the player has to imagine the solution. The software manages to interpret the direction of the shift by analyzing the movement of the eye, thanks to a

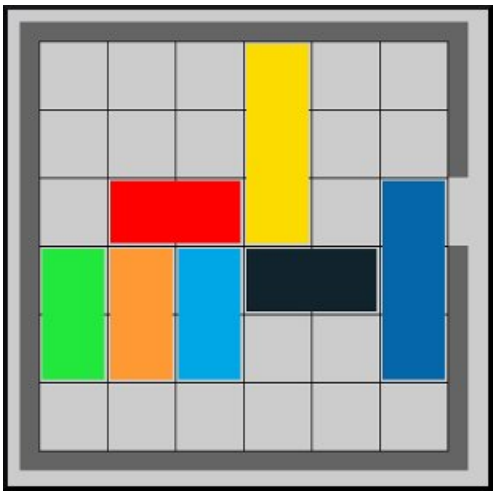

Figure 3.1.: Example of level of Rush Hour game

module of *eye tracking*. Moreover, it records the movements of the mouse, the clicks of the mouse, the time needed to give the solution and the correctness of the solution.

The model written in ACT-R, in order to solve a particular instance of Rush Hour problem, needs to receive as input the initial configuration. At the moment, such configuration is not an image, like the one shown to people during the experiment; rather it is a list of objects written in the language of the model. A desirable improvement of the model should be introducing an object recognition algorithm able to detect the rectangles in the image and generate the correspondent object list in ACT-R language. This approach would make ACT-R visual system more similar to humans'. Moreover, it would make more scalable the process of creating instances for the experiments.

The purposes described above represent the goals of the tool. To achieve them, the software analyzes the image created for the user, processes it extracting the features needed by ACT-R and stores the information in dedicated data structures. Then, when ACT-R requests them, it communicates all the stored data to the cognitive architecture using a format compatible with ACT-R model.

3.3. Requirements

The requirements are grouped in the following sections according to the functional/non-functional classification.

3.3.1. Functional Requirements

The objective of the work is to design and implement a standalone software module which receives as input an image and is able to analyze it and extract some features of it.

The input image is a color image which contains simple shapes. The shapes must not be overlapping and must have the same color hue.

The software must recognize simple shapes, in particular:

- triangles;
- rectangles;
- quadrilaterals;
- circles.

For each shape, it must calculate:

- area;
- perimeter;
- dimensions;
- rotation;
- a rectangular bounding box;
- center;

Moreover, the software has to:

- recognize the color of a single pixel;
- recognize the color of a shape;
- calculate distances between objects;
- make dimensional comparisons between objects;
- calculate the relative position of one object in respect with another one.

The software must be able to communicate with ACT-R. In particular, ACT-R must signal to it which is the image to process and ask for the features to be extracted. The module must return all the information extracted by ACT-R.

3.3.2. Non-Functional Requirements

Product Requirements

The software module must:

- be multi-purpose;
- be portable;
- work in background;
- communicate with ACT-R.

Multi-Purpose The software should be easily adapted to work with all the experiments that will be put in place in the future. Moreover, there should be the possibility to adapt the software in order to use it as a shape recognition tool in the orientation inside a building. For this topic, see 3.1.1.

Portability The software has to work on several operating systems.

Communication with ACT-R The software must transmit the extracted information to ACT-R in a standardized format.

Organizational Requirements

- The language of the implementation of the software must be C++;
- the computer vision library to be used must be OpenCV;
- a strict monitoring of the work is required.

4. Development Process

This chapter, after having introduced the agile development framework *Scrum*, describes the development process adopted by the team in order to complete the work.

4.1. Scrum

Scrum is a framework for the agile management of the project development. As such, it does not define the technical way in which the developers must do the job but it follows the development process. The framework has many dimensions: roles, events, rules and artifacts within the framework serve specific purposes and contribute the Scrum's success [Botb].

The following sections describe one by one these components. The first section describes the main roles of people within the framework, the second focuses on the events and the meetings necessary for the implementation of the methodology and the third one puts in evidence the artifacts needed.

All the information about Scrum are taken from [Botb]. See it for a more detailed description.

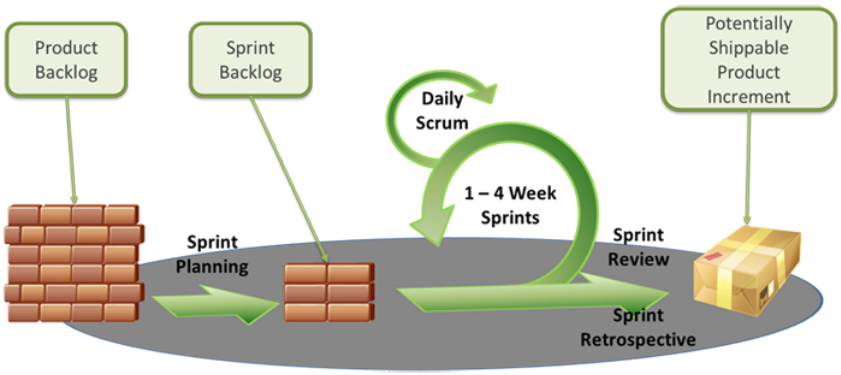

Figure 4.1.: Overview of the Scrum Process

4.1.1. The Scrum Team

The *Scrum Team* is composed of a *Product Owner*, the *Development Team*, and a *Scrum Master*. These figures have different roles but all of them have to reach the same goal, add to the product a finite and usable set of new features at constant time intervals.

Scrum Teams are self-organized and cross-functional. Self-organization gives the members the possibility to choose the best way to accomplish their work, without being directed by others outside the team. This fact not only avoids the team to be directed by someone who has not the technical knowledge to solve specific problems and, consequently, can not find the best ways to achieve the goals, but also allows the Team to optimize its own development process. Moreover, the consequent improvement of the sense of ownership that the Team feels on the product increases the motivation of the members. Cross-functionality gives the team all the competencies needed to accomplish the work without depending on others not part of the team. In this way, all the tasks can be accomplished directly by the team, thus the team is never blocked waiting for the work of someone outside it.

The delivery of the products is iterative and incremental. This fact guarantees a constant feedback on the correctness of the job and ensures that a deliverable version of the product is always available.

The following paragraphs will describe each of the three roles.

Figure 4.2.: The Scrum Team

Product Owner

The *Product Owner* is the person who represents the stakeholders of the product and is responsible for the performance of the team. His role is to define the requirements for

each new version of the product, assign them the priorities and explain them to the team in detail. The requirements are called *Backlog Items* and are included in the *Product Backlog*. This document is described in more details in section 4.1.3.

Scrum rules require a person to assume the role of Product Owner to represent the will of a committee. In this way, he assumes an intermediation role for all the communications between the developers and the stakeholders and becomes a reference point for both these categories. On one side, he has to give the stakeholders all the information about the progress of the development and the performances of the team. On the other hand, he has to decide the priorities of the tasks and deliver the team the messages of the stakeholders.

Development Team

The *Development Team* consists of professionals programmers, whose role is to implement the new functionalities to the product. The team is usually composed of a number of members which varies from three to nine. One team must be self-organizing, this means that only the members can decide step by step the tasks in which the items are split and how to accomplish each of them in order to add functionalities to the product. Every team is also cross-functional, i.e. is composed by people who have different skills. In this way it can be autonomous and it does not have to depend on other people outside the team to accomplish its job. The more synergy the Development Team has, the higher its overall efficiency and effectiveness are.

Scrum Master

The *Scrum Master* has the role to verify that the Scrum process is understood and put in place. He does this by checking that everyone in the team follows Scrum theory, practices, and rules. The Scrum Master is the enforcer of the rules and interacts with all the people inside and outside the Scrum Team in order to avoid useless interactions. On one side, he helps the Product Owner finding techniques for managing the *Product Backlog*, teaching him how to communicate in clear way with the Development Team and understanding and practicing the agile development. On the other hand, he coaches the Development Team in self-organization and cross-functionality, he protects it from unhelpful interruptions and keeps it focused on the tasks. For this, often this role is referred as a *"servant-leader for the Scrum Team"* [Botb, p.6].

4.1.2. Events

Prefixed meetings in Scrum have the purpose of marking time and hence minimizing the need for meetings not defined in Scrum, which can distract the members of the team from their tasks. In this events, different roles in the Scrum Team can interact with regularity, without the need for continuous interruptions. All the events are time-boxed, i.e. every event has a maximum duration. This ensures that only the appropriate amount of time is spent planning, avoiding wasting time.

The following sections describe the *Sprint*, the *Sprint Planning Meeting*, the *Daily Scrum*, the *Sprint Review* and the *Sprint Retrospective*.

Figure 4.3.: The Scrum Events

Sprint

The *Sprint* is the primary unit of development in Scrum. It is a time-box with a prefixed duration and can last from one week to one month. During this period of time the team creates a self-contained portion of product.

Each Sprint starts with the *Sprint Planning Meeting* and ends with the *Sprint Review Meeting* and the *Sprint Retrospective Meeting*. All these events will be described with more details in the next sections.

Sprint Planning Meeting

The *Sprint Planning Meeting* is a meeting at which all the member of the Scrum Team participate. It is the beginning of every Sprint. In this meeting the team decides the new

features the product will include by the end of the Sprint.

At first the Product Owner informs the team about the features that he wants to be added to the product, each one with its priority. Each one is an item in the *Product Backlog*, which is described in the section 4.1.3. Then, the Development Team estimates how long it would take to add every new feature to the product and, consequently, how many of them will be added by the end of the Sprint. Often in this part of the meeting the Product Owner and the Development Team have a talk in order to define better the requirements. This fact guarantees that the developed features adhere strictly to the requirements and leads to a new time estimation for most of the goals.

The goals are split into tasks, each of which takes no more than two days to be accomplished by the Development Team. Every task and the delivery plan are collected in the *Sprint Backlog*, described in the section 4.1.3.

Every Sprint must have a *Sprint Goal*. The Sprint Goal acts like a sort of motivation which reminds the Development Team during the whole Sprint which is in a wider context the goal of their activities. For this reason the Sprint Goal should not be changed during the Sprint.

Daily Scrum

The *Daily Scrum* is a fifteen minutes event for the Development Team and the Scrum Master, that allows the team to synchronize the work and plan the next day activities. This purpose is achieved analyzing the work done in the last day and forecasting the work for the next one. During the meeting, each Development Team member explains what he did since the last meeting, what he is going to do before the next meeting and the problems he had to front. The meeting has the purpose of evaluating the progress towards the Sprint Goal and analyzing the trend of the progress in comparison with the Sprint Backlog. This allows the Scrum Team to measure its speed and, consequently, to make better time estimations in the next Sprints.

Sprint Review

The *Sprint Review* is a time-boxed meeting which closes every Sprint. In this meeting the increment of the work is analyzed and, if needed, the Product Backlog is updated. During the meeting, the Product Owner analyzes what has been completed and what has not. The Development Team describes the problems it encountered, how it solved them and shows the new functionalities added to the product. Then the whole team discusses of new features to be added, old ones to be deleted or how to improve other ones and, more

generally, collaborates on what to do next, updating consequently the Product Backlog.

Sprint Retrospective

The *Sprint Retrospective* is a time-boxed meeting that takes place after ever Sprint Review. It is an opportunity for the Scrum Team to analyze its Scrum implementation and create a plan to improve the next Sprint. During the meeting the focus is set on people, relationships, processes and tools. The most important successes are shown and a plan for implementing potential improvements is created. The purpose of this meeting is to make the implementation of the Scrum method more effective by optimizing the development process and introducing techniques that make the method more productive and enjoyable.

4.1.3. Artifacts

Scrum's *artifacts* represent the work in many different ways. The basic artifacts required by the framework are the *Product Backlog* and the *Sprint Backlog*.

Product Backlog

The *Product Backlog* is an artifact which contains the list of the requirements for a product. The elements of the list, called *Backlog Items*, are sorted by priority.

Every Item must have a name, a description, an estimate and a priority. For each Item, the Product Owner, representing the stakeholder, sets the priority and the Development Team defines the estimate.

The Product Backlog is dynamic. Its earliest versions only contain the initially and best understood requirements of the product. Then, as long as the product evolves and new requirements are introduced, it is continuously updated. In this way, it always contains all the features, enhancements and fixes that must be added in the future to the product.

As the higher ordered Items are more important than lower ordered ones, they are clearer and more detailed. The team can make more precise estimates thanks to the greater clarity and increased detail.

The activity of adding detail, estimates, and order to the items in the Product Backlog is called *grooming*. The Scrum Team decides how and when to do it. Anyway the grooming activity must not consume more than the 10% of the capacity of the team.

Sprint Backlog

The *Sprint Backlog* contains the Product Backlog Items selected for the current Sprint and a plan for completing all the modifications to the product and realizing the Sprint Goal. The Sprint Backlog contains all the functionalities that should be added to the product by the end of the Sprint, each of which with its estimates and priorities, and, at the same time, monitors the state of the work during every day of the Sprint.

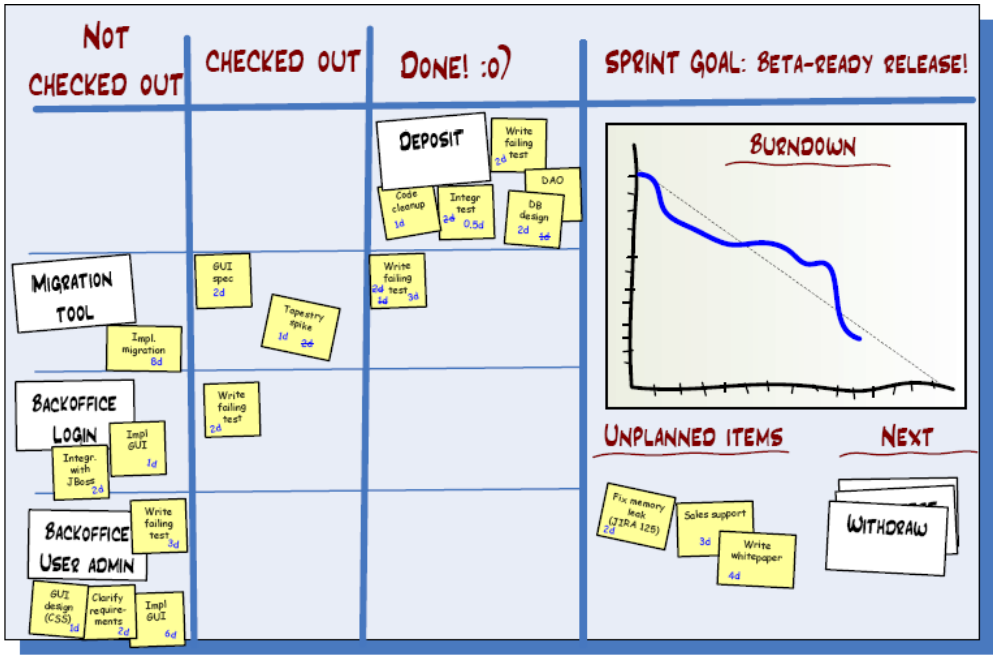

Figure 4.4.: Sprint Backlog and Burn Down Chart

The level of detail of the work is such that it can be measured during the Daily Scrum. In this meeting, in fact, the amount of work done in the day is calculated and the activities for the next day are planned. Usually every Product Backlog Item is split into tasks, each of which must be accomplished in not more than sixteen hours of work. The tasks are never assigned; rather, each member of the team takes charge of some of them during the Daily Scrum, according to the set priority and his own skills.

The Sprint Backlog is updated every day during the Daily Scrum. If some new work emerges to be necessary in order to complete some requirements, the Development Team adds it to the Sprint Backlog. If there are some elements which are deemed unnecessary, they are removed.

The state of the ongoing activities can be monitored by a *Burn Down Chart*, which measures the number of completed tasks per day and compares it with an ideal trend, which represents the estimates of the tasks. This chart allows to analyze the accuracy of

the estimates and to have a visual representation of the ongoing work. The knowledge of the speed of the team allows the team to make better estimations in the future Sprints.

4.2. The Adopted Development Process

The development team decided to adopt an incremental and iterative development process, implementing the *Scrum* framework.

The main motivation of this choice is the capacity of this method to front very well the modifications of the requirements. In most of cases, in fact, even if the requirements at the beginning seem to be well defined and clear, they change during the development. For example, it could happen that new requirements reveal to be more important than other ones or that after some clarifications between the client and the developers some features are redefined, some are deleted and some others are added. The possibility of an incremental and iterative development process to make continuous reviews of the product faces better such changes in the requirements.

Moreover, the introduction of meetings in which is possible to discuss the state of the work and the working method improves considerably the performances of the team itself. In this way, in fact, the supervisors of the development team can postpone all the not essential communications until the next meetings, without the need to interrupt his work nor the developers'. This leads to a more focused and less stressed work both for the developers and the supervisors.

The possibility to check the work at every Sprint increases the compliance between the system and the requirements. During the Sprint Reviews, the new increment of the software is shown. In this meetings, the clients give the Scrum Team a feedback about what they want and if the feature developed satisfy their need. So if there have been some misunderstandings in the comprehension of the requirements, they can be fixed in the next Sprints. This continuous product revision leads to a strict adherence between the system and the requirements.

The adopted development process allows the Product Owner and the Scrum Master to evaluate the performance of the team. As the features to be added and the estimates are decided during the Spring Planning Meeting, the Product Owner and all the members of the team have the plan of the whole Sprint. During the Sprint Review, the members can monitor how the actual work has been done in comparison with the plan. On a shorter period, the Daily Scrum is used to monitor the activity of the days, while the Product Backlog contains the history of all the modifications done on the software since it was born. For all there reasons, with Scrum, the performances of the team are constantly

measured.

5. Design

This chapter describes the design of the software, focusing on its main important aspects. The first section describes the architecture from a high-level point of view. The second one shows the organization of processes and threads. The third one explains the choices about the communication mechanism and the last one illustrates the architecture of the visual module.

5.1. The Architecture

This section gives a high level overview of the architecture of the software by introducing its main building blocks and the main modules they interact with.

Image 5.1 on the left side contains the vision module, composed by the sub-modules *Server* and *Visual Operations*, and, on the right, Lisp interpreters that run ACT-R models. The two parts communicate to each other thanks to a client-server communication mechanism based on TCP/IP protocol in which the Lisp interpreter represents the client and the visual module the server.

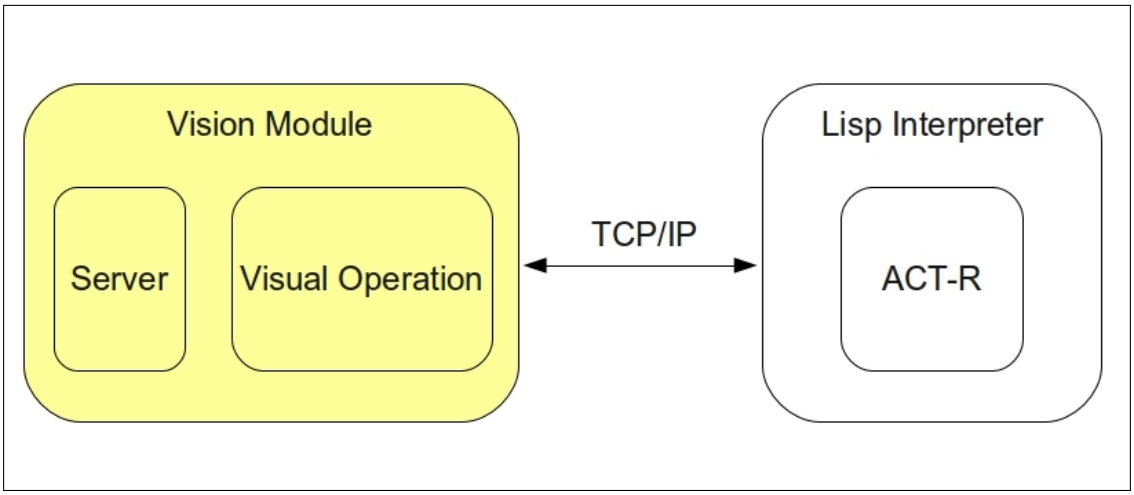

Figure 5.1.: High level scheme of the architecture

ACT-R part is not discussed in this work but is fundamental for the choices about the communication protocol. This document focuses on the *Vision Module*, whose main operations are extracting features from the input data and sending them to the models in a way that can be understood by the cognitive architecture. These functions are realized by the two sub-modules contained in Vision Module.

5.2. Processes

Figure 5.2 represents a scheme of processes and threads that compose the architecture. As shown in the picture, the Lisp interpreter and the Visual Module run on two separated processes. The processes are independent and their management is left to the operative system.

The design on the communication protocol is such that the two processes can be run only on the same computer. The images to be processed, in fact, at the moment are not sent as messages from the client to the server but are stored on the computer which runs both the processes. Anyway, it is enough to modify the message exchange by adding a message sent from the client to the server encapsulating the image data in order to run them on two different computers.

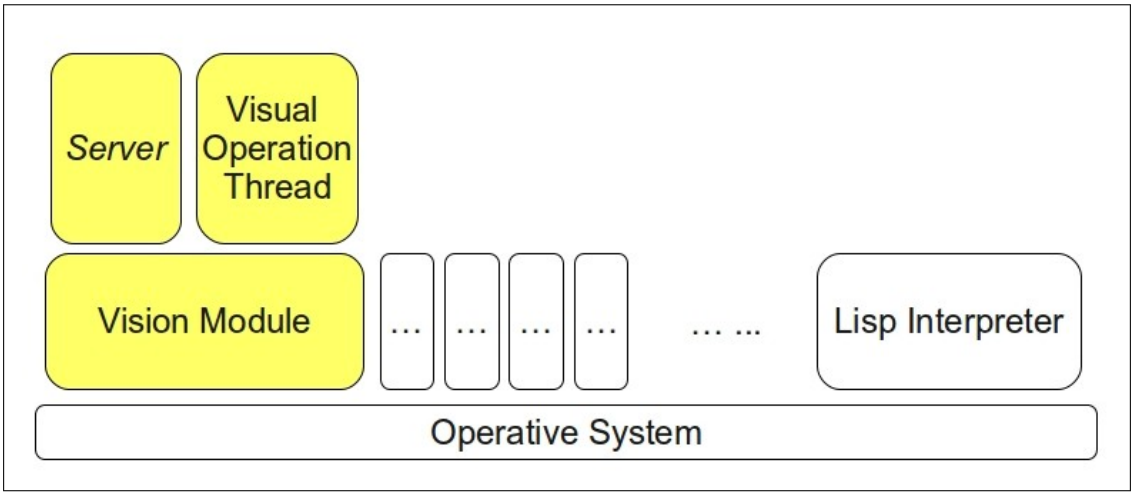

Figure 5.2.: High level scheme of the architecture

The Vision Module is composed by two threads: one for the server and the other for the visual processing. The server thread runs in background while the visual operation one is the main thread. The problems regarding race conditions are dealt with *semaphores*; all the other issues related to threads management are automatically handled by the *Boost*

library. The thread organization of ACT-R is not discussed in this work.

5.3. Communication

In order to exchange information between each other, ACT-R and the developed module use a synchronous *client-server* communication mechanism based on TCP. ACT-R models represent the clients, the developed software the server. The format used for the messages is JSON.

The client-server approach has been chosen because it is standardized and is supported by the interpreters that run ACT-R. Moreover, the time spent for coding and decoding the message is negligible if compared to the time needed for the image processing. Finally, the possibility of using shared resources and the transparency offered by the mechanism ensure very high flexibility and scalability.

Another solution analyzed for the communication between ACT-R and the Visual Module is compiling the Visual Module as a static library and then loading it directly in the Lisp interpreter. The latter is able to call the static functions thanks to *foreign functions*. In this way, the communication is more efficient because all the overhead introduced due to the inter process communication is avoided. This method, though, is not maintainable. In fact, every Lisp interpreter has a different method of calling foreign functions [SWI]. As consequence of this, every time the interpreter is changed, the interface of the visual module must be re-written. At design time it has been decided to prefer maintainability to an irrelevant improvement of the performances, thus the client-server has been chosen.

TCP has been preferred to UDP mainly because of the reliability guaranteed by that protocol. Secondly, the bi-directionality of the communication and the small quantity of data exchanged make it more suitable for the purpose.

The communication is synchronous. The client, once having sent the request, stays pending until it receives the response. This choice is coherent with the application domain: as the software module acts like a vision system for the cognitive architecture, it makes sense that the model does not continue the reasoning until the available visual information is received.

The Lisp interpreter which runs ACT-R models is the client, while the developed software is the server. When a model needs to acquire information about an image, it connects to the server. Once connected, it can encode and send a request to the server. After the server has received and decoded successfully the request, it processes the input data. When the visual information is ready to be sent, the server encodes it and send it back to the client. After this, the message is decoded and its content is translated in chunks

and then delivered to the model, that uses it for the cognitive reasoning.

All the messages exchanged between client and server are encoded in JSON. Appendix B contains an example of the messages.

5.4. Vision Operation Module

This section describes the design of the Vision Operation Module. For reasons of clarity, the description of the architecture follows a top-down approach and makes use of diagrams. The first subsection gives an overview of the module and the following ones study in deep specific aspects of it.

To better explain the contents and the structure of the design, the description is supported by UML diagrams. The following sections describe the module at different levels of detail and the diagrams reflect this approach. In subsection 5.4.1, for example, they are used only to present how the classes are organized in the overall structure and that is why they contain only the names of the classes. Further details are added in the diagrams in the next subsections, according with the top-down approach of the chapter.

The reader will notice that some classes that appear in the following diagrams are not described in the chapter. The reason of this is that they do not accomplish to any functions described in the requirements section of chapter 3. Their presence makes sense in the context of the whole software, described in 3.2. It has been decided not to remove such classes from the diagrams because they influence the architecture and motivate particular design and implementation choices.

This section explains only the most important architectural aspects of the software. Not particularly interesting classes, like, for example, the utility ones are not described here. For a more detailed description about all the aspects of the architecture, see appendix A.

5.4.1. Architecture

The vision operation module is composed of three main parts, one for processing images and videos, one for containing the hierarchy of the objects recognized during such processing and one containing the exception classes and the utility functions used by all the other components.

Image 5.3 shows a class diagram which describes two of the parts that compose the software. The part dedicated to the processing of images and videos is located on the left and is composed by the classes *Proxy*, *FeatureGetter*, *FeatureExtractor*, *Input*, *MarkerDetector* and *QRScanner*. The object hierarchy, which is shown on the right, contains

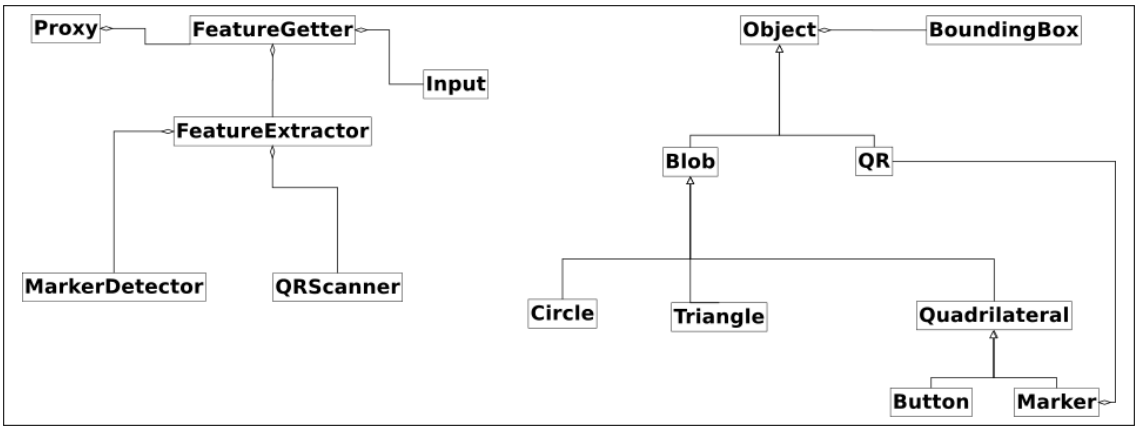

Figure 5.3.: Class diagram illustrating a high level overview of the vision operation module

the classes *Object*, *BoundingBox*, *Blob*, *QR*, *Circle*, *Triangle*, *Quadrilateral*, *Button* and *Marker*.

The third part is not included in the section because it contains only utility classes and functions. The reader can find a description of that part in appendix A.

5.4.2. Feature Extraction Classes

The classes that play the role of acquiring the images and extracting features from them are *Input*, *FeatureGetter* and *FeatureExtractor*. *Proxy* has the role of intermediation between this part of the software and the one which manages the communication with ACT-R.

Figure 5.4 contains the class diagram that designs the interface of these classes. *FeatureExtractor* represents the core of this part. It receives in input an image and processes it in order to extract the features that contains. For each extracted feature it creates a correspondent element in the *object hierarchy*. Such hierarchy is described in section 5.5. Once the objects are generated, the class creates a list containing them and delivers it to FeatureGetter. FeatureExtractor is also responsible for accomplishing other goals described in the requirements, like recognizing colors of pixels and objects, calculating the distances of two objects as well as the relative position of one object in respect with another one and making dimensional comparisons between two objects. These features have not been included in the initial design of the class.

Input is responsible for interacting with the file system and the hardware of the computer. In particular its tasks are loading images from files, saving images to files and loading video streams from the web-cam. The acquired data are delivered to FeatureGet-

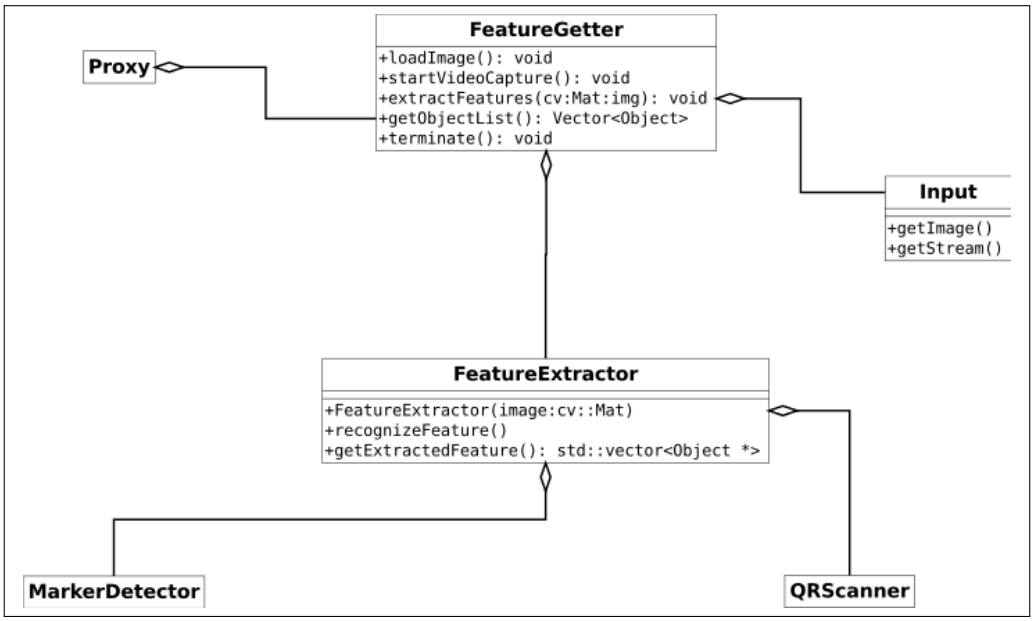

Figure 5.4.: Class diagram illustrating the main roles in the feature extraction process

ter.

FeatureGetter represents a level of abstraction that avoids to call directly the functions of FeatureExtractor and Input, providing a simpler interface. Its functions create the Input object, receive from it the input data, delivers it to FeatureExtractor for the extraction of the features and delivers the received object list to Proxy.

Proxy represents a further level of abstraction between FeatureGetter and the part of the software which manages the communication with ACT-R. Most of the procedures for the communication with the cognitive architecture, in fact, need to use specific data types. The main function of this class is to transform each object of the object list in the type necessary for the communication to ACT-R.

As the classes *MarkerDetector* and *QRScanner* do not accomplish to any functions described in the requirements section of chapter 3, they are not described in this work.

5.5. Class Hierarchy

The class hierarchy contains all the objects that are recognized during the processing of the image. Each feature has a correspondent object that describes it.

Figure 5.5 contains a class diagrams that specifies such hierarchy. *Object* is the superclass that represents all the possible objects recognized in an image during the feature extraction process. All the elements of this type must have two attributes: a *rotation*

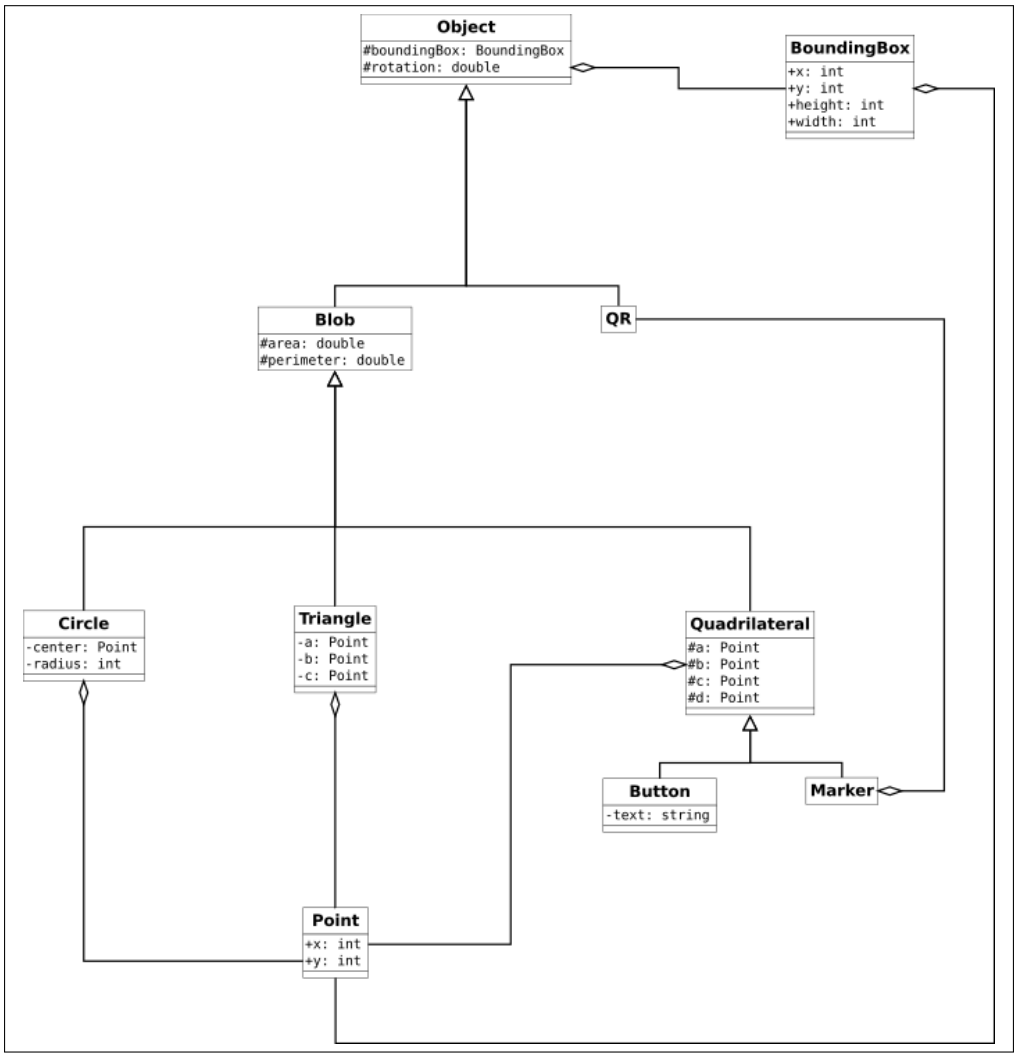

Figure 5.5.: Class diagram illustrating the class hierarchy of the recognized objects

value and a *boundingBox*. The latter is defined by the class *BoundingBox*. Such element can have only rectangular shape, in fact its attributes are the coordinates of a point and horizontal and vertical dimensions.

The type Object can be specified by two sub-types: *Blob*, that represents a generic shape, and *QR*, representing a QR element. The presence of QR is not important for the shape recognition task but it justifies the presence of Blob in the hierarchy. Without QR class, in fact, Blob could have been incorporated in Object. Blob is provided of the attributes *area* and *perimeter*, which, as suggested by the names, represent respectively the area and the perimeter of the shape.

The classes *Circle*, *Triangle* and *Quadrilateral* extend *Blob*. They represent the simple shapes that can be recognized during the recognition process. Each item is defined starting

by basic geometrical elements: the circle by center and radius, the triangle by three points, the quadrilateral by four points. *Point* class, in particular, has been introduced in this diagram for reasons of completeness.

The class *Button* specifies Quadrilateral by adding the attribute *text*, which represents the message of such button. The class *Marker* is not described in this document.

6. Implementation And Testing

6.1. Shape Recognition Extraction Algorithm

6.2. Testing

7. Conclusions

7.1. Sviluppi futuri

aggiungere: riconoscimento ellissi riconoscimento testo

introdurre il flusso video –¿ predisposto

migliorare le performance dell'algoritmo in modo tale che la shape detection sia utilizzata in tempo reale pnell'ambito della navigazione con robot.

A. Appendix A

A.1. Actual implementation of the software

As in chapter 5, in this section class diagrams are used in order to clarify the description. Again, such description follows a top-down approach: section A.1.1 gives an overview of the architecture and for this reason the diagram in figure A.1 contains only the names of the classes. The following sections study in deep specific aspects of the software and, consequently, contain more detailed diagrams.

A.1.1. Overview

Figure A.1 shows the class diagram of the whole software. According to the design specifications defined in chapter 5, the software is organized in four main parts, each of which has its own role.

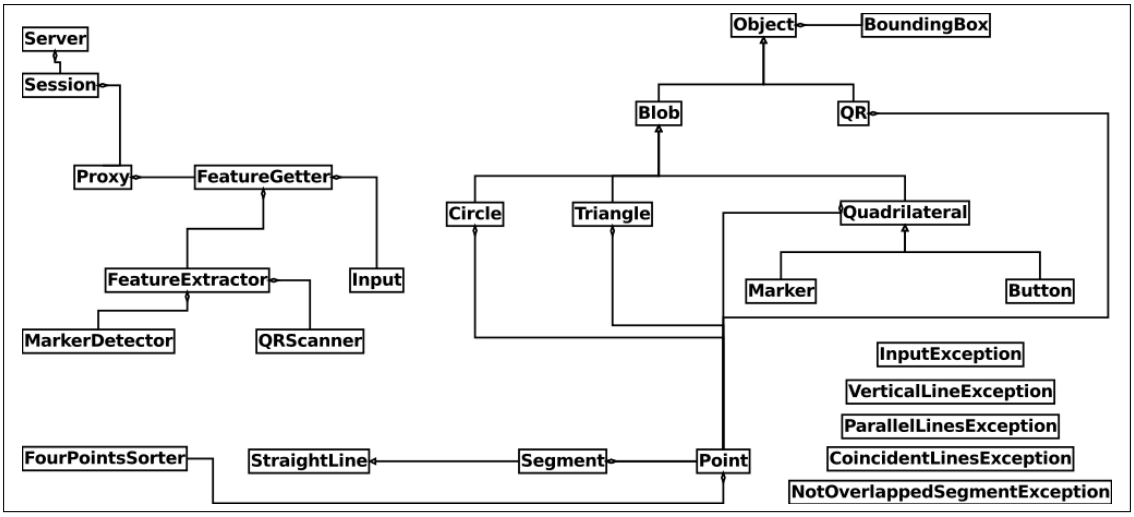

Figure A.1.: Class diagram which illustrates the whole architecture of the software

Classes *Server* and *Session* manage the communication with ACT-R, implemented by a client server approach. Classes *Proxy*, *FeatureGetter*, *Input*, *FeatureExtractor*, *MarkerDe-*

tector and *QRScanner* have the role of extracting features from the input data and preparing the messages to be sent to the cognitive architecture. *Object*, *BoundingBox*, *Blob*, *QR*, *Circle*, *Triangle*, *Quadrilateral*, *Marker* and *Button* form the object hierarchy. *InputException*, *NotOverlappedSegmentException*, *ParallelLinesException*, *VerticalLineException*, *CoincidentLinesException*, *FourPointsSorter*, *StraightLine*, *Segment* and *Point* represent the utility classes.

Comparing the classes shown in figure A.1 with the design choices defined in chapter 5, the reader can notice two facts: both the feature extractor and the class hierarchy parts strictly adhere to the project and, accordingly to the design choices, the classes containing the utility functions and the features for the communication with ACT-R have been properly developed.

A.1.2. Communication with ACT-R

In order to exchange information between each other, ACT-R and the developed module use a synchronous *client-server* communication mechanism based on TCP. ACT-R models represent the clients, the developed software the server. The format used for the messages is JSON.

The client-server approach has been chosen because it is standardized and is supported by the interpreters that run ACT-R. In addition, the possibility of using shared resources and the transparency offered by the mechanism ensure very high flexibility and scalability.

TCP has been preferred to UDP mainly because of the reliability guaranteed by that protocol. Secondly, the bi-directionality of the communication and the small quantity of data exchanged make it more suitable for the purpose.

The communication is synchronous. The client, once having sent the request, stays pending until it receives the response. This choice is coherent with the application domain. As the software module acts like a vision system for the cognitive architecture, it makes sense that the model does not continue the reasoning until the available visual information is received.

The Lisp interpreter which runs ACT-R models is the client, while the developed software is the server. When a model needs to acquire information about an image, it connects to the server. Once connected, it can encode and send a request to the server. After the server has received and decoded successfully the request, it processes the input data. When the visual information are ready to be sent, the server encodes them and send them back to the client. After this, the information contained in the message is translated in chunks and then delivered to the model, that uses it for the cognitive reasoning.

All the messages exchanged between client and server are encoded in JSON. B contains an example of the messages.

The part responsible for the communication with the cognitive architecture is composed by the classes *Server* and *Session*.

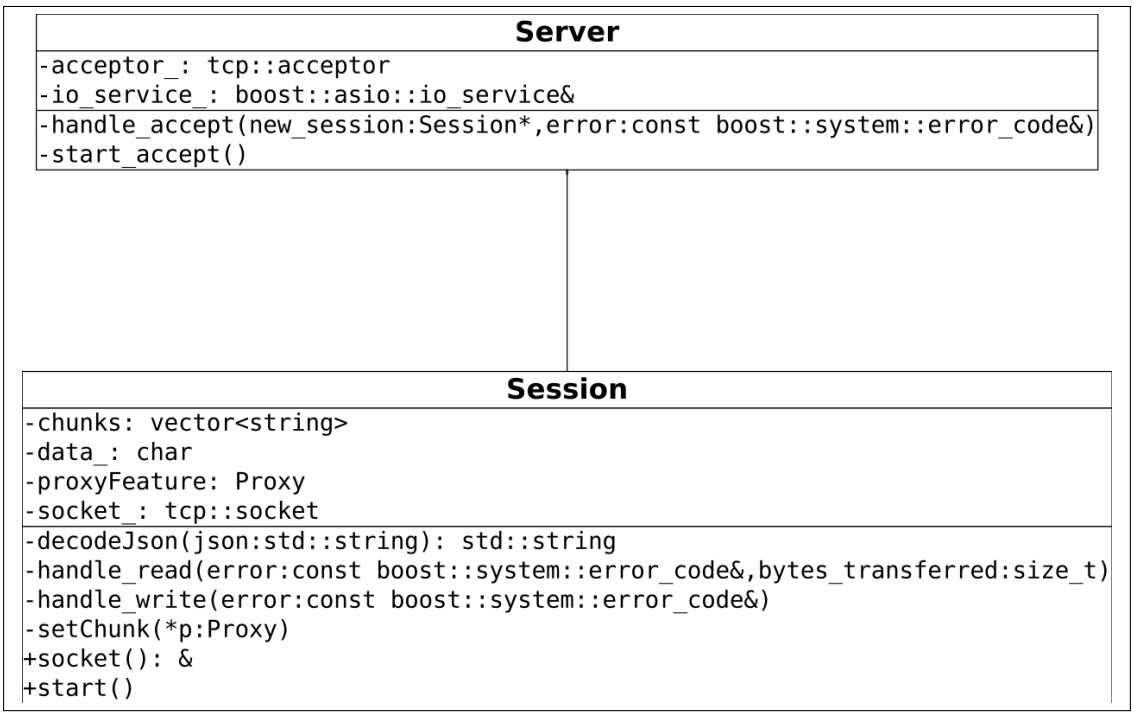

Figure A.2.: Class diagram describing the implementation part of the software dedicated to the communication

A.1.3. Feature Extraction

Image A.3 shows a class diagram of the implementation of the feature extractor part of the software. Analyzing it, the reader can notice that the implementation adheres very good to the design specifications, described at page 35. The classes, in fact, are the same but with more attributes and methods. Such additions have been introduced at development time, as the requirements became clearer. Notice that some methods, like for example *setCroppedImage* in FeatureExtractor class, are used to obtain goals different from the one described in this work, but make sense in the general context of the team work. This topic is better explained in the introductory part of chapter 5.

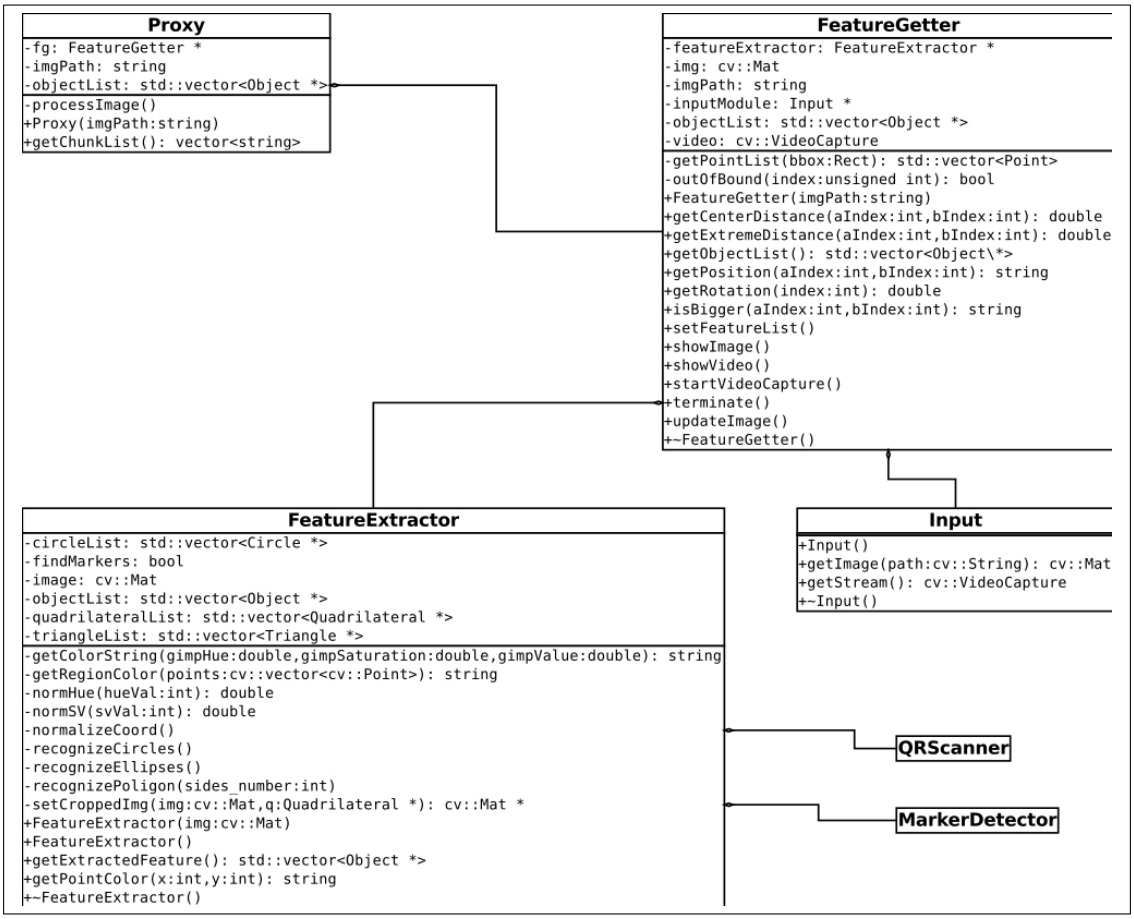

Figure A.3.: Class diagram describing the implementation of the feature extraction part of the software

A.1.4. Object Hierarchy

As it can be noticed comparing the diagram in figure A.4 with the diagram of the design at page 37, the differences between the implementation and the project are minimal. In

fact, further classes have not been introduced; rather, the existing ones have been provided with additional parameters and methods, as, for example, the attributes *color* and *center*. According to the iterative development process, different parameters have been added at different moments of the development.

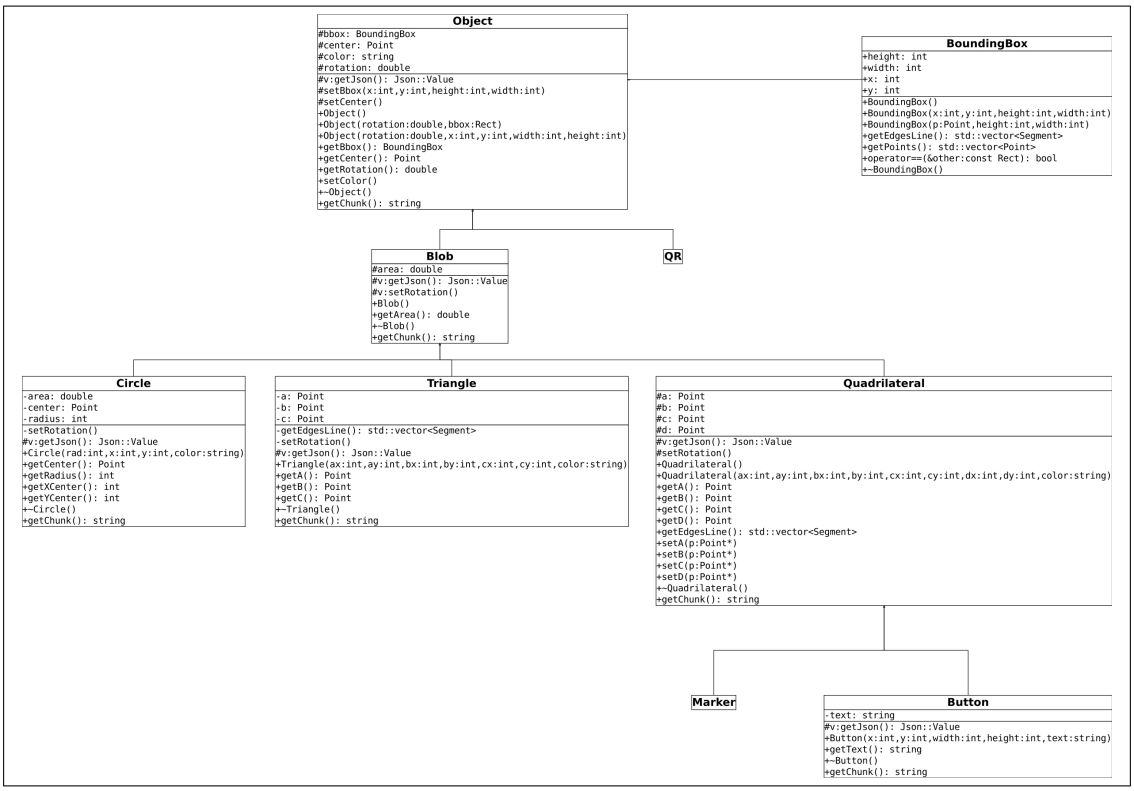

Figure A.4.: Class diagram that shows the implementation of the object hierarchy

A.1.5. Utility Part

The classes which form the utility parts of the software are divided in two categories: the exception classes and the ones representing all the other utilities. To these, two modules containing functions are added. Figure A.5 shows a class diagram containing all the classes that constitute this part of the software. The modules, as not classes, can not be shown in the diagram.

Classes *InputException*, *NotOverlappedSegmentException*, *ParallelLinesException*, *VerticalLineException* and *CoincidentLinesException* are the exceptions classes. As such, they are used to handle all the anomalous or exceptional events occurring at run-time.

StraightLine, *Segment* and *Point* represent geometrical entities. These elements serve as basic structures for creating the objects in the hierarchy and making evaluations between

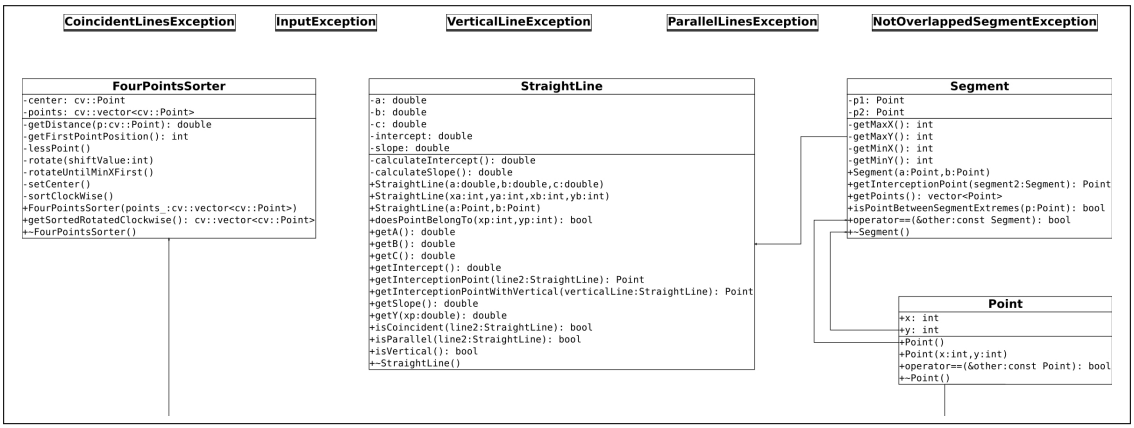

Figure A.5.: Class diagram showing the utility part of the software

couples of objects. Such evaluations can be both qualitative and quantitative, as, for example, the distances between the centers of two objects or the relative positions of one object in respect with another.

FourPointsSorter and the function modules contain the basic methods which are fundamental for all the operations executed by all the other classes of the software. For reasons of completeness the prototypes of the functions are listed in appendix A.

A.2. Prototypes of the utility functions

This section contains the prototypes of the functions that are not contained in classes and thus are not inserted in chapter 6

formattare decentemente sta roba

Functions prototypes in extractorUtils.h

```
    double angle( cv::Point pt1, cv::Point pt2, cv::Point pt0 );
    double getDistance(cv::Point a, cv::Point b);
    bool similar(cv::vector<cv::Point> a, cv::vector<cv::Point> b);
    bool tooClose(cv::Point a, cv::Point b);
    bool isFalse(cv::vector<cv::Point> square);
    bool isTriangle(cv::vector<cv::Point> terna);
    cv::vector<cv::vector<cv::Point> > deleteOverlapped(cv::vector<cv::
vector<cv::Point> > oldList);
    cv::vector<cv::vector<cv::Point> > squaresSort(cv::vector<cv::vector<cv
::Point> > squareList);
```

```
    cv::vector<cv::vector<cv::Point> > deleteFalseSquares(cv::vector<cv::
vector<cv::Point> > oldList);
    cv::vector<cv::Point> sort4PointsClockwise(cv::vector<cv::Point> points
);
    cv::vector<cv::vector<cv::Point> > deleteFalseTriangles(cv::vector<cv::
vector<cv::Point> > oldList);
```

Functions prototypes in utils.h

```
    int getMinMax(const std::vector<Point>& coords, int type);
    std::vector<Point> sort4PointsClockwise(std::vector<Point> points);
    double erone(Point a, Point b, Point c);
    bool inLinePoints(int ax, int ay, int bx, int by, int cx, int cy );
    double getDistance(Point a, Point b);
    bool areSame(double a, double b);
    double getMin(const std::vector<double>& values);
    bool fileExists(char * path);
    string intToString(int a);
```

B. Appendix B

B.1. Message Structure

During the communication between the *Lisp* interpreter and the vision module, two types of message are used:

- the requests, sent from the client to the server;
- the responses, sent from the server to the client.

B.1.1. Request Format

The request message consists of two parts, one fixed and one variable. The fixed part is the attribute `cmd`. The variable part is the value. The only meaningful value for this work is `getFeature`. This command tells the server to extract all the possible features from the input image. Listing B.1 shows the message for this request:

```
{"cmd":"getFeature"}
```

Listing B.1: Request Message

B.1.2. Response Format

The response message contains a list of all the shapes recognized in the input image. Each item of the list is represented by a list of attributes:

- `Bbox`: a rectangular bounding box, defined by two points;
- `Color`: the color of the shape;
- `Type`: the type of the shape;
- `Vertices`: the vertixes of the shape, if present.

Listing B.1.2 shows an example of response, in which a rectangle and a triangle are detected:


```
[
 {
   "Bbox":[
            {
               "x":30,
               "y":7
            },
            {
               "x":44,
               "y":22
            }
          ],
   "Color":"green",
   "Type":"Quadrilateral",
   "Vertices":[
                {
                   "x":30,
                   "y":7
                },
                {
                   "x":30,
                   "y":21
                },
                {
                   "x":44,
                   "y":22
                },
                {
                   "x":44,
                   "y":7
                }
               ]
 },
 {
   "Bbox":[
            {
```

```
                    "x":4,
                    "y":7
                },
                {
                    "x":21,
                    "y":21
                }
            ],
        "Color":"red",
        "Type":"Triangle",
        "Vertices":[
                    {
                        "x":12,
                        "y":7
                    },
                    {
                        "x":4,
                        "y":21
                    },
                    {
                        "x":21,
                        "y":21
                    }
                ]
    }
]
```


Listing B.2: Example of Response Message

References

[ABB+04] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *PSYCHOLOGICAL REVIEW*, 111:1036–1060, 2004.

[Aga06] Gady Agam. Introduction to programming with OpenCV. Technical report, 2006.

[And76] J.R. Anderson. *Language Memory Thought*. The Experimental Psychology Series/ Arthur W. Melton consulting ed. Taylor & Francis Group, 1976.

[And93] J.R. Anderson. *Rules of the Mind with Mac Dis*. Taylor & Francis Group, 1993.

[BEAA09] A.D. Baddeley, M.W. Eysenck, M.C. Anderson, and M. Anderson. *Memory*. Taylor & Francis Group, 2009.

[BK08] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[Bota] Dan Bothell. *ACT-R 6 Reference Manual*.

[Botb] Dan Bothell. *Scrum Guide 2011*.

[Fau93] O. Faugeras. *Three-Dimensional Computer Vision: A Geometric Viewpoint*. Artificial Intelligence Series. Mit Press, 1993.

[FP11] D.A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Pearson, 2011.

[HZ03] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge books online. Cambridge University Press, 2003.

[Lag11] R. Laganière. *OpenCV 2 Computer Vision Application Programming Cookbook*. Packt Publishing, Limited, 2011.

[New94] A. Newell. *Unified Theories of Cognition*. The William James Lectures. Harvard University Press, 1994.

[Ope12a] Opencv change logs, 2012. Available on line.

[Ope12b] Opencv documentation page, 2012. Available on line.

[Ope12c] Opencv web page, 2012. Available on line.

[Sea02] Andrew Sears. *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. Lawrence Erlbaum, 2002.

[SWI] Swig documentation page. Available on line.

[TV98] E. Trucco and A. Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.