TECHNISCHE UNIVERSITÄT
CHEMNITZ

Department of Computer Science
Distributed and Self-organizing Systems Group

Master's Thesis

Visual Analysis of User Interfaces: Integrating Analyzers for Improved UI Object Detection

Juan Sebastian Quintero Victor Flamenco

Chemnitz, October 8, 2019

Examiner: Prof. Dr.-Ing. Martin Gaedke
Supervisor: Sebastian Heil

Quintero, Juan Sebastian
Flamenco, Victor
Visual Analysis of User Interfaces: Integrating Analyzers for Improved UI Object Detection
Master's Thesis, Department of Computer Science
Technische Universität Chemnitz, October 2019

Abstract

User Interfaces (UIs) transform rapidly due to frequent changes in user requirements and evolving technologies; this can affect user-adoption negatively, creating the need for UI analysis. Such analysis depends on the computation of UI metrics. Some metrics such as *wide-space region* or *visual-clutter* are object-dependent, meaning that computing them requires knowing the types and locations of the objects that are present in the UI. Page segmentation is a traditional technique at this regard; manual page segmentation is time-consuming, and many automatic approaches in the context of Web User Interfaces (WUIs) implement Document Object Model (DOM) parsing, being strongly dependent on specific implementations and with limited feasibility. More recent computer-vision approaches provide alternatives to segment interfaces resembling a more natural strategy; nevertheless, the segmentation accuracy still requires improvement.

Existing UI object detection approaches have individual strengths and weaknesses; some detectors perform well in particular scenarios and with specific object types, whereas others perform better in different conditions, meaning that a single approach with outstanding detection results in all cases does not exist. This formulation leads to the idea of finding a suitable way of combining diverse approaches, and as a consequence, improve the overall quality of UI object detection. The present thesis aims at achieving such improvement with the integration of analyzers emphasizing on vision-based techniques; it investigates alternatives in the fields of *Computer Vision*, *Deep Learning*, and *Workflow-oriented Architectures*.

This research produces the concept and prototype of a solution that integrates UI-oriented object detection approaches, with a strong emphasis on vision-based state-of-the-art techniques; it consists of two solution parts: On the one hand, the first solution part - UI Object Detection Using DNNs- addresses two main challenges. First, obtaining a sufficient training dataset as required by Deep Neural Networks (DNNs); and second, avoiding the computationally expensive task of re-training the network with datasets that grow over time. This part proposes an automated data-collection procedure and an object detection strategy that learns progressively. On the other hand, the second part - Combining UI Object Detection Analyzers- addresses the unification of a wide variety of object detection tools, a task that faces the challenges of ensuring uniform communication interfaces between tools and the platform, resolving conflicts in inconsistent outputs, and exploring suitable mechanisms to assign a weight to tools. This part proposes the architecture of the Thulium

Platform, a workflow-oriented tool for integration of various UI object detection implementations; it offers a common infrastructure for virtually joining and reusing distributed tools that can be dynamically plugged in, and become accessible through well-defined interfaces.

In pursuit of testing the implemented solutions, an evaluation is performed using 75 randomly-selected UI screenshots with representative examples across a variety of domains of webpages, and representative object types; the assessment proves, among other findings, that the proposed data-collection mechanisms are reliable, and the DNN-based architecture can learn progressively. Moreover, the Thulium Platform can suitably combine different detectors, so that the detection quality of a combination of detectors is improved when compared to the results of isolated detectors.

Table of Contents

List of Figures

List of Tables

List of Listings

1. Introduction

The information era has come to stay. People depend nowadays more on technological devices to perform daily life tasks, and this is far from changing. “There will be 12.3 billion mobile-connected devices by 2022, including Machine-to-Machine (M2M) modules—exceeding the world’s projected population at that time (8 billion) by one and a half times” [16]. From the moment devices became available to everybody, Human-Computer-Interaction (HCI) became vital to keep this human-computer dependency growth. HCI studies the interactions between humans and computers, and researches the development and improvement of usability, efficiency, effectiveness, utility, and safety of systems including machines [23]. The HCI discipline has had a strong influence on how user interfaces have been designed and have evolved across time.

1.1. Current Situation

UIs are one of the most critical aspects of human-computer-interactions as they constitute the realization of the mechanism with which users interact with machines. UIs are the only means in which humans perceive a system or device. It is hence crucial to design and build high-quality UIs since it influences the users’ perception of the whole system. Designers and engineers often follow accessible and well-researched guidelines and principles to create high-quality interfaces. Several toolsets, such as User Interface Management Systems (UIMs) have been created to assist in the implementation of UIs, aiming to decrease the time required to design and develop UIs. Nevertheless, these approaches are still not enough to ensure high-quality UIs.

The quality of a UI is measured through a so-called UI Analysis, which consists of assessing quality aspects of UIs, such as, among others, *Usability* and *Aesthetics*.

Usability is the extent to which users can use a UI in order to complete a task with efficiency [72]. This aspect is an indicator of how fast users can become familiar with the interface, how easily they achieve their desired outcomes, and whether they like the visual and procedural aspects. The better the Usability of a UI, the more likely it is that users learn to use it and adapt to it, and the more likely they get engaged. A decrease in the users’ desire to use an application again can lead to business loses.

Aesthetics is a term with a philosophical background that refers to understanding the beauty of things and how it is perceived [75]. In the context of UIs, aesthetics is a measurement of how appealing the UI is to users. Users prefer utilizing software with attractive interfaces and feeling joy while doing it. Good aesthetics may impact current and future usage and user opinions positively.

Evaluating these previously mentioned quality aspects of UIs is not a trivial task. For instance, aspects such as Aesthetics can sound subjective on their own, and Usability can depend on several other factors. Therefore, it is necessary to come up with more systematic and objective evaluation approaches.

Several studies propose systematic UIs analysis. For example, Zen and Vanderdonckt proposed an evaluation method of Aesthetics of Graphical User Interfaces (GUIs) based on metrics. First, they defined a model that captures aesthetics metrics. Later, they designed an evaluation for GUI aesthetics, based on the previously defined model, and finally, developed an implementation of the model as a Web Service [97].

A similar study by Miniukovich and De Angeli focused on searching for automatic metrics of GUIs [68], centering mainly on five determinants of visual complexity and aesthetics: clutter, symmetry, contour density, figure-ground contrast, and color variability. They were able to find correlations between their automatic computed metrics, and a survey conducted on real users asking them for their subjective perceptions about aesthetics and visual complexity of several UI examples. In a more recent study, Oulasvirta et al. developed a service called Aalto Interface Metrics (AIM), which allowed the computation of 17 different metrics under the categories of color perception, perceptual fluency, visual guidance, and accessibility [74].

There are several *Metrics* highly correlated with the quality aspects of UIs, for instance, Visual Complexity, which refers to the amount of detail contained in a UI. This metric directly affects the perception of UI elements. In a complex UI, it is more difficult for users to comprehend the UI functionality. Visual complexity increases muscle tension, cognitive load, and decreases interaction pleasure in users [69]. Besides, this metric has an inverse-proportional relation to the aesthetics of a UI [68]. Nevertheless, there are other aspects than visual complexity that also affect aesthetics.

In the context of the Web, users already have a long-lasting opinion about a web page within seconds of seeing it for the first time [82]. Colorfulness, visual appeal, and visual complexity are among the characteristics of a website that cause stronger impressions at first sight.

There exist a few different categorizations of UI metrics, described next.

Metrics can be categorized as *atomic* or *composite.* Atomic metrics are independent of other metrics; while the measurement of composite metrics depends on other atomic or composite metrics. Miniukovich and De Angeli measure complexity using several composite metrics identified through aspects, such as amount, organization, and the discriminability of information [68]. Under information amount, atomic metrics such as visual clutter and color variability were more prominent. Color variability represents the number of different colors present in the UI. Visual clutter refers to the state in which excess items, or their representation or organization, lead to a degradation of performance at some task and crowding [84]. Higher marks on the visual clutter and color variability measurements lead to higher UI complexities.

Several sources established a distinction of UI metrics into *Visual UI metrics* and *Structural metrics* [97]. The former ones rely on everything related to all the visual aspects of UI objects, while the latter ones rely particularly on the structural code and UI architecture.

Besides these distinctions, other subsets of metrics are established: *object-independent* and *object-dependent* UI metrics.

On one hand, object-independent UI metrics do not depend on the UI objects. An example of this kind of metrics is color variability, calculated by counting the number of dominant colors in the UI. On the other hand, object-dependent metrics rely on UI parts. Such metrics can also be visual UI metrics, which means that their calculation uses visual cues. Some of this type of metrics are, for instance, UI white-space region and UI density. In order to compute the white-space region of a UI, it is important to know which areas of the UI are free of elements, so it implies knowing where the elements are. The same happens for calculating the density of a UI. Computing density demands to know the area of the whole UI occupied by the different elements.

This research is particularly interested in object-dependent metrics that are also visual. Identifying UI parts for computing object-dependent metrics requires both UI elements classification and determining their locations within the UI. From this point, such identification is called *UI Object Detection*: the application of object detection techniques in the specific domain of UIs. The detectable UI object types are all the possible atomic, e.g., buttons and input text boxes; and composite blocks, e.g., navigation bars and pagination controls. The UI object detection process takes a representation of a UI as input, e.g. a screenshot of a rendered UI or a web page's DOM, and outputs the UI objects that it contains with their respective locations.

There are two different approaches that can be followed for applying UI object detection: *Naive* and *Automated* approaches, which can be further classified into *Parser-based* and *Vision-based.*

The naive approach consists of an empirical assessment performed by gathering well-trained and experienced people in order to make them identify the parts of the UI. The latter one does not require any human intervention in the identification of the parts of the UI.

Concerning automated approaches, parser-based techniques perform *page segmentation* using UI's underlying structure. Page segmentation is "the process of dividing a page into visually and semantically coherent segments called *blocks*" [85]. Based on their complexity, blocks are *atomic* or *composite*. Atomic blocks are indivisible page elements. One cannot obtain smaller functional and semantically coherent sub-elements. On the other hand, Composite Blocks are divisible. One can obtain either atomic or smaller composite sub-blocks with visual and semantic coherence.

In this research, page segmentation is considered an object detection mechanism in the sense that it identifies composite blocks, meaning it recognizes UI blocks to a higher level aiming for semantic elements, while object detection focuses on the atomic blocks, i.e., syntactic elements, although it is not limited to identifying only these as it can also detect composite ones as well. In the context of web applications, the input of a parser-based UI object detection process is the DOM tree. Atomic blocks generally refer to Hypertext Markup Language (HTML) tags but are not limited to them, e.g. buttons, links, and input text boxes; while composite blocks are groupings of atomic blocks, such as navigation bars, which may contain a series of links or buttons; or pagination controls, which may include a few buttons and text.

There are a few parser-based implementations out there. For example, the graph-theoretic algorithm provides segmentation using the structure and properties of the DOM tree, as well as the relationship between its nodes [15]. The Bento Page Segmentation algorithm generates a new DOM tree for cleaner and more consistent page segmentation using a four-step process [53]. Sanoja and Gançarski proposed Block-o-Matic (BoM), which uses an improved version of the Bento algorithm, in combination with heuristic rules defined by the World Wide Web Consortium (W3C) Web Standards in order to produce a labeled segmentation [87].

Vision-based techniques, on the other hand, rely on the visual representation of the UI, rather than on its underlying structure, allowing computers to understand the semantic structure and detect objects of a UI similar to how a human does. These techniques take advantage of computer vision and image processing in order to segment UIs accurately and detect atomic and composite blocks within. The process generally has two significant steps: computing the locations and sizes of rectangular blocks containing relevant objects, and determining for each block, whether it belongs to a semantic object type and which one.

The Vision-based Page Segmentation (VIPS) algorithm is a hybrid approach for extracting the semantic structure of UI since it utilizes the DOM tree of a web page, as well as its visual cues. The process first segments the page by recursively extracting all relevant blocks from the DOM tree and finding optimized division lines that visually separate them. The divisions are used then to describe the semantic structure [13].

The traditional vision-based object detection techniques consist of detecting instances of semantic classes, i.e., the location and the scale of each one, in an input image, given a predefined set of object classes [92]. These objects can be of any semantic class, e.g., a person, a car, a dog. The object detection process studied in this research takes an image as input, and from this, it computes a series of locations of rectangular boxes called Region of Interests (RoIs) where there is a high probability of the existence of an object. The process classifies each region in order to determine whether it contains an object and which one. Additionally, the model outputs also a confidence score that represents how positive the model is that the predicted regions contain an object and how accurate the model thinks the predicted semantic class is [79].

For RoI extraction and classification tasks, implementations use a few different techniques. The classification task is particularly interesting because of the variety of widely-used techniques, e.g., Support Vector Machines (SVMs) and Neural Networks; and types of classifiers, i.e., Binary and Multi-Class classifiers. Choosing a classifier that performs outstandingly in the context of its use case impacts the system results.

Machine Learning addresses object detection by performing *feature engineering*, i.e., identifying and extracting features for further usage in the learning and prediction phases. An instance is the Scale Invariant Feature Transformation (SIFT) algorithm, is capable of generating image features invariant to properties such as translation, rotation, scaling, and partially invariant to illumination and three-dimensional projection [62]. Histogram of Oriented Gradients (HOG) is another feature extractor of images for object recognition, based on the evaluation of well-normalized local histograms of image gradient orientations in a dense grid [19]. The Viola-Jones object detection framework specializes in face-detection tasks and [93].

With the major improvements in computer hardware in recent years, computational power is now no longer such a limitation as it was before, and therefore, as pointed by LeCun et al., deep learning has become a more feasible field of study for addressing, among others, visual object recognition and object detection tasks [55]. There are several deep learning implementations for solving these kinds of tasks, the most common approach is using Convolutional Neural Networks (CNNs), since their architecture allows end-to-end object detection, progressively extracting and learning complex features and RoIs from images, as well as classification of these regions. Current implementations that use CNNs can be grouped into *Region-based* and *Real-time* oriented techniques.

Region-based CNNs [60] generally follow three steps: computing region proposals, extracting features from these regions, and classifying them for labeling. Over the years, new, improved versions of this architecture arose. R-CNN [32] was the first improvement in this context, which integrates AlexNet with a selective-search region proposal algorithm. SPPNet [39] combines the traditional Spatial Pyramid Pooling (SPP) with a CNN architecture, obtaining speed improvements while keeping the detection quality. Fast R-CNN [31] and Faster R-CNN [83] were designed to overcome some of the limitations of R-CNN and SPPNet and simultaneously increase the detection quality.

Real-time oriented techniques work under the premise that region proposals tend to be slow and hard to optimize. Implementations such as You Only Look Once (YOLO) [79] and Single Shot Multibox Detector (SSD) came into the picture. YOLO uses a single neural network capable of predicting both bounding boxes and class probabilities directly from raw images in one single evaluation. Since its unified architecture is high-speed, it can process images in real-time at 45 frames per second. SSD can improve speed and accuracy by adding some improvements over existing approaches such as YOLO [61]. Improvements included: using a small convolutional filter to predict categories and offsets in bounding box locations, using separate filters for different aspect ratio detections, and applying them to multiple feature maps from the later stages of a network in order to perform detection at multiple scales.

Evaluating the detection quality of vision- and parser-based approaches requires counting with an input testing dataset, i.e., DOM trees or rendered UI screenshots, which need to be manually processed by experts who determine blocks and objects, and label them accordingly. Manual answers are aggregated using a technique such as a majority vote in order to compose the ground truth for the test dataset. For evaluating detection results, one compares the system result to the ground truth annotations for the dataset.

The most widely used metrics for evaluating the detection quality of vision- and parser-based approaches are *Precision* and *Recall* [45]. Precision represents how many positive predictions were also positive according to the ground truth from all the classifications performed by a system. On the other hand, Recall represents how many predictions were correctly classified by the system from all the positive examples according to the ground truth or test dataset.

The importance of Precision and Recall can vary depending on the application scenario. In some cases, one metric is more critical than the other, while in some others, a balance between them is the correct choice. One example scenario that can require high importance in Recall is cancer detection. The identification and treatment of all patients with cancer are critical. Treating people without the disease is less critical than not treating people who have it.

In the context of UI object detection, it is not sufficient to spot all the possible UI objects. It also requires that the set of identified UI is correctly labeled. For this reason, it is necessary to have an appropriate balance between Precision and Recall.

1.2. Motivation

Existing UI object detection approaches have individual strengths and weaknesses; some detectors perform well in particular scenarios and with specific object types, whereas others perform better in different conditions, meaning that a single approach with outstanding detection results in all cases does not exist. This formulation leads to the idea of finding a suitable way of combining diverse approaches, and as a consequence, improve the overall quality of UI object detection. Such improvement benefits the further computation of low-level object-dependent metrics, because having more precisely detected UI parts, leads to a more precise calculation of object-dependent metrics. Furthermore, it allows making more accurate statements about quality aspects correlated to such metrics.

There are several use-case scenarios, in which, knowing the UI parts is useful, e.g., (1) Web Migration, (2) making statements about visual aesthetics and visual complexity, (3) usability evaluation, (4) Web Data Extraction, and (5) UI Similarity Assessment.

In the field of Web Migration, finding all the general properties of a UI layout to describe the structure of web pages is required. Sanoja uses the segmentation to extract relevant blocks from an HTML4 page in order to migrate it into a new version with the same structure and content but using the HTML5 format. Segmentation results help to produce metrics related to the similarity between the previous and new page versions [86].

Making statements about visual aesthetics and visual complexity requires understanding a page's layout [75]. Measures for these characteristics include, among others: balance, symmetry, and density, whose values can be computed using information about the sizes and locations of UI objects. Visual complexity is probably the principal factor for appeal perceived by users [82]. Complexity is essential for making statements about comprehensibility and accessibility. Some complexity measurements include the number of objects, the density, visual clutter, the space used by text and images, and the layout. The calculation of these measurements requires knowing the visual structure as well as the UI objects and their locations.

Usability evaluation involves describing: content organization, and again, page layout, as well as striving for a minimalistic design, i.e., the reduction of complexity and unnecessary information [77]. Zhai and Liu study the problem of web data extraction, for which they propose a two-stage strategy that requires, in its first stage, understanding the layout and the objects contained, in order to map useful information contained in the page into the predefined database structure for storage (or archiving) [98].

As UIs evolve rapidly due to constant changes in user requirements and evolving technologies, a critical use case for UI object detection is the UI similarity assessment. Drastic changes can affect user-adoption negatively, and therefore, it is necessary to have ways for measuring the degree of change in UIs, achieved through an evaluation of similarity.

1.3. Problem Description

The circumstances described previously enable the discovery of current problems still to be addressed. These problems belong to one of two categories. The former category emphasizes on solving challenges concerning UI object detection using DNNs, while the latter focuses on the challenges that arise from the desire of combining UI object detectors. The following list offers categorization details:

- Problems Related to UI Object Detection Using DNNs
  - **PA1.** How to obtain a sufficient training dataset in terms of magnitude and frequency distributions of training examples across each of the classes, as required by DNNs in order to perform well in object detection.
  - **PA2.** How to avoid the computationally expensive task of re-training DNNs with big training datasets that grow over time.
- Problems Related to Combining UI Object Detection Analyzers
  - **PB1.** How to support multiple implementations of UI object detectors.
  - **PB2.** How to aggregate results from the outputs of different object detectors into one single output.
  - **PB3.** How to resolve conflicts among the outputs of different object detectors.

- **PB4.** How to assign a weight to object detectors towards aggregating results.

For deeper analyzing these problems, the starting point is describing the advantages and drawbacks of the existing UI object detection approaches.

Naive object detection techniques are useful when the amount of UIs to evaluate low since it requires fewer people and less time and training. However, this is rarely the case, as the internet keeps growing by the day and new mobile, desktop, and web applications come to the stage; Consequently, in most scenarios where a plethora of UIs must undergo analysis, it is both an expensive and a slow alternative.

Parser-based techniques are automated approaches that provide a faster mechanism for identifying UI parts, compared to manual approaches. They generally run well and are potentially more complete as they can output a full description of semantic classes. Besides, in UIs described in a standardized manner where all elements of the same type share the same description and semantics, parser-based approaches can identify a high percentage of the UI parts.

Parser-based techniques require a semantic representation of the UI. Representations of desktop applications demand knowledge about the description language in which the UI is built, introducing a limitation, as it is not always possible to obtain such representation.

For Web UIs, counting with the DOM tree is mandatory. The tree is much easier to obtain compared to obtaining the structure of desktop applications. However, Web UIs do not necessarily describe their semantics following internationally recognized standards such as HTML5, proposed by the W3C[1]. For this reason, Web UIs can have different possible HTML-tag combinations for building similar UI parts, and it is also difficult to predict how different web browsers in different platforms render HTML and Cascading Style Sheet (CSS) code.

Moreover, the *Web Components*[2] meta-specification allows defining so-called *Custom Elements*, which are in essence, encapsulated extensions of standard HTML tags with personalized visual and behavioral aspects, that can be plugged into and reused in web applications. Web Components add a new complexity layer to the DOM tree of web UIs. All the factors above introduce high web UI heterogeneity, and this is a significant limitation for parser-based techniques in the context of the web.

Vision-based techniques are faster than manual approaches and provide a perception of UIs close to the one humans have. Visual techniques overcome the limitation of UI

[1]https://www.w3.org/

[2]https://www.webcomponents.org/specs

heterogeneity. Furthermore, they provide features for more straightforward evaluation, such as, among others, the absence of noise, glare, and differences in lighting, a perfect angle of view, the lack of motion and covered objects [5]. However, in contrast to parser-based techniques, the outputs of vision-based techniques involve an error probability, meaning that complete certainty on the detections is not guaranteed.

Problems Related to UI Object Detection Using DNNs

Vision-based approaches make part of a younger discipline. These approaches have not been broadly applied UI analysis until recently. Deep learning has gained an outstanding momentum in object detection and has become State of the Art (SotA) for addressing the detection of real-world objects. Implementing generic object detection techniques in the domain of UIs demands the following aspects.

The first aspect **PA1** is determining how to obtain a sufficient training dataset in terms of magnitude and frequency distributions of training examples across each of the classes, as required by DNNs in order to perform well in object detection. Finding ways in order to gather a big training dataset is mandatory, having UI Object Detection by using DNNs as one of the main subpaths of this research.

A significant training dataset is one of the main prerequisites for succeeding in object detection through deep learning approaches. Besides, dataset storage aspects are crucial. Counting with a sufficient training dataset is an advantage, but it is still not enough for achieving good results with DNNs.

The second challenge **PA2** is how to avoid the computationally expensive task of re-training DNNs with big training datasets that grow over time. Training an object detection model with a big training dataset may be a challenging task since it requires high computational power. Therefore it is vital to implement a learning approach in which the model can learn progressively from the training dataset as it grows in size.

Also, input images from real-world objects have notorious differences compared to images of user interfaces. Being A a set of images that contain real-world objects in their usual environments, and B a set of screenshots of UIs, By comparing A and B, one can note that generally, the complexity in terms of noisy backgrounds, clutter and color variability may get considerably higher in A than in B; this is an advantage in favor of object detection for B since there are less noise and fewer distractions for detecting objects in this set.

However, for most of the cases in A, the similarity between objects of different classes has high variance, while in B, certain elements become very similar to each other. For instance, a pedestrian is very different from a cat in A, while in B, the difference between a button and an input text may not be evident in certain situations. These differences between A and B generate a tradeoff that may have a positive/negative impact on the results.

Since an essential part of this thesis is to make particular emphasis on visual-based approaches in the field of object detection using DNNs since its results have excelled in broader domains, it becomes relevant to evaluate its applicability to the specific domain of UI Object Detection. From this necessity, one can state the first thesis hypothesis:

Hypothesis A: The quality results provided by deep learning approaches in generic object detection can be replicated in the specific domain of UI object detection.

Problems Related to Combining UI Object Detection Analyzers

Parser-based and vision-based techniques for UI object detection have advantages and limitations for particular use cases. Some of these implementations follow processes composed by several separable tasks, i.e., segmentation, RoIs extraction, classification; while other implementations provide end-to-end solutions.

Some approaches perform better at one particular task, while some end-to-end implementations show good results in specific contexts. There is not one particular approach that guarantees optimal results in all scenarios, and even though it is possible to optimize one technique for the entire detection process, one misses the benefits from other techniques. From this, the second thesis hypothesis arises.

Hypothesis B: The combination of different UI object detectors and techniques improves the quality of object detection results in the domain of UIs compared to the quality results of isolated techniques.

This work is interested in taking advantage of a variety of implementations for specific tasks that are known to output better results in particular scenarios, as well as the benefits of end-to-end implementations, and being able to combine them. Accomplishing this requires therefore solving the challenge **PB1** of how to support multiple implementations of UI object detectors. The combination is intended to be done at definition time, i.e., defining a UI object detection process using multiple detectors.

Combining parser- and vision-based approaches introduces one crucial step, which is defining a reference process description that can fit all of these techniques and their tasks generically to a certain extent. Having defined such generic UI object detection process, executing with an input testing dataset is desired.

The execution phase of the process above should be able to aggregate results of the different configured task-divided and end-to-end implementations into one single system output. Consequently, a second challenge **PB2** questions how to aggregate results from the outputs of different object detectors into one single output. One can answer PB2 by tackling the two following derived subproblems.

By the nature of the different algorithms and techniques for UI object detection, it is possible that their results, i.e., RoIs, class labels, and confidence scores, differ. In order to have a unified system output, it is necessary to resolve conflicts that may occur in the domains of RoIs extraction and classification. **PB3** analyzes how to resolve conflicts among the outputs of different detectors.

Furthermore, as the intention of this thesis is to join several implementations within a single UI object detection process, a good results aggregation depends additionally on the weight distribution among the different configured providers within the custom process. Hence, another derived problem arises. **PB4** reviews how to assign a weight to detectors.

1.4. Objective Statement

This research has two main objectives, one for each problems category. Objective A describes the thesis intention in relation to Problems Related to UI Object Detection Using DNNs, while Objective B emphasizes on the ambition to overcome the Problems Related to Combining UI Object Detection Analyzers.

Objective A

Conceptualize a DNN-based UI object detector that can learn progressively using sufficient and continuously-growing training datasets.

The previous section exposes the necessity for detection of UI parts. Therefore, one of the aims of this thesis is the elaboration of a concept that considers current approaches that excel at generic object detection in order to apply them to UI object detection. Current approaches state within deep learning thanks to the advances in computational resources, algorithms, and an increase of labeled data.

However, in the context of UIs, gathering a sufficient dataset is still needed. Sufficiency is measured using the size and frequency distribution of training examples across all the possible object types. In order to gather a sufficient training dataset, several generic techniques are possible, e.g., crowdsourcing, synthetical generation of data, automatic data labeling, and data augmentation. Therefore, another aim of this part of the thesis is the acquisition of a sufficient training dataset, implementing one or several of these means.

Having a sufficient training dataset is essential for training deep-learning-based object detectors; nevertheless, it also has counterparts. Considering a sufficient dataset from the very beginning, that is subject to growth over time, it becomes necessary that the solution learns progressively instead of only being able to learn from scratch.

The elaboration of a theoretical approach -or combination of approaches- for addressing the stated aims is seasoned with a prototype. This work describes the prototype feasibility and implementation aspects.

Objective B

Conceptualize an integration platform for vision- and parser-based UI object detectors to improve detection quality.

Based on the stated problems related to Combining UI Object Detection Analyzers, the second thesis objective looks after the elaboration of an approach that dynamically integrates existing distributed and independently-developed object detection providers into single object detection processes, aiming at the improvement of detection results compared to the results of the providers measured in isolation.

The approach shall execute a full UI object detection process using existing providers, automatically handling inter-provider communication, i.e., arranging the process execution without human intervention. The designed platform approach shall guarantee detectors interoperability, meaning that it must support a wide variety of detectors that have different architectures and implementation algorithms, by implementing a generic UI object detection reference process, which allows subsequently determining suitable provider communication interfaces.

A prototype implementation is to be developed to test the concept. This work describes the prototype feasibility and implementation aspects.

1.5. Document Structure

The remainder content of this thesis is structured as follows. First, chapter 2 elaborates on a set of requirements demanded by an ideal solution that addresses the problems stated in this chapter. Subsequently, in chapter 3, state-of-the-art techniques are grouped, analyzed, and assessed against the requirements to determine the degree to which they can contribute to a suitable solution. Based on this analysis and assessment, chapter 4 proposes possible solutions, chooses the most suitable, and lastly, creates an abstract draft of it. Next, chapter 5 covers more in-depth the core implementation aspects for building a prototype that follows the outlined design. In chapter 6, the resulting prototype is evaluated against the thesis requirements in order to test its conformance. Lastly, chapter 7 concludes the findings of this thesis and proposes future research work.

2. Requirements Analysis

This chapter analyzes requirements derived from the thesis objectives defined in the previous chapter. Since each objective belongs to a single solution part in this work, they are analyzed individually. section 2.1 exposes requirements regarding UI Object Detection Using DNNs, and section 2.2 examins requirements concerning Combining UI Object Detection Analyzers .

2.1. UI Object Detection Using DNNs

Part written by Juan Sebastian Quintero

As already mentioned in Objective A, an essential part of this work is finding a solution that allows the visual analysis of UIs using the SotA in regards to DNNs for generic object detection. Coming up with a successful solution that enables testing the *Hypothesis A* depends on the fulfillment of several requirements across different aspects. Such requirements can be subdivided as process-related requirements and result-related requirements. Process-related requirements correspond to those regarding how the process must be addressed while result-related requirements are the ones referring to the quality of the results obtained.

In this case, process-related requirements are further subdivided into the following groups:

- Object Detection Requirements
- Learning Requirements

The following subsections include more detailed information about each of the requirement types introduced.

2.1.1. Object Detection Requirements

The Object Detection requirements are those that outline both functional and non-functional aspects related to the object detection process. The following list contains the requirements that are identified within this group:

- RA1. Detection of the Parts of a UI
- RA2. Automation of Object Detection
- RA3. Visual Input
- RA4. Deep Learning

The subsequent paragraphs contain a detailed description of each of the identified requirements at this matter.

RA1. Detection of the Parts of a UI

Analyzing UIs depends on the computation of different metrics. Calculating most of these metrics require knowing each of the objects that make part of the UI. For this reason, one main functional requirement is identified: The support for the *detection* of the elements that comprise a UI. In this context, detection refers to the process that includes the localization of potential UI objects by enclosing them with a bounding box, and their respective classification according to several predefined types.

There are mainly two categories of UI objects that are identified, atomic objects and composite objects. The former comprises objects that cannot be divided nor contain other objects, e.g., buttons, checkboxes, labels. The latter ones comprise objects that act as containers of other composite or atomic objects. In addition, there also exists a distinction between UI objects that relate to Optical Character Recogniton (OCR) e.g. paragraphs, text, text-blocks and the rest. A detailed description of such information can be found at Table B.1; this table describes 20 different UI object types from which 17 are non-OCR-related and 3 are OCR-related. For this case, only those that are non-OCR-related are taken into consideration. Approaches must be assessed in terms of its support to detect (both classify and locate) such UI object types. Hence, this requirement is considered as:

- **Fully satisfied** if the detection of the 17 non-OCR-related UI object types is supported by the assessed approach.
- **Partially satisfied** if the detection of at least 9 of the non-OCR-related UI object types is supported by the assessed approach.
- **Not satisfied** otherwise.

RA2. Automation of Object Detection

Object detection within UIs through a manual approach is expensive in several aspects such as time, effort, and monetary-expenses. It constitutes a relatively easy job for humans; however, performing manually such task does not scale up for potentially thousands of UIs. Thanks to the advances in machine learning and its usefulness in computer vision problems, machines have started to achieve better results than humans at this task [3], and at the same time solving the scalability problem mentioned above.

For assessing this requirement, only the degree of automation on the prediction stage (classification and localization performed on unlabeled data) of the object detection process is taken into consideration. Consequently, other phases of the overall process, such as the training stage are out of the scope.

The prediction stage is:

- Manual, if there is human intervention present across any of its substages.
- Semi-automated, if any of its substages requires human intervention.
- Automated, if No human intervention is required at all.

Based on the criteria defined above, this requirement is considered as:

- **Fully satisfied** if the prediction stage of the assessed approach is "automated"
- **Partially satisfied** if the prediction stage of the assessed approach is "semi-automated"
- **Not satisfied** if the prediction stage of the assessed approach is "manual"

RA3. Visual Input

An essential aspect of this part of the thesis is the detection of UI Objects on a visual basis. Based on that, a primary requirement is to be able to perform such analysis having screenshots of UIs as the only source/input.

The definition of the requirement relies on having the ability to mimic how human beings perform the detection of objects. Object detection is a relatively easy task for humans, and several studies have been conducted to decipher how they do it [22]. The most prominent approaches have been inspired in the way humans perform this task [3]. Besides, being human-like, allows the UI analysis to be implementation agnostic, which means that its applicability goes beyond the technologies in which the UIs are created. In the case of WUIs, factors such as the HTML structure and tags used will not matter at all in the results given by the UI object detector. The same applies to general GUIs in terms of the programming languages, frameworks, or libraries. For such reasons, a similar approach in which only visual aspects of the UI are considered is desired.

Based on the previous statements, the assessment of approaches in this regard can take the following values:

- **Fully satisfied** if the approach is capable of accepting the visual representation of a UI e.g. a screenshot of it.
- **Partially satisfied** if the approach does accept non-visual representations of a UI from which visual aspects can be inferred e.g. DOM.
- **Not satisfied** otherwise

RA4. Deep Learning

Deep learning in the general area of object detection has shown to be successful thanks to both advances in hardware resources and the emergence of a set of approaches that make the learning process from raw images feasible with CNNs. For that reason, in this work, it is wanted to investigate the applicability in the specific domain that this work researches (see *Hypothesis A*).

Deep Learning is describing the procedure of performing machine learning tasks with deep artificial neural networks [34]. The term deep refers to the number of layers that comprise a Artificial Neural Network (ANN) [3]. In contrast to a shallow network, A network is considered to be deep when there is more than one hidden layer between its input and output layers.

As already defined in RA2, the ideal approach should be able to have UI screenshots as its only source of input to detect objects in a UI. According to [3], CNNs are a type of ANN designed specially to work with images and they comprise the SotA regarding generic object detection. That is why particular attention should be put on those kinds of networks. A CNN that follows a deep architecture is called to be a Deep Convolutional Neural Network (DCNN).

Furthermore, just like with traditional machine learning, in deep learning, there is also unsupervised learning and supervised learning. For the general area of object detection, supervised learning is typically used. Therefore the ideal solution in this thesis should be cataloged within supervised learning.

This requirement is considered to be:

- **Fully satisfied** if the assessed approach makes use of a DCNN with an internal architecture that is usable in supervised learning problems, and applicable to the object detection domain.
- **Partially satisfied** if the approach is usable in supervised learning problems and also applicable to solve object detection problems, but without being deep-learning based.
- **Not satisfied** otherwise

2.1.2. Learning Requirements

In contrast to the object detection requirements, learning requirements are merely non-functional aspects that outline what the ideal solution should consider from a learning/training perspective. Such requirements should include training dataset considerations as well as learning approaches to be applied. The following list contains the identified requirements:

- RA5. Sufficient Training Dataset Size
- RA6. Sufficient Training Dataset Frequency of Distribution
- RA7. Training Dataset Collection Efficiency
- RA8. Avoid Re-training from Scratch

Those requirements are illustrated in more detail in the following paragraphs.

RA5. Sufficient Training Dataset Size

As stated in the requirement RA4, it is desired to try deep learning approaches because, in the general area of object detection, they have shown to be successful. Nevertheless, it is known from other domains that for achieving satisfactory results, deep learning approaches require significant training datasets in terms of size.

The size of the training dataset is an essential aspect since this constitutes a significant fact on the training process of a model implemented through DNNs. That is why establishing a sufficient training dataset from the very beginning is critical since it is desired that the detection of UI objects gives acceptable results. There are existing training datasets that are widely used in the context of generic object detection [58, 27]. The Microsoft Common Objects in Context (COCO) training dataset registers around 2.5 million objects labeled from 91 different classes across 328K images [58]. Also, the Pascal Visual Object Classes (VOC) training dataset consists of 27450 labeled objects in 11530 images with 20 different classes [29].

In this context, a training dataset comprised of images and labeled objects of the different classes described at RA1 with sufficient training examples in terms of size is required. The requirement is considered as:

- **Fully satisfied** if the assessed approach can gather a training dataset that covers all the classes described at RA1 and has similar properties in the order of size compared to the Pascal VOC dataset.
- **Partially satisfied** if the assessed approach can gather a training dataset that covers at least half of the classes described at RA1 and shares proportional properties in the order of size compared to the Pascal VOC dataset
- **Not satisfied** otherwise

RA6. Sufficient Training Dataset Frequency of Distribution

The criteria specified at RA5 demands sufficient rates for the training dataset in terms of size. The site of the dataset generally speaking is essential, but having an adequate set of examples representing each of the classes that model predicts is more critical. That is why it is also necessary to consider the frequency of distribution of the training examples across the different classes that make part of them.

Randomly selecting UIs for further labeling, is not enough because the distribution of examples per class can end up being unbalanced. The frequency distributions of the training dataset are also crucial if an object detector capable of predicting all the UI Objects is desired. While gathering the training dataset, it is vital to have mechanisms in place that ensure minimum frequency thresholds of examples across each of the classes that the object detector supports. Some classes need a higher frequency than others to have a sufficient number of examples that allow the object detector to learn the complexity of their patterns. In the UI domain, something similar can happen with elements that are declared to be composite, e.g., a panel, or pagination versus atomic elements, e.g., a button.

In this context, a training dataset comprised of images and labeled objects of the different classes described at RA1 with sufficient training examples in terms of frequency distributions across classes is required. The requirement is considered as:

- **Fully satisfied** if the assessed approach is capable of gathering a training dataset that covers all the classes described at RA1 and has similar properties in the order of frequency distribution across classes compared to the Pascal VOC dataset.
- **Partially satisfied** if the assessed approach is capable of gathering ad training dataset that covers at least half of the classes described at RA1 and shares proportional properties in the order of frequency distribution across the supported classes compared to the Pascal VOC dataset
- **Not satisfied** otherwise

RA7. Training Dataset Collection Efficiency

Finding an efficient way of gathering labeled data that constitutes the training dataset is critical. Said that it is crucial to evaluate the efficiency that different approaches for the acquisition of the training dataset require.

For this requirement, efficiency is scoped in terms of costs and time that is required to gather a training dataset with sufficient size and frequency of distribution of training examples across classes. For such reason this requirement is highly related to the requirements RA5 and RA6.

Both costs and time are related to a task that human and computers perform in the training dataset collection process. Nevertheless, the scope of such efforts is only at a human level since it is found that this fact is the main bottleneck in terms of efficiency. Some of the approaches require human efforts at the collection and labeling phases directly. Some others require such attempt at a definition of the process phase. Other procedures require it at more earlier stages such as finding existing suitable training datasets. Besides, the level of human effort varies per approach, so it is vital to minimize the level of energy that is needed.

Considering what was mentioned above, the requirement is assessed as:

- **Fully satisfied** if the approach allows gathering a sufficient training dataset within the timeframes required for this research by having just one single person in charge of the process.
- **Not satisfied** otherwise

RA8. Avoid Re-training from Scratch

Finding a suitable learning approach is a requirement derived from RA5 since learning a model from a training dataset with a considerable size is a computationally expensive task.

Besides that, one must consider that the initial training dataset created is not necessarily a perfect one at the beginning since it can be collected by using different methods. Some methods can imply input from people (crowdsourcing), and others can also involve the generation of labeled data synthetically. Regardless of the methods used for collecting the training dataset, it is destined to grow across time and demanding the execution of new relearning processes. Therefore, it becomes an additional challenge because learning from scratch each time new labeled data is added to the training dataset increases progressively computational costs.

To overcome such problems, a learning approach that scales well on large datasets that grow in size has to be considered. The classical *Batch Learning* consists of updating the weights of the model after calculating the "loss" using the entire dataset. This approach is not well suited for large dataset because it requires all the training examples to be loaded into memory and can surpass memory limits. For addressing such problem *Online Learning* is pointed out to be a more suitable learning method at this regard [9]. In contrast to a batch learning approach, an online learning approach allows the model to update its weights after calculating a "loss" against one single training example. A more generalized way that includes both approaches is called *Mini-Batch Learning*; With this approach, there is an additional parameter to consider: the *batch size*. If the batch size is equal to 1, it is considered online learning; if the batch size is equal to the dataset size, it is batch learning. Additionally, intermediate values for the batch size can also be used.

Using mini-batch learning solves the problem of executing one training iteration over a large dataset; however, it is still required to count on a technique that allows learning over different training iterations from different dataset deltas, i.e., in a progressive manner. A progressive training process can still be challenging since it requires a lot of intermediate actions such as dataset collection, dataset transformation, model assets preparation, training process initialization, models export, among others. That it is why it is desired to have a mechanism that automates those intermediate steps allowing a cleaner and more natural progressive training process.

A rating is given to a certain approach by checking its support for training progressively. The requirement is considered as:

- **Fully satisfied** if the approach supports being trained progressively and encapsulates all of its related intermediate actions.

-

- **Fully satisfied** if the approach supports being trained progressively but does not encapsulate its related intermediate actions.

- **Not satisfied** otherwise.

2.1.3. Result-related Requirements

The result-related requirements are those that define the evaluation criteria expected from the execution of the process outlined by the process-related requirements. In this thesis, the main objective is clear: Detect different types of objects across UIs. 17 types have been defined (see Table B.1). Being able to detect them is one thing, but it is important to do it with good precision and recall rates so the detections can be considered. Just like precision, the recall is another key aspect to measure the successfulness of an approach. Therefore, two requirements arise from this analysis:

- RA9. High Precision

- RA10. High Recall

The following subsections explain each of the requirements identified in this regard.

RA9. High Precision

Analyzing UIs with the help of metrics requires that those metrics are calculated at first; computing most of these metrics require knowing each of the objects that make part of the UI (section 1.1). A low precision on the detection of UI objects implies a wrong calculation of metrics. That is why it must be considered achieving sufficient precision rates for having more accurate metrics.

Measuring precision is achieved by applying the following formula

$$R = \frac{TP}{TP + FP}$$

where TP are the true positives, and FP correspond to the false positives that result from the comparison between the object detector outputs and the ground truth (test dataset).

This requirement is cataloged as:

- **Fully satisfied** if the approach reaches a sufficient precision rate in the UI object detection domain.
- **Partially satisfied** if the approach reaches at least the half of a sufficient precision rate in the UI object detection domain.
- **Not satisfied** otherwise

RA10. High Recall

For the same reason as with precision, the recall must also be considered equally. A high recall implies the inclusion of relevant (positive) UI objects from the ground truth with high confidence. It means that an object detection approach can classify and locate buttons, or text-boxes from UIs with a sufficient rate.

Both precision and recall are equally important for having accurate metrics since a perfect object detector ideally has the following values for precision and recall respectively

$$P = 1, R = 1$$

Given that, the metrics derived from such precision and recall values can always be correct assuming that the calculation that is done for each of the metrics is also accurate.

Such values of precision and recall in the context of UI object detection represent the perfect solution. Nevertheless, those ideal values for precision and recall are not achievable in reality because it has been proven that an inversely proportional relation exists between them [12]. Despite that fact, an approximation to those ideal values for precision and recall is still desired since getting closer to these ideal values signifies more accurate metrics.

Measuring recall is achieved by applying the following formula

$$R = \frac{TP}{TP + FN}$$

where TP are the true positives, and FN correspond to the false negatives that result from the comparison between the object detector and the ground truth (test dataset) respectively.

The requirement is cataloged as:

- **Fully satisfied** if the approach reaches a sufficient recall rate in the UI object detection domain.

- **Partially satisfied** if the approach reaches at least the half of a sufficient recall rate in the UI object detection domain.
- **Not satisfied** otherwise.

2.2. Combining UI Object Detection Analyzers

Part written by Victor Flamenco

Reaching Objective B requires conceptualizing a software platform capable of integrating implementations of UI object detection providers, with the aim of improving UI object detection quality. Requirements are grouped into two categories: *Process-related* and *Results-related*.

2.2.1. Process-related Requirements

Process-related requirements describe the architecture and fundamental features of an integration platform that supports multiple UI object detectors. Moreover, this part describes the necessity of evaluating diverse combinations of available detectors. Requirements respecting this group are:

- RB1. Integration Platform
- RB2. Supporting a Wide Variety of Tools
- RB3. Results Aggregation
- RB4. Conflict Resolution
- RB5. Weight Distribution Analysis
- RB6. Uniform Communication Interfaces

RB1. Integration Platform

Diverse UI object detection techniques perform object detection through a set of sequential subtasks, and others, via a single subtask. Techniques have advantages and drawbacks for particular use cases, meaning that there is not one particular approach that guarantees optimal results always, and that, even though it is possible to try to optimize one technique to perform the full object detection, one can still miss a few benefits from other techniques.

For figuring out ways to support multiple UI object detectors, there is an apparent necessity for a common infrastructure enabling reuse of various distributed *software components* specialized in one or a series of object detection subtasks. A software component is “a unit of composition with contractually specified interfaces and explicit context dependencies only” [96].

A solution of such fashion is known as a *Platform*, a collection of sub-systems and interfaces that compose a common infrastructure [30]. Concretely, this work is interested in a specific type of platform: *Software Platforms*, formed by modules that perform various tasks and provide services to be used by other application parts [26]. Defined Application Programming Interfaces (APIs) enable access to platform services. An **Integration Platform** refers to a software platform that virtually joins a set of distributed software components into a shared infrastructure.

Configurability and *Distribution* are two essential characteristics of a software platform. The former term demands that components have inherently plug-and-play nature and become accessible to the platform via defined interfaces, without having to redeploy the solution [96]. Distribution refers to “spreading of computation and data across several computers connected by a network” [1].

Components integration implies devising communication instruments, including interfaces and protocols. This requirement focuses on joining distributed software components and leaves the communication challenge to the following derived requirements, as it needs further analysis. A suitable integration platform must connect distributed software components in a configurable fashion. Requirement assessment possibilities are:

- **Fully satisfied**: the platform allows configurable distributed software components.
- **Partially satisfied**: the platform allows external software components, but not in a configurable or distributed fashion.
- **Not satisfied**: the platform does not allow external software components at all.

RB2. Supporting a Wide Variety of Tools

Assuming the existence of a platform complying with RB1, it must support a potentially wide range of detectors. Object detection techniques follow different approaches and implementations, introducing variance in the quality of their results when compared to others.

Parser-based techniques have potentially complete outputs with a full description of semantic classes. A semantic representation of the UI is required, a particular challenge for desktop applications, and a more manageable task in the domain of web UIs. The heterogeneity of Web UIs introduced by sites lacking technological standards, different implementation possibilities, customizable tags, and other variables, compose a significant limitation for parser-based techniques.

Vision-based techniques provide a perception of UIs close to the one humans have, overcoming the limitation of web UIs heterogeneity. In contrast to parser-based techniques, the outputs of visual-based techniques involve an error probability. Under certain circumstances, these approaches are unable to detect particular objects correctly.

For both architectures, there are mostly two strategies: *end-to-end* and *divisible* implementations.

End-to-end implementations behave like black boxes, executing a series of subtasks virtually in a single step that receives a UI representation and returns, in essence, a set of identified objects. As an example, an end-to-end implementation can first identify rectangles that potentially contain relevant objects, and label each rectangle with an object type, to finally output a set of identified objects.

Divisible implementations, on the other hand, detect objects through a series of concrete sequential steps, where the first step receives a UI representation, and its output is pipelined to the next sequence step until the last step produces the final detection output. As an example, one step can identify potential object rectangles, and a second step can label each rectangle. The significant aspect here is the possibility to delegate each step to a different distributed software component.

Supporting multiple UI object detection techniques requires defining a workflow that describes the arrangement of subtasks execution generically enough to fit most techniques, and shall be designed aiming at identifying the four aspects in the following list. Of these, the first three are covered in this requirement, while the fourth one belongs to the derived requirement RB6.

1. Generic object detection tasks within the process,

2. Artifacts that these tasks generate,
3. Tasks arrangement within the workflow, and
4. Uniform communication interfaces.

This requirement defines two evaluation criteria to identify software platforms that support configurable software components as part of a composite workflow.

Criterion No. 1 focuses on the organization of software components at **definition time**. The platform must provide a mechanism to define the subtask(s) that each component performs within the workflow, and that allows describing the subtasks order and style of execution (parallel- or sequentially).

Criterion No. 2, on the other hand, focuses on the organization of software components at **execution time**. The platform must take some input and start a new UI object detection process. It shall run the process subtasks and organize the flow of work as described during definition time, and communicate different components via established interfaces. Finally, the platform must return some detection output.

Supporting a wide variety of tools means that the platform complies with both these criteria. The platform evaluation is categorized as follows:

- **Fully satisfied**: the platform allows the organization of software components at definition and execution times.
- **Partially satisfied**: the platform allows the organization of software components either at definition time or at execution time, but not both.
- **Not satisfied**: the platform does not allow the organization of software components at definition time nor execution time.

RB3. Results Aggregation

Taking advantage of the benefits of combining a variety of different UI object detection techniques is a compelling interest in this thesis. Combining implementations at execution time demands a results unification after tasks completion. The upcoming three aspects together make results aggregation possible.

First, each detector predicts a series of potentially inconsistent objects, and therefore, for unifying outputs, the platform must be able to disguise conflicting results and define strategies to solve them. The derived requirement RB4 contains a more in-depth discussion in this direction.

A second significant aspect for results unification is determining the importance of particular implementations towards computing the final system output. Figuring out this importance, or, *weight*, is achievable through several strategies, such as evaluating the quality of results thrown by the available object detectors, a challenge studied in the derived requirement RB5.

Even more, a third essential aspect for aggregating results is defining uniform interfaces, covering communication protocols, and input/output formats for UI object detection implementations and their steps, an aspect analyzed in the derived requirement RB6.

In an integration platform, this requirement evaluates the aggregation of diverse results into a single system output. The following categories compose this requirement's assessment.

- **Fully satisfied**: the platform aggregates results of different software components.
- **Not satisfied**: the platform does not aggregate results of different software components.

RB4. Conflict Resolution

By the nature of UI object detectors, especially vision-based ones since they output probabilistic results, they predict different results for the same input UI representation. When two implementations are given the same input and predict inconsistent outputs, unifying results requires first the resolution of conflicts. *Conflict resolution* refers to deciding which party predicted a more accurate detection. This is a derived requirement from RB3, and studies two **conflict dimensions**: bounding-box and classification.

Bounding-box dimension conflicts cover overlapping predicted bounding boxes labeled with the same object type, meaning that there is a considerable degree of similarity in their coordinates and sizes. **Classification dimension** conflicts, on the other hand, refers to cases of overlapping bounding boxes labeled with distinct object types.

A suitable platform includes a conflict-resolution component that describes rules to disguise conflictive results from the outputs of a series of software components that perform similar tasks, as well as defining mechanisms to make decisions in instances of each dimension in its domain of study. A mechanism of this type can be a rule-based or an algorithm-based decision-making subprocess situated towards the end of a macro process and functioning as a key step concerning results unification. The requirement's validity is evaluated based on these aspects:

- **Fully satisfied**: the platform dictates mechanisms to resolve conflicts in outputs of software components.
- **Not satisfied**: the platform does not provide mechanisms to resolve conflicts in outputs of software components.

RB5. Weight Distribution Analysis

Following the analysis of results aggregation, this requirement is an extension of RB3. It contributes to answering the question: how to aggregate results from the outputs of different detectors into one single output.

A significant aspect for results unification is determining the importance of particular implementations towards computing the final system output. Figuring out the importance -or **weight**- is achievable through several strategies, such as evaluating the *performance* of object detectors, in terms of the accuracy in detections.

Knowledge about the detectors' performance assists in determining a suitable weight to assign to each one when combining them and computing final system results. Moreover, this information is useful to discover their strengths and weaknesses at a diverse range of scenarios. These benefits aim at the improvement of detection quality.

In the context of UI object detection, an alternative for exploring the possible combinations of detectors is computing a permutation of the available software components that execute object detection subtasks, and filtering combinations are suitable for an object detection workflow.

A suitable weight-distribution analysis considers two criteria. First, the preparation of the analysis scenario: the inputs and arrangement of software components; and second, the evaluation of results based on metrics. Thus, the requirement is assessed as follows:

- **Fully satisfied**: the platform prepares the inputs and arrangement of software components, and evaluates results based on metrics.
- **Partially satisfied**: the platform either prepares inputs and arrangement of software components or evaluates results based on metrics, but not both.

- **Not satisfied**: the platform does not prepare the inputs and arrangement of software components, nor evaluates results based on metrics.

RB6. Uniform Communication Interfaces

An essential aspect for aggregating results is defining uniform interfaces, covering communication protocols, and input/output formats for UI object detection implementations and their steps. This requirement extends RB3.

Object detectors that address end-to-end and divisible implementations use potentially diverse input, intermediate, and output result representations. Therefore, a crucial aspect of accomplishing the unification of results is describing uniform communication interfaces that guarantee interoperability among software components, comprising two key considerations: uniform *representation formats* and *messaging protocols*.

Representation formats incorporate the syntax and semantics to describe input, intermediate and output artifacts in a process. The first criterion for suitable uniform communication interfaces is defining standard representation formats for ensuring detectors interoperability.

Messaging protocols, on the other hand, emphasize on mechanisms to invoke remote software components over the internet. Fundamentally, such protocols help to answer the question of how to uniformly communicate distributed software tools, to ultimately enable the platform to synchronize their execution and to support service completion/error notifications. The use of standard messaging protocols defines compliance with the second criterion.

A suitable platform complies with both criteria. It defines a set of uniform representation formats and messaging protocols for integrating diverse software components. The following categories describe the requirement's evaluation.

- **Fully satisfied**: the platform defines uniform representation formats and messaging protocols for integrating software components.
- **Partially satisfied**: the platform defines either uniform representation formats or messaging protocols for integrating software components, but not both.
- **Not satisfied**: the platform does not define uniform representation formats and messaging protocols for integrating software components.

fixi

2.2.2. Results-related Requirements

This category describes requirements aligned with enhancing the quality of UI object detection results obtained with the combination of detectors in contrast with the results shown by isolated detectors. These requirements aim at determining whether hypothesis B is valid, and since this is only testable after a prototype implementation, these are considered results-related requirements.

RB7. Enhancement of Detection Quality

This requirement aims at evaluating whether hypothesis B is true. Assuming an integration platform as demanded by process-related requirements is built, it must facilitate testing whether the UI object detection quality results improve with the combination of techniques when compared to the quality of detection results of isolated techniques.

An appropriate testing dataset, composed by a sufficient number of UI representations with a significant variety of objects shall be defined. The dataset is to be evaluated with each isolated detector, and subsequently, with combinations of detectors. The next step comprises the comparison of performances in these two scenarios in order to determine whether the combination achieved an improvement. Requirement satisfaction assessment follows these categories:

- **Fully satisfied**: the UI object detection accuracy obtained with the combination of techniques is better than the accuracy obtained with isolated techniques.
- **Not satisfied**: the UI object detection accuracy obtained with the combination of techniques is equal or lower than the accuracy obtained with isolated techniques.

2.3. Summary

This chapter defines two sets of requirements, one per thesis solution part. Each set categorizes requirements in either of two groups: process-related and results-related. Requirements describe an ideal solution to solve the problems described in section 1.3, simultaneously aligned with the objective statements pointed out in section 1.4.

Achieving Objective A demands a solution for visual UI analysis using SotA DNNs strategies for generic object detection. Process-related requirements discuss the object-detection and learning subsets. The former subset outlines functional and non-functional aspects related to the object detection process. In contrast, the latter subset describes non-functional aspects that outline what an ideal solution should consider from a learning and training perspective. The subset includes training-dataset and learning considerations.

The achievement of Objective B needs the conceptualization of a software platform capable of integrating implementations of UI object detection providers. Process-related requirements emphasize on the platform architecture, and derived aspects, including support for various detectors, uniform communication interfaces, and aggregating results. Moreover, this part described the necessity to evaluate different combinations of detectors for figuring out the most suitable ones for particular scenarios. On the other hand, results-related requirements focused on measuring the improvement of quality in the UI object detection results thrown by the unified output in comparison with the results of isolated detectors.

The following chapter evaluates the existence of approaches that fit the ideal solution, either partially or entirely. Groups of approaches sharing similar characteristics are defined. The analysis consists of revising these groups and a few concrete group representatives. The goal is to identify requirements that are already satisfied by existing solutions, and requirements that demand more attention as no current solution exists.

3. State of the Art Analysis

This chapter describes an ideal solution to address all problems specified in section 1.3, simultaneously aligned with the objective statements in section 1.4. The SotA analysis has the goal to discover approaches that fit the ideal solution either partially or completely.

As a first step, this analysis defines groups of UI object detection approaches sharing similar characteristics. Groups are revised, instead of individual approaches. Each group describes two essential aspects: (1) the group definition, a set of criteria to disguise whether an approach belongs to the group, and (2) a short description of a few concrete group representatives.

Subsequently, a table summarizes the groups' comparison, aiming at linking the requirements in chapter 2 with each SotA group. The evaluation assigns a mark to each group based on the scale described at the beginning of the previous chapter. The comparison table is followed by conclusions that guide to the identification of requirements that are entirely satisfied by existing solutions, and requirements that demand more attention as they are not currently well addressed.

The upcoming section 3.1 revises approaches aligned with the requirements in section 2.1, while section 3.2 inspects approaches relevant to the requirements in section 2.2.

3.1. UI Object Detection Using DNNs

Part written by Juan Sebastian Quintero

The following section examines the degree to which existing approaches and techniques already meet what is required for the object detection of the parts of a UI using DNNs (further details on the requirements defined can be found at section 2.1). For this purpose, the identification of different groups that contain approaches that share similar properties in the way they work is conducted. Each group is defined and further analyzed in detail. The final section of the chapter summarizes the result that comes from the performed analysis.

3.1.1. Groups

Given the nature and scope of the already identified requirements, the definition of two different groups is performed. On the one hand, there is a group that merely relates to object detectors which share the property of being based on DNN architectures. Furthermore, those object detectors are subdivided into two different groups: Two-Stage detectors and One-Stage detectors. On the other hand, there is a second group that relates to different approaches to the collection of training datasets. Within this group, the differentiation between existing training datasets as well as manual, and automated data generation approaches is stated.

The following list contains the definition of each group with their respective subdivisions in a structured manner:

- GA1. Object Detectors
  - GA1.1. Two-Stage Detectors
  - GA1.2. One-Stage Detectors
- GA2. Training Dataset Collection
  - GA2.1. Existing Training Datasets
  - GA2.2. Manual
  - GA2.3. Automated

Additionally, their relation to the requirements defined at section 2.1 is also specified in Table 3.1.

Group	**Requirements**
GA1. Object Detectors	RA1, RA2, RA3, RA4, RA8, RA9, RA10
GA2. Training Dataset Collection	RA5, RA6, RA7

Table 3.1. Mapping Groups and Requirements - UI Object Detection Using DNNs

GA1. Object Detectors

Currently, there exist different approaches that constitute the SotA for detecting generic objects such as pedestrians, cars, planes, or animals of any kind by visual means. An essential characteristic of such object detectors is the use of CNNs across their architectures.

In general, there exist two different types in which object detectors can be cataloged. The former one comprises *Two-Stage Detectors*, in which two separate phases are easily identified in their internal architectures; (1) the RoIs proposal phase that computes several regions across the image that contain potential objects, and (2) classification/localization phase where the labeling and computation of the location within those proposed regions is performed. The latter one refers to *One-Stage Detectors*, which perform both RoI proposals and classification/localization at once. The following subsections explain in detail each of those types, as well as describe a couple of current implementations, respectively.

GA1.1. Two-Stage Detectors

In two-stage detectors, also known as *Region Based* detectors, category-independent region proposals (or RoIs) are computed from an image as a first step. Then a feature extraction using CNNs from those RoIs is performed. Finally, category-specific classifiers are used to determine the category labels from each of the RoIs [3]. Such decomposition of the object detection pipeline is better illustrated at Figure 3.1.

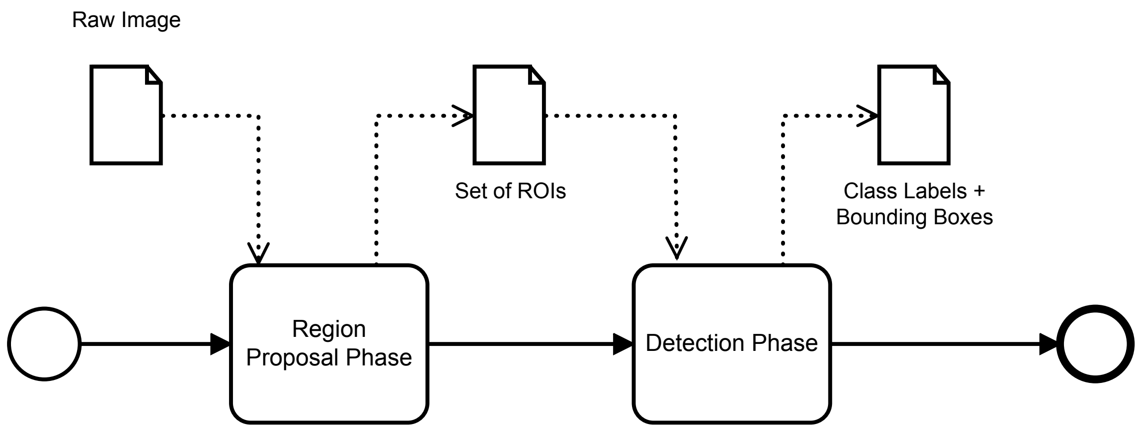


Figure 3.1. Two-Stage Detectors Overview

Several approaches follow this meta-architecture. They differ among others in the ways they compute RoIs, classify, or locate objects. Most of the literature highlights Regions with CNN features (RCNN) approaches at this matter where RCNN becomes the starting point, followed by *Fast-RCNN* and *Faster-RCNN* which improve the performance of their predecessors in several aspects such as execution time and accuracy of the predictions respectively. In the following paragraphs, both Faster-RCNN and Region-based Fully Convolutional Networks (R-FCN) are explained in detail.

The first approach is called **Faster-RCNN** [83] which is cataloged as one of the most performant representatives in terms of accuracy within two-stage detectors. This approach is the last one proposed by Girshick et al. within the RCNN family and has as its primary goal to get rid of the current bottleneck at the region proposals computation phase that exists in Fast-RCNN [31]. For such purpose, they introduce Region Proposal Networks (RPNs), which are in charge of the computation of region proposals having as input a previously computed convolutional feature map. Such setup allows sharing a set of convolutional layers between the detection network (basically the same one proposed in Fast-RCNN) and the RPN which is traduced in less computation time for RoIs. An overall description of its internal implementation is described at Listing 3.1.

```
predictions = {}
featureMaps = convolute(image)
rois = feedRegionProposalNetwork(featureMaps)

foreach (roi in rois) {
    projectedROI = computeROIPooling(featureMaps, roi)
    classScores, boundingBox = detectObjects(projectedROI)
    classProbabilities = softmax(classScores)
    predictions.add({classProbabilities, boundingBox})
}
```

Listing 3.1 Pseudocode - Faster-RCNN

Besides, to reduce computational time, this approach also claims to improve accuracy rates compared to previous RCNN-based implementations. There is evidence on the application of this approach at several domains such as the detection of pedestrians [100], Detection and diagnosis of colitis [59], elevator button recognition for autonomous elevator operation [101], among others.

The second approach described is named as **R-FCN** [18], it is proposed as a joint work between Microsoft Research and Tsinghua University, and emerges having as motivation the optimization of the tasks performed by Faster-RCNN after the computation of its RoI-pooling layer, where fully connected layers perform the prediction over each of the RoIs projected on its feature map, resulting in a computationally expensive operation. For such purpose, they go instead for a fully convolutional approach in order to be able to share even more computations across the prediction stage, and also to lower the number of parameters that need to be tuned resulting in cheaper and faster computations. To accomplish this, they introduce *Position-sensitive Score Maps* which are a set of feature maps that can detect different sub-regions of an object. Also, with the help of *Position-sensitive ROI-pooling*, a vote array is computed per each RoI over the score maps and further averaged in order to have a final class score. A summarized version of its core internal implementation is shown at Listing 3.2.

```
predictions = {}
featureMaps = convolute(image)
rois = feedRegionProposalNetwork(featureMaps)
scoreMaps = computePositionSensitiveScoreMaps(featureMaps)

foreach (roi in rois) {
    votes = computePositionSensitiveROIPooling(scoreMaps, roi)
    classScores, boundingBox = computeAverage(votes)
    classProbabilities = softmax(classScores)
    predictions.add({classProbabilities, boundingBox})
}
```

Listing 3.2 Pseudocode - R-FCN

The intuition behind this, is that it may be enough to detect an entire object by knowing just one part of the object e.g., a left eye of a face for detecting the whole face; but it may be even better if several parts of the object are known and such results can be combined together to have better detection rates e.g., knowing the left eye, the nose, the right eye, the mouth, among others. In their publication, they evidence competitive accuracy results compared to Faster-RCNN while at the same time, lowering computational times. There is literature that evidences the usage of this approach in scenarios such as self-driving cars [99].

GA1.2. One-Stage Detectors

In One-Stage detectors, also known as *Region-free Based* detectors, a *unified* detection strategy is performed. A Unified detection strategy comprises architectures that both classify (by predicting class probabilities) and locate (by predicting bounding boxes) objects from raw images with a single CNN in a monolithic setting that avoids the computation of RoI proposals as well as their respective classification/localization at separate stages [3]. The basics of this kind of detectors are illustrated at Figure 3.2.

Such architectures are ideal for devices with limited capacity since they perform object detection much faster than two-stage detectors. Nevertheless, their performance in terms of accuracy tends to decrease compared to two-stage detectors.

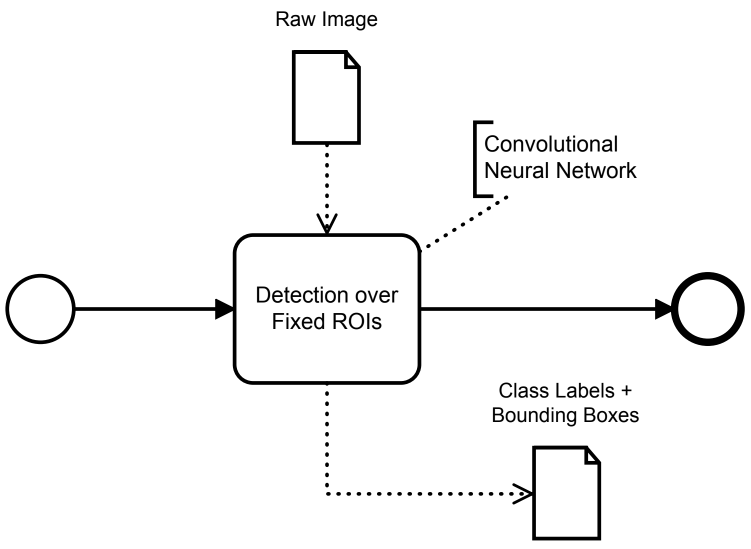

Raw Image
Convolutional Neural Network
Detection over Fixed ROIs
Class Labels + Bounding Boxes

Figure 3.2. One-Stage Detectors Overview

There are several implementations that follow this meta-architecture; some of the most cited implementations include all the versions of *YOLO* [79, 80, 81] approach, *SSD*, [61], *OverFeat* [89] and *RetinaNet* [57]. In the paragraphs below, the representatives YOLO and RetinaNet are better explained.

YOLO [79] is one of the former approaches that were proposed within the one-stage detectors family. One of the key aspects of this approach is that in it, the whole object detection problem is treated as a regression problem. The system divides the input image into a SxS grid. For each of the grid cells, it predicts B bounding boxes with a respective *confidence score* (confidence about containing an object). Each bounding box is represented as a set of 4 different values: (x, y, w, h), which constitute the coordinates indicating the central point of the bounding box (x, y) as well as its dimensions in terms of width and height (w, h). Besides, each grid cell predicts C conditional class probabilities. In summary, given an input image, YOLO outputs predictions encoded as an $SxSx(5B+C)$ tensor. It is important to highlight that all the operations are done with the help of a single convolutional network. This approach allows looking the input image all at once and enables having contextual information for the detection of objects.

Nevertheless, the first version of this approach evidences a set of limitations such as struggling with the detection of small and nearby objects, objects with unusual aspect ratios, or incorrect localizations. Its second and third versions eliminate many of the drawbacks evidenced in the first version by using different techniques. Further details of such techniques are explained in detail at [80] and [81].

RetinaNet is another approach proposed by T. Lin et al. [57] which arises under the goal of understanding if one-stage detectors can match or surpass two-stage detectors in terms of accuracy while running at similar or faster speeds. They find out that one-stage detectors tend to be less accurate than two-stage detectors because of extreme foreground-background class invariance. In order to compensate such class invariance, they reshape the cross-entropy loss function by adding a factor that they call the *Focal Loss*. The reshaped version of the function can be seen as a dynamically scaled cross-entropy loss, where the scaling factor decreases to zero as the confidence in the right class rises. The intuition behind this is that "easy examples" are down-weighted during training allowing to have a model that results from training focused on "hard examples". Being the former cross-entropy function $CE(pt) = -log(pt)$, the reshaped version looks like the following: $CE(p_t) = -(1 - p_t)^\gamma log(p_t)$. It introduces a new *focusing* meta-parameter $\gamma >= 0$ that controls the degree to which "easy examples" are down-weighted. Furthermore, the overall architecture of RetinaNet is comprised of a Feature Pyramid Netwrok (FPN) backbone (which generates a rich multi-scale convolutional feature pyramid) on top of a feedforward ResNet architecture (two-attached subnetworks for both classification and localization). The focal loss introduced in their work is used as the loss on the output of the classification subnetwork. There is evidence of usage of RetinaNet in different scenarios. Some few examples include vehicle surveillance [94] and the detection of lunar rockfalls [7].

GA2. Training Dataset Collection

Sufficient and meaningful training datasets can be acquired in different ways. The first one, instead of having to generate the training dataset, a good option is to look for existing ones that apply to the desired domain of knowledge. The second one, by following a manual procedure, in which with the help of qualified people, data is collected and labeled accordingly. The last one, by applying an automated approach in which data is synthetically generated from scratch, or existing training datasets.

The three groups are explained in detail in the following paragraphs.

GA2.1. Existing Training Datasets

Before starting to collect data and label it accordingly from scratch, a reasonable consideration is to look for an existing training dataset and reuse it if possible. Nevertheless, the difficulty in finding existing training datasets can vary depending on the domain for which the training dataset is required.

This task is merely dependent on what is generally referred by the literature as *Data Discovery* which contemplates a set of approaches for both sharing datasets and also for searching existing ones. For sharing datasets, there are already suitable platforms such as DataHub [6] which provides dataset management features on top of a Control Version System (CVS), or Google Fusion Tables [33] which allows sharing datasets right into the web in a structured format. For searching datasets, there are also some prominent implementations such as Google Data Search (GOODS) [36] which indexes metadata of several datasets within Google's storage systems, or Google Dataset Search [11], a service launched for searching repositories of datasets across the web.

Different existing training datasets have been identified in the context of generic object detection by using the approaches as mentioned earlier. Pascal VOC [27], COCO [58], and Open Images [54] are some of the most prominent open datasets at this regard. However, for the specific domain of detecting UI objects, an identification of a training dataset that fully or partially satisfies this constraint has not been possible. Despite this fact, it is worth analyzing existing open training datasets for generic object detection in order to check for particular properties that can be beneficial in the knowledge domain of this thesis.

The Pascal VOC Challenge [27] provides the vision and machine learning communities with (1) a public dataset of annotated images from 20 different classes of quotidian objects (airplane, bird, bicycle, boat, bottle, bus, car, cat, chair, cow, dining table, dog, horse, motorbike, person, potted plant, sheep, sofa, train, tv/monitor) together with standardized evaluation procedures through existing software and (2) An annual competition and workshop were different computer vision algorithms compete each other to become the SotA in this matter. The dataset contains annotations for different kinds of challenges such as image classification, object detection, and object segmentation. The main objectives of the VOC challenge are to provide high quality annotated and challenging images together with an evaluation methodology through "plug and play" so that the performance of different algorithms can be compared in a standard way. Since the goal of the VOC challenge is to investigate the performance of different approaches on a broad spectrum of natural images and also to ensure accurate training and evaluation, it requires that (1) the dataset contains significant variability in terms of object occlusion, position, illumination, pose, orientation and size, (2) that there are no systematic bias across the dataset e.g., favoring images with proper illumination or centered objects within them and (3) that the image annotations are consistent, accurate and exhaustive for the different classes. Having such requirements in mind, they collect the dataset images from Flickr[1], a website containing images of any kind uploaded and tagged by its community. This decision allows them to count on a very "unbiased" dataset i.e., photos not being taken with a

[1]https://www.flickr.com/

particular purpose in mind. After the image collection and the further elimination of "near duplicates", images are presented to annotators, that are in charge of pruning the images and their respective labeling having standardized guidelines to do so (what to label, how to label pose and bounding box, how to treat occlusion, acceptable image quality, among others). The annotation process is periodically observed in order to ensure that guidelines are followed correctly by the annotators; in addition they count on mechanisms that allow computing dataset statistics such as histograms of number of images and objects in the entire dataset for each of the classes that allow them to focus annotation efforts on "minority classes" in order to provide a reasonable minimum number of images/objects per class. After the annotation process, the data is divided into two subsets: training/validation data (trainval) and test data (test) and made available to the public.

GA2.2. Manual

Several training dataset collection approaches are manual-based. It means that either the data acquisition, labeling, or a combination of such tasks is performed with the help of human beings.

For such manual-based approaches, it is crucial to count on the ideal people, because the characteristics such as experience, or expertise in a specific knowledge field can vary according to the domain. There are cases where people with knowledge in a domain-specific area is required, and some others that only need a limited type of experience. Also, the number of people that are needed should be considered as well, because it impacts directly the time it takes to get the job done. For such requirements, there is a need to count on the right techniques that facilitate both spreading the description of the tasks and finding qualified people that carry out them. Thanks to the power that the web brings for connecting people, such requirements can be easily satisfied. The most prominent manual approaches follow the concept of *Crowdsourcing*. Such concept was first introduced by J. Howe [44] and is better explained in the following paragraphs.

Crowdsourcing is a model that "represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call. It can take the form of peer-production (when the job is performed collaboratively) but is also often undertaken by sole individuals. The crucial prerequisite is the use of the open call format and the large network of potential laborers" [10]. Such a model allows solving problems in a distributed and "outsourced" fashion with the help of the "crowd".

There is not a fixed way to apply this model so approaches can vary enormously, however, approaches under this model share something in common: A description of a task that needs to be performed, and the reward that is given in return (money, additional services, benefits of any kind, among others) for performing the task. In the context of training dataset collection, the overall task description is clear: performing acquisition/labeling of data.

Some application scenarios in the industry show evidence on crowdsourcing as the business model core of several companies, such as in the case of IStockphoto[2], a micro-stock photography provider that sales millions of media archives that come from different contributors across the network who get commissions whenever their artworks are sold.

Moreover, some platforms offer support for crowdsourcing in a generic way such as Amazon Mechanical Turk[3], where tasks of any kind created by an individual or business are assigned to human workers which are compensated in return when such tasks are finished. With this kind of platforms, the whole crowdsourcing infrastructure and flow are provided, enhancing businesses and individuals to focus on solving their main problems.

GA2.3. Automated

Automated approaches for collecting training datasets are those in which data is synthetically generated and further labeled by machines. It means that no human intervention is required for such approaches.

The applicability of automated approaches for the collection of a training dataset depends merely on two different aspects: (1) the domain of knowledge, and (2) the lack/existence of a current dataset. When the new data generated depends on existing data, it receives the name of *Data Augmentation*, and when the data is generated from scratch, it is called as *Data Generation*.

Commonly, data augmentation approaches can be applied generically regardless of the domain of knowledge because they depend merely on existing labeled data; Generative Adversarial Networks (GANs) [17] is the most prominent example that can be mentioned in this regard, where two neural networks are confronted against each other in order to get better at their particular goals. The *Generative Network* learns to map from a latent space to a data distribution in order to generate synthetic samples, and the *Discriminative Network* tells whether the input data samples are

[2]https://www.istockphoto.com
[3]https://www.mturk.com

synthetic (coming from the generative network) or authentic (coming from an existing training dataset). On the one hand, when the discriminative network recognizes a synthetic image, it is rewarded and the generative network penalized. On the other hand, when the discriminative network is not able to detect an image as synthetic, it is penalized and the generative network rewarded accordingly. Such setup allows both networks to confront each other, and learn simultaneously from their respective feedbacks. When the generative network is good enough to generate new data samples, it can be used as such.

Besides GANs, other data augmentation techniques can also be applied, but they tend to be dependent on the nature of the data that they handle. For instance, data represented as images can be augmented by using several techniques over the existing images such as scaling, zooming, rotation, reflection, color shifting, among others [67].

For data generation, there is not a specific set of approaches since they may vary widely concerning the target problem and domain of knowledge. Such techniques are considered to be *ad-hoc* because they are problem-specific and are not transferrable/reusable in other domains. An example of such approaches in the context of text generation is *Paraphrasing* [65] which consists of the generation of compatible expressions that have similar semantic meanings given an input expression.

3.1.2. Comparison

This section makes a comparison between all the groups defined in terms of fulfillment across the requirements that apply to them respectively (see Table 3.1). A summarized version of this comparison can be found in Table 3.2 and Table 3.3.

RA1. Detection of the Parts of a UI

Since detecting the parts of a UI can be seen as the application of object detection in a specific domain, it is feasible for both one-stage detectors and two-stage detectors because such set of detectors are agnostic from the application domain. There are differences in terms of performance between one-stage detectors and two-stage detectors. While the former group of approaches focuses on delivering real-time performance thanks to its “unified” and more straightforward meta-architecture, the latter tends to be more accurate. Across time, one-stage detectors that have comparable results in terms of accuracy with two-stage detectors while being faster, have emerged. Nevertheless, it cannot be guaranteed that both sets of approaches can give good results in this thesis domain, and it still needs to be proven.

Given those facts, the requirement is evaluated as *fully satisfied* for both groups, however, it is relevant to point out that the evaluation goes slightly higher for two-stage detection approaches because accuracy has more priority than time in this case.

RA2. Automation of Object Detection

As stated in the requirement “Process Automation”, the automation scope is targeted to only the prediction phase, i.e., given the visual representation of a UI, predict the UI objects with their corresponding bounding boxes. Both one-stage detectors and two-stage detectors can accomplish process automation according to the defined criteria. Regardless of their internal implementations, approaches in both groups can “predict” objects without the help of humans. Nevertheless, they can still require intermediate manual actions in order to call the prediction procedure, such as preparing the inputs or initiating the prediction procedure.

That is why the requirement of “process automation” is equally evaluated for both one-stage detectors and two-stage detectors with a *partially satisfied* rating.

RA3. Visual Input

Evaluating the requirement of visual input, all of the groups excel regardless of their respective internal meta-architectures. Two-stage detectors take an image as an input, to further compute RoIs, and calculate bounding boxes through regression and labels through classification afterward. One-stage detectors also take an image as an input and compute both labels and bounding boxes in a “unified” way through regression.

For such facts, the requirement is *fully satisfied* by one-stage detectors and two-stage detectors.

RA4. Deep Learning

Assessing the “Deep Learning” requirement, both one-stage detectors and two-stage detectors are based on deep architectures.

One-stage detectors tend to perform the calculation of feature-maps and the prediction of class labels as well as their respective bounding boxes with the help of deep

CNNs. Equally, Two-stage detectors also use deep CNNs, but what varies is that they can use them either in their RoI proposal stage, their prediction stage, or both.

Besides, one-stage and two-stage object detectors are cataloged within supervised-learning because all of their related approaches are merely dependent on labeled data in order to learn a particular object detection model.

With those facts in mind, this requirement is *fully satisfied* by both of the groups.

RA5. Sufficient Training Dataset Size

The requirement of collecting a training dataset comprised of images and labeled objects of the different classes described at RA1 with sufficient training examples in terms of size is assessed as follows:

By reusing an existing training dataset this requirement is usually satisfied, since usually datasets are published for further usage once they have sufficient sizes, nevertheless it was not possible to find a suitable existing training dataset that meets the defined criteria, for this reason, this requirement is *not satisfied* by the group of existing training datasets.

In the case of manual approaches, it is feasible to acquire a training dataset sufficient in size; nevertheless, it requires a high number of people, budget, and logistics in order to accomplish this. Despite those facts, the requirement is considered as *fully satisfied* by manual approaches.

The group of automated approaches tends to be more suitable for generating the training datasets sufficient in size. Given the nature of this thesis domain, the application of this group of approaches is possible. Generating synthetical data is, in this case, feasible using an ad-hoc approach because for the particular case of detection of UI objects, the DOM can be manipulated for both the generation and localization of UI objects across its API. For such reason, the requirement is *fully satisfied* by automated approaches.

RA6. Sufficient Training Dataset Frequency of Distribution

The requirement of collecting a training dataset comprised of images and labeled objects of the different classes described at RA1 with sufficient training examples in terms of frequency distributions across classes is assessed as follows:

For already existing and suitable training datasets this requirement usually is already addressed, nevertheless, it should be good to review the documentation of the dataset in order to validate the required criteria, and if not, try to add more training examples for satisfying the one that is missing. Unfortunately, it was not possible to find an existing training dataset meeting the specified criteria. For this reason, this requirement is *not satisfied* by the group of existing training datasets.

In the case of manual approaches, it is feasible to achieve this requirement by having proper control mechanisms in which the frequency distributions across data collected and labeled by people can be validated regularly to generate appropriate feedback. Therefore, such approaches provide a satisfactory solution, and the requirement is *fully satisfied*.

In the case of automated approaches, similarly to manual approaches, meeting the requirement can be accomplished by having proper control mechanisms that continuously validate frequency distributions across data collected/generated and labeled automatically. For such reason, the requirement is *fully satisfied* by automated approaches.

RA7. Training Dataset Collection Efficiency

The efficiency of the training dataset collection approaches is assessed as follows:

Reusing an existing and suitable training dataset is ideal when assessing it in terms of efficiency of the dataset collection since the only efforts that are needed are the ones related to finding and locating such datasets, and its further validation. Those are tasks that one person can accomplish entirely without significant problems. Said that this requirement is *fully satisfied* when it is possible to reuse an existing dataset.

Manual approaches can require more than one person in order to collect sufficient training dataset in terms of size and frequency of distribution. Since all the effort is performed manually, it results in high costs, more logistics, and time. In this case, there is a tradeoff between time and the number of people that work in parallel. The more people work on the task in parallel, the less time it takes for the task to be completed. By having such considerations, the requirement is *not satisfied* by manual approaches.

Automated approaches are also suitable since the manual effort relies on the development of software that is capable of collecting/labeling/synthetically generating data. One single person can make such an effort without significant problems if the person has the proper skills to develop the software. Hence, the requirement is *fully satisfied*.

RA8. Avoid Re-training from Scratch

This requirement is directly related to the learning approach that is followed in order to build the object detection model. Since both one-stage detectors and two-stage detectors are based on deep neural networks, they can be trained similarly with mini-batch learning algorithms such as backpropagation and mini-batch Stochastic Gradient Descent (SGD). Most of the frameworks that support the development of deep learning architectures also offer support on those learning algorithms and additionally offer the possibility to "freeze" the model weights for further training. It allows the development of systems that can learn progressively from different dataset deltas, and adapt to new changes over time. However, those object detection meta-architectures do not consider all the external actions that require learning progressively.

For this reason, this requirement is *partially satisfied* by one-stage detectors and two-stage detectors.

RA9. High Precision

In terms of precision, according to the literature, there is evidence that two-stage detectors have in general better performance than one-stage detectors but are at the same time slower to train/use [46].

Besides, there are internal and external factors that are intrinsic to the group of approaches that also contribute to the precision rates. Some external factors include the size of the input images (resolution) and the quantity and quality of the training dataset. Some internal factors include the feature extractors (or backbone CNNs), the learning algorithms and different hyper-parameters that the object detector architecture demands. Most of the mentioned internal and external factors introduce a tradeoff between the computations/time and the precision rates of the object detection model.

In summary, precision rates achieved by such groups can vary enormously depending on its internal configurations as well as the amount and quality of the data used for training. For achieving sufficient precision rates, there are several things to consider; nevertheless, such things are independent of the object detection groups defined. In general, one-stage detectors are slightly below two-stage detectors in precision rates, but both sets of approaches are feasible to achieve good results when configured and trained correctly.

Given those aspects, and also having into account that deep learning-based approaches for object detection outperform in terms of precision rates any other approach in the generic object detection domain, both sets of approaches look ideal for detecting objects, However, such aspects have been confirmed in the application of object detection in a generic domain and this behavior can vary given the specific conditions introduced by the domain of UI objects detection. At this point, there is no evidence showing the precision performance of such object detection strategies in the domain of interest. That is why this requirement is *not satisfied* at the moment by both two-stage detectors and one-stage detectors.

RA10. High Recall

Similar facts described for the precision requirement, occur for the recall. Most of such facts introduce a tradeoff between the computational power and time needed for training/using the object detection model and its respective recall rates. Also, some particular aspects influence recall rates, such as the number of RoIs configured for two-stage detectors. Having a higher number of RoIs influences positively the recall of the model; however, it increases the required computations [46].

Following the same considerations taken for the precision requirement, this requirement is *not satisfied* by both of the object detection groups.

	GA1.1. Two-Stage	**GA1.2. One-Stage**
RA1. UI Parts	++	++
RA2. Process Automation	+	+
RA3. Visual Input	++	++
RA4. Deep Learning	++	++
RA8. Avoid Re-training	+	+
RA9. High Precision	-	-
RA10. High Recall	-	-

++ fully satisfied, + partially satisfied, - not satisfied

Table 3.2. Groups Evaluation - Object Detection - UI Object Detection Using DNNs

	GA2.1. Existing	**GA2.2. Manual**	**GA2.3. Auto**
RA5. DS Size	-	++	++
RA6. DS Freq Dist	-	++	++
RA7. DS Coll Eff	++	-	++

++ fully satisfied, + partially satisfied, - not satisfied

Table 3.3. Groups Evaluation - Dataset Collection - UI Object Detection Using DNNs

3.1.3. Conclusion

In the previous sections, the definition of different groups of approaches according to their relationship to the set of requirements that arise from an ideal solution that allows the detection of UI parts using deep learning architectures has been addressed.

A first collection corresponding to Object Detection approaches includes two groups which are one-stage object detectors and two-stage object detectors respectively. Comparing such groups across the requirements, one can conclude that existing approaches in both of the groups satisfy in general all the criteria that need to be met. It is observed that most of the evidence of usage of such approaches rely on *mini-batch* learning. It is crucial, however not sufficient for learning progressively from different dataset deltas since it requires several intermediate steps. In addition, assuring sufficient precision and recall rates for detecting UI parts is still not possible because it is evidenced in the literature that the performance of the object detection techniques varies depending on both the dataset and the configuration parameters used for training. Hence, for assuring sufficient precision and recall rates for detecting UI parts, the different approaches need to be tested with several configurations against a training dataset that meets the requirements described in this thesis.

A second group targeted here includes different training dataset collection techniques. Within this group, re-usage of existing training datasets, manual dataset collection, and automated dataset collection approaches are defined and explained accordingly. From the comparison among these different approaches, it is concluded that the set of approaches that are *automated* are more suitable for this research thanks to the different ways in which the training dataset can be synthetically generated in the domain of this thesis (detecting parts of UIs), and also because they end up being cheaper and faster to implement compared to manual approaches. Reusing an existing training dataset with labeled UIs is ideal but unfortunately not possible at the moment because there are still not public training datasets easily locatable for this purpose. Despite those facts, it is important to make it clear that any of the approaches explained at this regard are valid, and a combination of them is always possible because such set of approaches are not mutually exclusive.

To sum up, DNN-based object detection techniques are robust nowadays for the detection of generic objects, nevertheless this fact can vary with respect to the detection UI objects; reusing a one-stage detector or a two-stage detector is feasible for the the thesis domain (detection of UI parts) but this needs further validation; it is also important to note that learning progressively with such detectors is supported but requires additional “automatable” steps. Furthermore, until the moment there is not an existing dataset that can be reused to train an object detector, this is why more effort needs to be put in the collection of a sufficient training dataset. With this in mind, automated dataset collection approaches better suit the needs at this matter; however, a combination of approaches is still possible at this regard.

3.2. Combining UI Object Detection Analyzers

Part written by Victor Flamenco

This section inspects existing implementations of platforms to integrate distributed software components in a configurable fashion, considering that the combination includes the arrangement of components during process modeling and components orchestration during process execution. This work furthermore researches current applications for merging outputs of different software components and resolving inconsistencies.

In order to evaluate solutions, subsection 3.2.1 first defines existing groups of approaches. Then, subsection 3.2.2 elaborates on an evaluation per group against the requirements related to Combining UI Object Detection Analyzers. Lastly, subsection 3.2.3 concludes the analysis.

3.2.1. Groups

This subsection identifies three groups of approaches that share similar characteristics. The main criterion to distinguish these groups is the *architectural design* followed by implementations to combine distributed software components.

Architectural design refers to the result of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system [1]. *Architecture* is a high-level pattern to arrange components of technical modules [76], a “set of rules to define a system’s structure and the interrelationships between its parts” [48]. This thesis emphasizes on *Software Components*.

The current analysis presents three architectural designs: *service-*, *agent-* and *workflow-oriented*. Per group, it provides two aspects: (1) the group definition, a set of criteria to disguise whether an approach belongs to it, and (2) a description of a few concrete group representatives.

- GB1. Service-Oriented Architectures
- GB2. Agent-Oriented Architectures
- GB3. Workflow-Oriented Architectures

GB1. Service-Oriented Architectures

In Service-oriented Architectures (SOAs), several services collaborate to provide an end set of functions [71]. A **Service** is "a set of reusable components that can be used to build new applications or integrate existing software asset" [42]. Services belong to completely separate operating system processes, and therefore, network calls are necessary for inter-service communication. One of the main motivations behind SOA is building service ecosystems, complex software systems that compose, combine, and encapsulate multiple services into a single platform [4]. SOA obtains *loose coupling* among the cooperating services [78].

Styles with service-orientation model interactions between three types of parties: **Providers**, **Consumers**, and **Registries**, and one type of artifacts, **Descriptions**. Some authors also refer interchangeably to a registry as a *broker*. For consistency, nevertheless, only the term registry is used from this point. Figure 3.3) depicts the SOA architecture.

A service is a single unit of work performed by a service provider, which presents results to a service consumer. Providers grant some independent functionality and deliver *metadata* to the registry. Metadata describe a service with information about its functionality, category, and prerequisites to access to and interact with the service. On the other hand, service consumers request information to the registry about available providers and their descriptions and can access to the available purposes given by service provider, via network calls. Services are autonomous in the sense that one can develop, modify and deploy them independenty and transparently to consumers.

A significant objective of SOAs is to maximize the reuse of functionality of existing systems in a wide variety of scenarios, minimizing the dependencies of a component on functionality provided by another. Dependency minimization enables loose couping [37]. In SOA, services are accessed in a standardized fashion, allowing high services *interoperability* and reusability enhancement.

The service-oriented paradigm has one modern subfield of study: the **Microservices** architecture. It emerged since the conventional SOA does not provide practical ways to guarantee loosely-coupled services as it does not clearly define how big or how

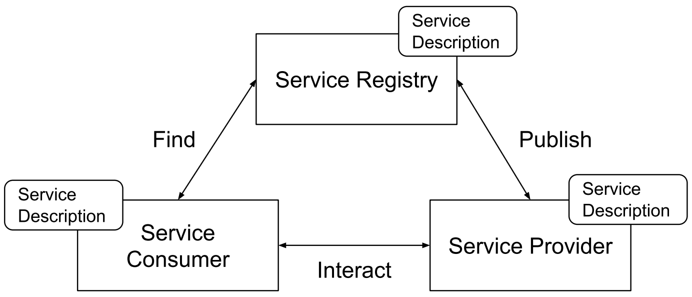


Figure 3.3. Service-Oriented Architecture [78]

small services should be. Microservices are *small* and *autonomous* services that work together, emphasizing on a more practical and real-world view of services [71].

A microservice has an explicitly-bounded functionality and is small enough to focus on executing one task well. The smaller the microservice, the more benefits of loose coupling are obtained. Nevertheless, this implies a higher number of microservices to process complex divisible tasks, and thus, more parts need to be maintained. Finding the right balance between the size and number of services is crucial to a successful microservices architecture. Microservices facilitate services composability and replaceability.

The *API-Gateway pattern*, a paradigm for composing microservices, is motivated by the necessity of server-side aggregation procedures in end-user systems built as a composition of microservices. It minimizes the number of interactions between consumers and providers by acting as the application's entry point, receiving calls from consumers, and redirecting requests to one or multiple appropriate services that compute some task and return sets of results to the gateway. These results are aggregated and sent to the original requestor. Use cases for this paradigm are, for example, authentication and load balancing.

With the above concepts in mind, an approach that belongs to this group defines the three key parties of a service-oriented model and specifies standard service descriptions. This subsection explores two instances of service-oriented approaches, namely *Sens-ation* and *InterSCity*.

Sens-ation, designed by the Computer-Supported Cooperative Work Group, is a platform for the development of sensor-based infrastructures, systems that inspect their surroundings utilizing sensors and based on these evaluations, adapt their actions [35]. The solution offers a generic model to compose distributed software -and hardware- components, and to connect them via uniform communication mechanisms for querying/sending information among sensors and between the platform and sensors. Service providers are Sens-ation servers having one or more sensors attached.

Consumers can request the Sens-ation registry information about available providers, and additionally, communicate directly to each of them.

Several sensors collect and send data from their environment to a server via adapters. These adapters facilitate communication between sensors and the first platform layer: the registry layer, which acts as a sensors registry. Once data reaches this point, it is persisted, aggregated, and processed by inference engines. External consumers can access the processed information via the platform's registry. Sens-ation's architecture provides two types of registries: one for discovering sensors within a service provider, and one for external consumers to discover the platform's services.

SOA in sensor infrastructures is implementable for: *surveillance* and *ubiquitous computing*. The former consists of collecting and analyzing geospatial data of the Earth using networks of remote sensors that act upon demand, and using the gathered information to make decisions. The Geospatial Information and Communication Technology Lab created a web-based platform named Sensor Web. An example in the latter application scenario is SensorJini, an infrastructure for software sensors and their communication, that presents a LookupService utilized to bind providers and consumers [35].

Sens-ation has several use cases. For example, in industry scenarios where two or more offices from geographically separated companies cooperate. Cooperation requires communication among employees in different locations. A crucial aspect of accomplishing this is that each employee that needs to communicate shall know about the presence and availability of the target employee. A sensor per employee can capture, among other data, these two state properties of an employee and make them publically available to others.

InterSCity is an open-source microservices platform for smart cities, designed and built in the context of the INCT of the Future Internet for Smart Cities project [24]. The field of smart cities studies technologies to handle commonly found problems in big cities by using city resources efficiently and presenting their population with good quality of services. A city resource is a physical entity present within the city, composed by a set of attributes, such as a description, a location, and defined functionalities, e.g., cars, buses, and traffic lights.

The platform presents a microservices architecture that provides high-level RESTful services for handling diverse Internet of Things (IoT) devices, data management and computation, and service discovery. Services composition is achieved through asynchronous communication using the lightweight message bus RabbitMQ3 using the Advanced Message Queuing Protocol (AMQP). Six microservices compose this architecture.

The resource-adaptor microservice enables registration and update of city resources. It functions as a proxy for IoT gateways that publish data collected from city resources and bind to events for the command of actions to the resources. The data-collector microservice persists data and provides a sophisticated filter-supporting interface to query context data captured by city resources. The resource-catalog microservice assigns Universally Unique Identifiers (UUIDs) per resource and notifies other microservices upon resources creation. The resource-discovery microservice provides consumers with flexible querying of city resources.

Evidence of a use case for building smart cities on top of the InterSCity platform is *Smart Parking*, an app that helps drivers find available parking places in a city through a map with live-reporting information about spots. It counts with a set of sensors that notify the presence of a car in a spot [24]. In this application, parking sensors push data continuously to the IoT gateway, which records availability per parking place and communicates the platform each time a spot changes its availability status. It keeps track of the resources UUIDs and notifies upon state changes. Drivers set a target location or use their Global Positioning System (GPS) and personalized parameters for refining parking spaces. The app displays a map with the query results and additional information about a specific spot and its availability history.

GB2. Agent-Oriented Architectures

Agent-based architectures (Figure 3.4) revolve around the concept of agents. An **Agent** is a computer system located in an environment, with encapsulated behavior, that acts flexibly and autonomously, by controlling its internal state and actions [50]. Agents have clear and specific functionality boundaries and communication interfaces; they can perceive inputs from their environment and execute tasks that change the environment. Agents are traditionally categorized as *cognitive* -understands its environment and can reason- and *reactive* -has simple knowledge and cannot reason, it responds to events- [66].

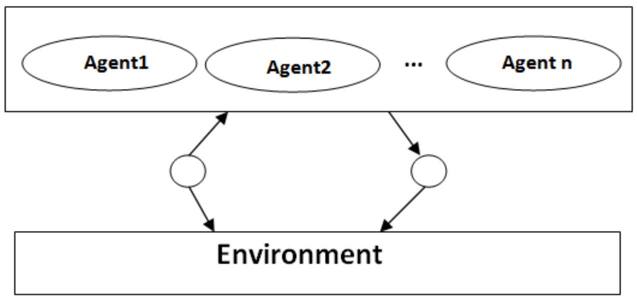


Figure 3.4. Agent-Oriented Architecture [70]

Research of systems composed by agents is traditionally part of the field of Distributed Artificial Intelligence (DAI), and from there, two subfields emerge: Distributed Problem Solving (DPS) and Multi-agent Systems (MASs) (see Figure 3.5). The latter ones are presently no longer limited to the scope of DAI, referring to distributed systems composed of multiple semi-autonomous components. A MAS is "a loosely-coupled network of problem solvers that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver" [49]. These platforms compose agents for cooperative DPS when some agents are unable to do one particular task, or when other agents do better. The traditional multi-agent models are described in terms of a cognitive two-level architecture, in which, a macro-agent organizes the system behavior by coordinating agents [66].

MASs enable robustness and efficiency by combining agents to solve problems. They enable interoperability for pre-existing agents [49], facilitate the reuse and cohabitation of distinct algorithms [64], and are fit for distributed computing, as complex tasks can be organized as sublayers or components [95].

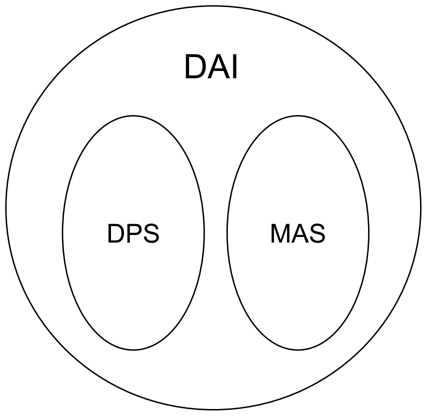


Figure 3.5. Multi-Agent Systems - Traditional Scope

Building a MAS requires the system's designer to guarantee that the appropriate combination of agents is present within the system [14]. Agent Communication Languages (ACLs) enable agents cooperation by sharing an agreed understanding of the meaning of message content and its communication contexts [73]. Cooperation and coordination of agents are possible due to their social ability through messaging. Agent systems shall ideally adhere to internationally recognized standards for inter-agent communication. Similar to known communication mechanisms, inter-agent communication is handled by several layers: *message transport*, *message format*, and *message content*.

An approach that belongs to this group provides an architecture for composing distributed agents situated within a shared environment, and that can perceive changes and alter the environment's state. Agents in this infrastructure shall interact together in order to solve a common problem.

The agent-based architectural style is exemplified with the *FIPA* framework and *Cougaar*.

The **Foundation for Intelligent Physical Agents (FIPA) Framework**[4] is the de facto standard for developing agents in a few fields. It covers message transport by describing the lifecycle of an agent that resides on a platform in six phases: creation, registration, location, communication, migration, and retirement [14].

The platform instantiates agents and handles messaging with transport protocols such as Hypertext Transfer Protocol (HTTP), and tracks agents at any given moment. A Directory Facilitator (DF) agent acts as a directory of services and agents in the platform. Agents joining the platform are registered at the DF.

The FIPA-ACL formalizes the *message format*. This standard covers the structure and message flow within the platform. A message has one mandatory field: the message's intention. Common intentions are, e.g., requests for actions, queries for information, or notification passing. A message typically includes three optional fields: the addresses of communicating agents, the sequences of messages, and message contents.

Content languages and ontologies describe the syntax and semantics of the *message content* structure, respectively. The former aspect may be explained using the FIPA-Semantic Language (SL), which provides messages grammar. The latter aspect requires application-specific design, as ontologies cannot be generic. Therefore, when developing a multi-agent platform, the designer must create a fitting ontology for agents operation.

Cougaar is a configurable platform for autonomous cooperative distributed systems, designed with aerospace applications in mind, developed in the context of the UltraLog program [41]. Mission control is costly and demands the composition and coordination of several subsystems, a design that can take advantage of an agent-based model. The platform supports a wide variety of agent applications. It was used to build a large-scale distributed prototype planning application, showing the benefits of autonomous agent architectures for reliable and cost-effective aerospace systems, overcoming challenges such as unreliable networks and dynamic operation situations.

The platform is formed by multiple applications that share a common infrastructure during their lifecycle, i.e., instantiation, communication, and survival. It enables

[4]http://www.fipa.org/

agents to communicate through message passing, and within each agent, dynamically loading *plugins* that interact via a publish-and-subscribe methodology. The key architecture parts are: *plugins*, the *blackboard* and the *message transport system.* Moreover, it supports *naming services*, *communities*, and *service discovery.*

Plugins are software pieces embedded to an agent for doing a specific task. Plugins are first loaded, and they interact with each other through the agent's environment: the blackboard. The platform offers a global agents environment formed by the union of all of their blackboards. The Message Transport Service (MTS) facilitates communication among agents by providing a flexible mechanism to enable asynchronous transport protocols to be attached as plugins.

Cougaar naming services include *white pages* -a flexible distributed table for mapping agent names to network addresses- and *yellow pages* -an attribute-based queryable directory service for registering agents and their capabilities-. Communities are groupings of agents based on their purposes, roles similarities, and physical attributes. Messages are attribute-based, and can, therefore, be broadcasted to target an entire community or agents with a specific role. Lastly, the Service Discovery (SD) enables agents acting as either consumers or providers, to know about the existence of each other, and how to establish communication.

Evidences of uses cases for autonomous agents include aerospace applications, such as *ground-based control* and *autonomous decision making* [41]. The National Aeronautics and Space Administration (NASA) and the Goddard Space Flight Center enhance ground-based control by developing a component-based model to build agent communities, where agents communicate using the FIPA-ACL. An evidence work in the latter field is the Autonomous Sciencecraft Experiment from the Jet Propulsion Laboratory (JPL), which uses a single autonomous agent to integrate the operation of sensor plugins based on dynamic planning.

GB3. Workflow-Oriented Architectures

Workflow-oriented architectures support the implementation of diverse distributed execution environments where interrelated tasks are executed and supervised [2]. Workflow Management Systems (WFMSs) are designed to coordinate *business processes* -"procedures where documents, information or tasks are passed between participants according to defined sets of rules to achieve, or contribute to, an overall business goal"-, which are concretized as *workflows*, computer-readable representations that specify all aspects needed for process execution, among which are: the order and conditions for tasks execution, task input parameters, flow of data, and entities responsible for tasks.

WFMSs offer a collection of tools for modeling and describing workflows, an environment for process execution, and interfaces to connect participant entities (generally people or software). The Workflow Management Coalition published a *Workflow Reference Model* [43] to standardize these platforms.

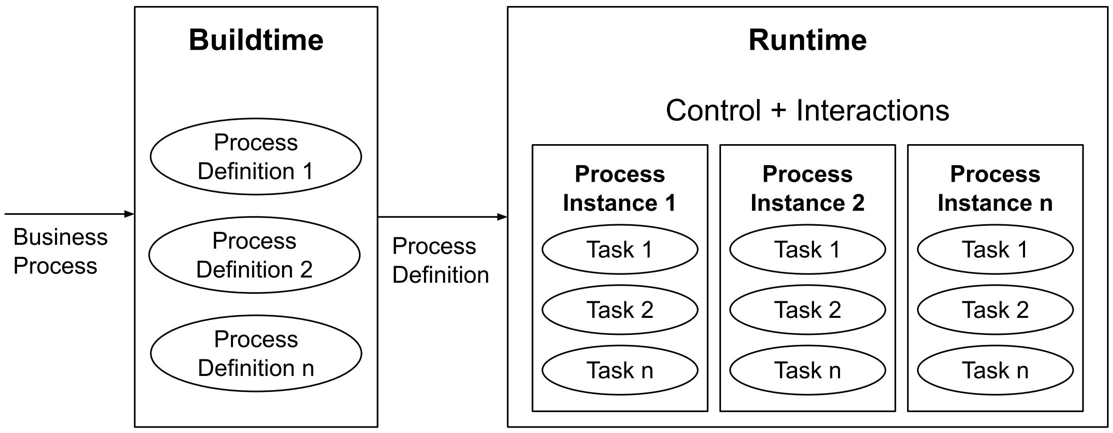


Figure 3.6. Workflow-Oriented Architecture

This model supports three functional areas: *buildtime*, *runtime control*, and *runtime interactions*. See Figure 3.6. The first two areas are likely to be centralized [2].

Buildtime covers workflows definition and modeling, i.e, the translation of a business process into a computer processable *process definition* [43]. A definition involves a series of discrete tasks -delegated to either human or software entities- and rules for tasks execution. Process definitions have a name, a version number, conditions for start and termination, and additional fields for handling security and auditing.

During **Runtime Control**, the platform interprets a process definition and executes new process instances, where task coordination and completion happen, by invoking the necessary people and software entities. The Business Process and Model Notation (BPMN) standard [5] enables the visual design of process definitions, which can be later translated to a machine-readable workflow representation in a Business Process Execution Language (BPEL)[6]. During runtime, *workflow control data* is persisted to allow execution auditing and recovery from failures, and a *Workflow Management (WFM) Engine* processes workflow navigation.

Runtime Interactions enable communication with users and software. User interactions are commonly concretized as a user interface, consisting primarily of a *worklist* assigned per user. *Invoked applications* are interfaces that enable invocation of remote applications as part of a task [2]. This framework enables the reuse of

[5]http://www.bpmn.org/

[6]https://www.oasis-open.org/committees/wsbpel

diverse distributed applications to work together, a crucial benefit of workflow platforms [43]. The flow of tasks then allows delegating work to different vendors and tools executed in completely separate environments. These are known as composite workflow applications.

An approach that belongs to this group provides a platform for modeling process definitions, an engine for executing and monitoring process instances, and provides interfaces to integrate human tasks and, more importantly, software tasks delegatable to remote components. This section examines two instances, namely *Camunda* and *BioExtract*.

Camunda[7] provides an architecture for building workflow applications and process automation. It implements widely recognized standards, such as BPMN, Case Management Model and Notation (CMMN) and Decision Model and Notation (DMN). It offers high customization of workflow modeling and execution, enables organizations to migrate legacy Business Process Management Systems (BPMSs) fairly straightforward, and to compose complex systems using diverse software products, by providing mechanisms for microservices orchestration, monitoring and management. Among Camunda's infrastructure components, the core is the *process engine*, accessible via a Representational State Transfer (REST) API to several end-user and developer applications, i.e., the *modeler*, *tasklist*, *custom applications*, *cockpit*, and *admin*.

The process engine executes BPMN processes, CMMN cases and DMN decisions, persists data in a relational database, and stores relevant artifacts in a file repository. It is composed of two sub engines accessible via a REST API to external applications, enabling cooperation and distribution of tasks to specialized microservices that can pull work from the engine, execute a task, and send a result back to the engine.

First, an engine for BPMN 2.0 processes execution, allowing the description of service orchestration, the flow of human tasks, event handling, and other workflow-related features. It offers integration with the DMN decision engine for executing decision tables within a workflow.

Second, a DMN engine that executes decision tables and decision requirements diagrams. The benefit of this feature is the possibility to easily create and maintain flexible executable business rules represented as DMN tables. Such decisions are evaluated automatically as part of processes.

Camunda includes a modeler to create BPMN workflows graphically and define their properties for technical execution. The *Cockpit* component is a dashboard to manage process definitions and running process instances, monitor task execution, and in general, control the overall process execution runtime. It enables discovery, analysis

[7] https://camunda.com/

and solving of incidents, by enabling the revision of execution logs, an inspection of flow history, and process modification and re-execution. For tasks completion, the platform offers two possibilities. First, the *Tasklist* component, a UI with functionalities to pull, lock, and complete assigned tasks. Second, developers can create custom UIs perform all these task-related steps by interacting with the process engine using REST calls.

A few pieces of evidence of use cases for Camunda are Zalando, T-Mobile, and 24 Hours Fitness. The former is an e-commerce company that implemented Camunda for internal processes in order to easily involve all stakeholders in their process and facilitating technical implementation. T-Mobile used Camunda for improving their service provisioning with automated order execution. The latter company uses the solution as a real-time orchestration system, enabling its staff to collaborate better and share the workload.

BioExtract is a web-based bioinformatics workflow platform [63], designed to support researchers to analyze genomic data, i.e., an organism's full set of Deoxyribonucleic Acid (DNA)[8], using bioinformatics workflows. Bioinformatics is an "interdisciplinary field mainly involving molecular biology and genetics, computer science, mathematics, and statistics[9]".

The benefits of using scientific workflows are: (1) the automation of time-consuming, repetitive, expensive tasks, and enabling the reproduction of scientific computations by formally describing analysis steps. With well-documented algorithms to reproduce experiments, scientists increase their readers' engagement with their publications; and (2), access to exponentially growing amounts of DNA data and analytic tools, which are distributed in a variety of formats, data structures, and software components, e.g., on-premise, websites, and web services. Researchers can interconnect components and data sources into automatically-executable workflows.

The platform offers flexible querying to multiple data sources and retrieving and filtering results. Query construction involves selecting one or more data sources and composing them. A query result is turned into a data extract that can be filtered, exported, and used as input to analytic tools. Search results can be stored and incorporated into the system as resources.

BioExtract offers access to external analytic tools and integrated tools. Tools execution begins with pre-set input parameters. The analysis results may be downloaded and used as input for a subsequent analyzing tool. Researchers execute sequential tasks that are recorded and automatically saved by the BioExtract server as a series of steps composed into a reproducible and sharable workflow. The physical represen-

[8]https://ghr.nlm.nih.gov/primer/hgp/genome
[9]https://www.ncbi.nlm.nih.gov/pubmed/24272431

tation of a workflow is a directed acyclic graph, where nodes depict tasks, and their connecting arcs represent dependencies of data.

3.2.2. Comparison

An evaluation of groups of approaches against each process-related requirement defined in subsection 2.2.1 is the next step. Table 3.4 summarizes the evaluation. Subsequent conclusions help to identify the requirements that are already satisfied by existing solutions, and the requirements that demand more attention as current solutions do not address them entirely.

RB1. Integration Platform

A suitable approach must connect distributed software components in a configurable fashion. Configurability, on the one hand, demands that components have inherently plug-and-play nature and become accessible to the platform dynamically via defined interfaces. Distribution, on the other hand, demands that components are physically spread and connected by a network.

Service-oriented approaches (GB1) use the term *Services* referring to software components that can execute one or more tasks, and the term *Microservices* for services with a single, small, and concrete task. By nature, services distributed and consumable. Service registries enable configurability, as new services can be registered and made available to consumers on the fly. These aspects allow a well-defined configuration of distributed software components, and therefore, they *fully satisfy* this requirement's criteria.

In Agent-oriented approaches (GB2), agents are autonomous software components that can interact with each other in an environment, which acts as a virtual shared infrastructure for execution. On the one hand, this architecture supports distributed agents since one can independently develop and deploy them. On the other hand, these approaches do not define mechanisms for agents configurability during system runtime. Therefore, this group *partially satisfies* the requirement.

Workflow-oriented approaches (GB3) state that workflows consist of a series of tasks and rules for tasks execution. Tasks can be completed either by a human or by a software component. This model enables the invocation of remote software components as part of task completion, satisfying the distribution criterion. Moreover, process definitions enable configurable tasks; one can rearrange software components in process definitions for later instantiation during the Runtime Control phase. With this, the configurability criterion is also satisfied. Having two positive criteria, GB3 *fully satisfies* this requirement.

RB2. Supporting a Wide Variety of Tools

This requirement defines two criteria to identify approaches that support configurable software components as part of a composite workflow. First, the approach must organize software components at definition time; and second, it must arrange components during execution time. Supporting a wide variety of tools means that the approach complies with both criteria.

Although Service-oriented approaches (GB1) offer registries for tracking available services and enables consumers to find providers, they do not consider mechanisms to arrange the order of services execution at definition time. However, more modern concepts such as the API-gateway introduce an execution infrastructure for adding metaservices -services that consume other services-, making services orchestration during runtime possible. However, the development of such orchestration components is not standardized. Therefore, this group *partialy satisfies* the requirement.

Agent-oriented approaches (GB2) implement agents embedded in an environment that acts as a shared infrastructure for execution. The environment signals the agents, triggering actions within them. Agents perform an action that may potentially alter the environment; they are self-organized and collaborate towards achieving a common goal, meaning that the environment does not arrange agents interactions. As it is not possible to set agents organization at definition time, this group only satisfies one criterion, and thus, it *partially satisfies* the requirement.

Workflow-oriented approaches (GB3) enable the design of process definitions to establish the order and style of tasks execution. Such ability automatically allows this group to satisfy the first criterion. Moreover, one can instantiate process definitions recurringly; a Runtime Control Engine handles execution orchestration, meaning that the second criterion is also positive, and in consequence, the group *fully satisfies* the requirement.

RB3. Results Aggregation

Combining object detection implementations at execution time demands a results unification after tasks completion. Unifying outputs is only possible after a conflicts resolution, is facilitated with uniform communication interfaces, and improves with a weights-distribution analysis.

Service-oriented approaches (GB1) utilize Descriptions to expose a service's capabilities, location, consumption details, and concrete input and output types. This architecture does not consider the aggregation of services; nevertheless, alternative solutions exist. On the one hand, one can build an independent meta service that consumes two or more other services and aggregates their outputs. The API-gateway pattern, on the other hand, enables server-side aggregation procedures through microservices composition; however, this is not standardized. These alternatives are not part of the architecture by nature, and thus, the group *does not satisfy* the requirement.

In Agent-oriented approaches (GB2), an environment triggers agents with specific roles to execute a specific task; after tasks completion, agents can alter the environment by posting results. The environment can process results, such as building aggregations. In Cougaar, agents post results to the global blackboard, which aggregates results. In GEAMAS, the architecture's first level collects outputs from its agents society, making possible an aggregation. However, a limitation is that the system designer must do such aggregations. Consequently, GB2 *partially satisfies* this requirement.

Workflow-oriented approaches (GB3) use Process Definitions in written in standardized computer-readable representations to describe parallel gateways for forking the execution; each path executes a thread of tasks, and afterward, the gate joins back the execution into a single thread. One can add a subsequent task in charge of computing aggregations. These are system-specific implementations, as the architecture itself does not define a procedure for aggregation of outputs. Therefore, the group *partially satisfies* the requirement.

RB4. Conflict Resolution

A suitable approach includes a conflict-resolution component that describes rules to disguise conflictive results from the outputs of a series of software components that perform similar tasks, as well as defining mechanisms to make decisions in instances of each dimension in its domain of study. A mechanism of this type can be a rule-based or an algorithm-based decision-making subprocess situated towards the end of a macro process and functioning as a critical step concerning results unification.

Service-oriented approaches (GB1) provide a flexible solution for resolving conflicts in outputs of different services. Creating one independent meta service that resolves inconsistencies is possible. Furthermore, the API-gateway model is an alternative solution if the architecture uses microservices. Nevertheless, there is no clear mechanism to develop this, and therefore, this group *partially satisfies* the requirement.

Agent-oriented approaches (GB2) implement agents in an environment; the second one triggers its composing agents with specific roles to execute a specific task, and after tasks completion, agents can alter the environment by posting results. The environment can use the results to compute, among others, conflict resolution and results aggregation. Consequently, the group *fully satisfies* this requirement.

Workflow-oriented approaches (GB3) utilize Process Definitions. One can include parallel gateways for forking the flow of execution, where each path executes one series of tasks. After each composing thread completes its execution, the gate joins them back into a single thread. In the definition, one can place a custom conflict-resolution task after the gate. Nevertheless, a custom implementation is still necessary; thus, this group *partially satisfies* the requirement.

RB5. Weight Distribution Analysis

A suitable weight-distribution analysis considers two criteria. First, the preparation of the analysis scenario: the inputs and arrangement of software components; and second, the evaluation of results based on metrics.

Service-oriented approaches (GB1) allow finding available role-specific services in a Registry, enabling the development of, for example, a macro service that consumes different combinations of services arranged as series of sequential tasks and obtains one result per combination. With the API-gateway model, microservice-based architectures provide a solution with a similar outcome; it enables services orchestration and comparison of results against the results of individual services. However, this demands custom development efforts, making this group *partially satisfy* the requirement.

In Agent-oriented approaches (GB2), an environment can find role-specific agents and request them to execute their tasks; the environment waits for agents to provide individual results. This situation can be useful for measuring the quality of results obtained by the chained execution of multiple agents. However, this strategy lacks standardization, and therefore, this group is said to *partially satisfy* the requirement.

Workflow-oriented approaches (GB3) offer a high flexibility of tasks configurations through Process Definitions; an evaluation of collaborative results is achievable through the creation of several definitions, where each contains one combination of tasks. One can compute comparisons afterward. However, this architecture does not consider the creation of these combinations of tools; one must manually model and instantiate multiple processes definitions. Consequently, this group *partially satisfies* the requirement.

RB6. Uniform Communication Interfaces

A suitable approach ensures tools interoperability by defining a set of uniform representation formats and messaging protocols for integrating diverse software components.

Service-oriented approaches (GB1) utilize Descriptions that contain metadata about a service's functionality, its category, and its interaction prerequisites. This idea conforms a uniform communication mechanism as it includes fixed representation formats and messaging protocols used for service consumption. By satisfying both criteria, this group *fully satisfies* the requirement.

In Agent-oriented approaches (GB2), the message transport, message format, message content layers handle inter-agent communication. The FIPA framework standardizes these layers; the first layer handles, among other aspects, the messaging protocols for agents communication. With the use of FIPA-ACLs for message formatting, and FIPA-SL and system-specific ontologies, this framework considers all the necessary aspect to describe uniform inter-agent communication mechanisms. Therefore, it *fully satisfies* the requirement.

Workflow-oriented approaches (GB3) describe tasks configuration, and aspects such as their input and output types, using standardized languages, e.g., BPMN. Moreover, most of these approaches have implementation-specific and standardized messaging protocols. This group *fully satisfies* the requirement.

3.2.3. Conclusion

At least one group of approaches fully solves requirements RB1 (integration platform), RB2 (support for wide range of tools), RB4 (conflict resolution) and RB6 (uniform communication interfaces). On the other hand, no group addresses entirely the requirements RB3 (results aggregation) and RB5 (weights analysis). From Table 3.4, one can infer that Workflow-oriented approaches (GB3) have the best evaluation scores, followed by the Service-oriented (GB1) and Agent-oriented (GB2) groups, in order. Therefore, an ideal integration platform is to be designed using a combination of approaches, replicating ideas of each group that fit the best to each requirement.

Conceptualizing a suitable architecture for diverse software components is, in general terms, a problem solvable with Service- and Workflow-oriented approaches as they can join distributed and configurable services and software tools, respectively. On the other hand, Agent-oriented approaches allow distributed agents but lack configura-

	GB1. Service	**GB2. Agent**	**GB3. Workflow**
RB1. Platform	++	+	++
RB2. Variety	+	+	++
RB3. Aggregation	-	+	+
RB4. Conflicts	+	++	+
RB5. Weights	+	+	+
RB6. Uniformity	++	++	++

++ fully satisfied, + partially satisfied, - not satisfied

Table 3.4. Groups of Approaches Evaluation - Combining UI Object Detection Analyzers

bility since agents are self-organizing and proactive. The principal architecture of an integration platform must be built using the ideas of one of Service- and Workflow-oriented groups, or a combination of both.

An ideal integration platform organizes and orchestrates several software tools with independent implementations. Service-oriented platforms can facilitate tools arrangement with the API-gateway model. Agent-oriented solutions partially help in this context, as agents' organization demands development efforts; agents should be smart enough to understand the environment and act proactively, which is a complex development task. Workflow-oriented approaches are the way-to-go in this area, as the concepts of process definitions and process instances allow tools arrangements by design.

Assuming a platform integrates distributed and heterogeneous tools, it must be able to aggregate tools outputs and provide a unified system result. Service-oriented approaches do not consider such scenarios by design unless the system creator includes them; an alternative is implementing the API-gateway paradigm. On the other hand, both Agent-oriented and Workflow-oriented approaches enable aggregations via an Agents Environment and Runtime Control, respectively. However, one must manually design and develop such aggregations as these groups do not include them by design. None of the SotA groups complies entirely with RB3, and thus, this thesis must combine groups to solve it.

When joining outputs of different tools, conflicts naturally emerge; resolving them is possible using the concepts of the three groups of approaches. Alternatives include using: the API-gateway in Service-oriented platforms, environments or blackboards in Agent-based approaches, and parallel gateways in Workflow-based solutions. In Agent-oriented approaches, the environment can use the results that agents post to compute, among others, conflict resolution. The concept of an aggregating-environment can be replicated to solve conflict resolution.

Service-oriented and Agent-oriented approaches provide alternative mechanisms to compare outputs of different combinations of software tools, with the limitation of not being standardized. Workflow-oriented approaches, on the other hand, provide a flexible and standardized mechanism to test several combinations of process tasks and obtaining unified outputs per combination, enabling their comparison; this group is, therefore, a suitable pattern to solve requirement RB5. Nevertheless, one must still consider that these approaches provide generic architectures and do not provide system-specific applications.

Service Descriptions in Service-oriented approaches cover aspects for message representation formats and messaging protocols. The FIPA framework in Agent-oriented platforms embraces these two aspects as well, using communication and semantic languages. Workflow-oriented approaches describe tasks configuration, and aspects such as their input and output types, using standardized languages, and have implementation-specific and standardized messaging protocols. Consequently, an ideal solution for RB6 ensures tools interoperability by taking a combination of the ideas of all groups of approaches.

3.3. Summary

This chapter identifies existing approaches for potentially solving this thesis' requirements and groups them based on shared characteristics. Each group exhibits a set of criteria to disguise whether an approach belongs to it, and a description of a few concrete group representatives. An evaluation of groups of approaches against the requirements defined in chapter 2 is the following step; it serves to identify the requirements that are already satisfied by existing solutions, and the requirements that demand more attention as current solutions do not address them entirely.

The next chapter presents alternative solutions combining the ideas of the groups of approaches that better fit the thesis' requirements, it compares them, and lastly, conceptually drafts the most suitable solution.

4. Concept

Learnings obtained in chapter 3 lead to identifying the requirements that are already addressed by groups of approaches and the requirements that are still not entirely solved. This chapter conceptualizes a solution that aims at meetings all this thesis' requirements. Each thesis solution part identifies and outlines potential concept candidates; this includes an overview of the candidate and a few possible concept designs. Each candidate is submitted to discourse to decide which one is the most suitable. This chapter finally drafts the fittest solutions abstractly.

4.1. Top-Level Architecture

This section proposes a top-level architecture that aims at providing a solution aligned with the thesis' requirements and objectives. Meeting all the thesis' requirements demands to build a solution composed of **two subparts**.

On the one hand, the **first solution part** consists of designing a DNN-based UI object detector able to learn progressively. The SotA analysis concludes that DNN-based object detection techniques are reliable for generic object detection; this part attempts to replicate this knowledge in the specific domain of UI object detection. Reusing one-stage or two-stage detectors is feasible for UI parts detection; however, this statement demands further testing. Moreover, there is currently no existing reusable dataset to train an object detector in the current domain, and consequently, the collection of a sufficient training dataset requires more effort. Automated dataset collection approaches suit better the needs at this matter; however, a combination of approaches is still possible.

On the other hand, the **second solution part** requires the conceptualization of an integration platform for UI object detectors. The SotA analysis exposes there that four out of the six process-related requirements for this part are entirely addressed by at least one of the researched groups of approaches. Nevertheless, two requirements still demand special attention. There is currently no group of approaches that entirely fits the solution that this part demands; one can infer that Workflow-oriented approaches have the best evaluation scores, followed by the Service-oriented, and

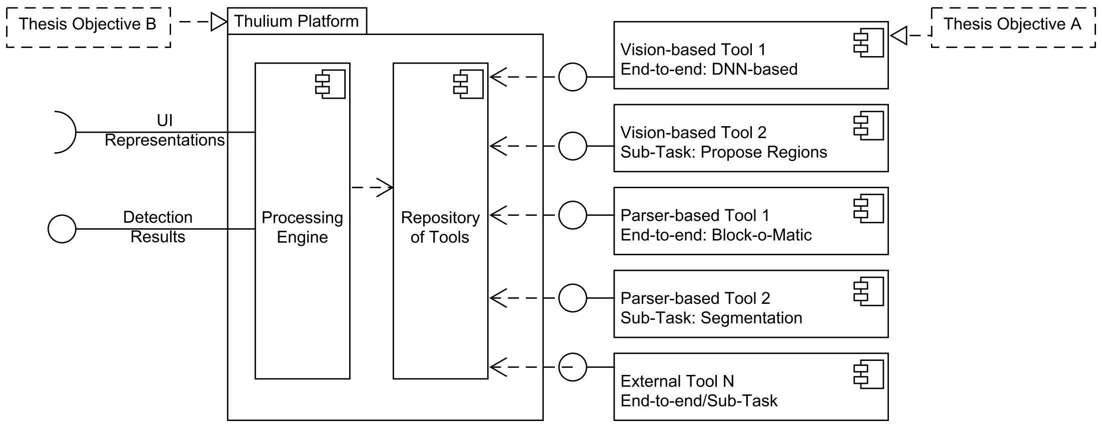


Figure 4.1. Top-Level Solution Architecture

Agent-oriented ones, in order. An ideal integration platform is to be designed using a combination of these approaches.

The proposed **top-level solution architecture** in Figure 4.1 presents a solution for the thesis' requirements and objectives; it includes a DNN-based object detection tool, and an integration platform to enable tools combination. The platform consists of an infrastructure to virtually arrange and coordinate multiple end-to-end and divisible UI object detection tools into single execution processes. Such processes take UI representations as input, arrange tools execution, and output detection results. Object detection tools are independently developed and deployed in separate environments, demanding communication between platform and tools over a network. Of these remote tools, the most important for this concept is the end-to-end DNN-based object detection tool.

In the following two subsections, each thesis solution part is individually analyzed deeper to achieve a concept that fulfills all the requirements exposed in this work. section 4.2 examines a concept for achieving objective A, while section 4.3 investigates a concept towards accomplishing objective B.

4.2. UI Object Detection Using DNNs

Part written by Juan Sebastian Quintero

Knowledge acquired in the SotA analysis section corresponding to the thesis' first path of work, i.e., UI Object Detection Using DNNs (see section 3.1), included the analysis of existing approaches that are considered cutting edge as well as and their possible adoption to solve partially or entirely the requirements conditioned by an ideal solution (see section 2.1). The conclusion from such analysis showed that there are robust approaches for detecting generic objects; nevertheless, training them in a progressive manner can still be challenging because there are several intermediate actions that need to be considered. Also, the fastest way to collect a sufficient training dataset in the thesis domain is possible through an automated approach; however, other ways of dataset collection are still valid since the different approaches are not mutually exclusive.

In this section, the identification of a set of solution candidates that address the defined requirements having as a base the approaches learned in the state of the art analysis chapter is conducted. In subsection 4.2.1, an overview of each of these candidates with their respective advantages and disadvantages is presented. Afterward, in subsection 4.2.2 a discussion in which a decision for the most suitable alternative for the problem this thesis wants to solve is held. The most suitable solution is further drafted conceptually in subsection 4.2.3.

4.2.1. Possible Solutions

For defining a solution that enables object detection by using DNNs from a conceptual level, a set of alternatives that address this goal considering the findings in the SotA analysis chapter is proposed. It is important to keep in mind that the solutions presented here are abstract, theoretical solution concepts which have not yet been implemented up to the full extent detailed in this section. The upcoming alternatives' descriptions take into consideration the requirements described in section 2.1 in combination with the findings described in section 3.1. Since the work in this part involves the creation of a concept that contains both a detection mechanism of UI parts by using DNNs and the collection of a sufficient dataset, the problem is divided into two subpaths. For each of the paths, a pair of alternatives is presented.

The following alternatives comprise the ones evaluated for the object detection mechanism:

- Object Detection using a One-stage Approach
- Object Detection using a Two-stage Approach

Subsequently, the alternatives considered for the dataset collection approach are listed here:

- Dataset Collection through Crowdsourcing

- Dataset Collection through WUIs DOM Manipulation

The upcoming alternative descriptions take into consideration the requirements described in section 2.1 in combination with the findings for this path of work described in section 3.1. In subsection 4.2.2, a discourse of alternative solutions is presented, and the most suitable ones in their respective subpaths are selected. Subsequently, an architectural draft outlining the combination of the selected alternatives is exposed.

Object Detection Alternatives

This subsection introduces a pair of alternatives that address the object detection problem. The former alternative constitutes a one-stage object detector that can learn progressively. Following similar guidelines, the latter one encompasses a two-stage object detector that is also able to learn progressively. Both alternatives are explained below.

Object Detection using a One-stage Approach

One-Stage detectors, also known as *Region-free Based* detectors, are distinguished because of their *unified* detection strategy. It means that both classification and location of objects are performed with a single CNN in a monolithic setting that avoids the computation of region proposals at separate stages [3].

An alternative for solving the detection of UI objects is using one of those detectors. According to the research previously done in the literature, it is found that the most prominent approaches following a single-stage detection strategy are **YOLO** [81] and **RetinaNet** [57]. While the former characterizes itself for being the fastest object detector and ideal for scenarios that require real-time processing, the latter can obtain more accurate results while having good speed rates.

Since most accurate object detectors are traduced in higher precision and recall rates which are vital for fulfilling the result-related requirements (see subsection 2.1.3), the **RetinaNet** approach is adopted for this alternative. Adopting this object detector implies fulfilling all of the object detection requirements defined in subsection 2.1.1.

The overall structure of the RetinaNet looks as follows:

- An **FPN** [56] on top of a **ResNet** [40] that are in combination the *backbone* that allows future extraction.

- A pair of **Fully Convolutional Networks (FCN) subnetworks** per each FPN level that are in charge of classification and localization respectively

In addition to the object detector, it is also essential to consider ways that allow training it progressively. Such aspect is vital for meeting the requirement "RA8. Avoid Re-training from Scratch". Hence, it is essential to provide means that allow the detector to be both trained and used on-demand. Allowing the model to be trained progressively adds different challenges, such as handling the training data in terms of storage and management, and feeding the model with the training data to learn from it. For it is vital to contemplate a mechanism that on one side, allows the training data to be managed by resources with high storage capacity, and on the other side allows both the training and inference processes to be handled by resources with high memory and processing power.

For such a set of reasons, it is also considered:

- An **API Gateway** That exposes methods for (1) storing training examples and (2) invoking inference process given an image. It should also integrate the underlying dataset storage and object detection services.
- A **Dataset Service** that is in charge of managing the lifecycle of the training examples since its initial storage to its further deletion when not needed anymore.
- An **Object Detection Service** that wraps the object detector implementation and provides an inference method and a background training strategy.

Object Detection using a Two-stage Approach

Two-stage detectors, also known as *Region Based* detectors, are distinguished because, in the detection process, two phases are performed one after the other. The first phase corresponds to extracting region proposals, and the second one corresponds to the detection of objects within those proposals [3].

An alternative for detecting UI objects is using one object detector under this category. The most suitable two-stage object detector for this case is **Faster-RCNN** since this is the most accurate two-stage object detector found in the literature [83, 46].

In a similar fashion as in "Object Detection using a One-stage Approach", this object detector is considered here given the fact that most accurate object detectors obtain higher precision and recall rates which are vital for fulfilling the result-related

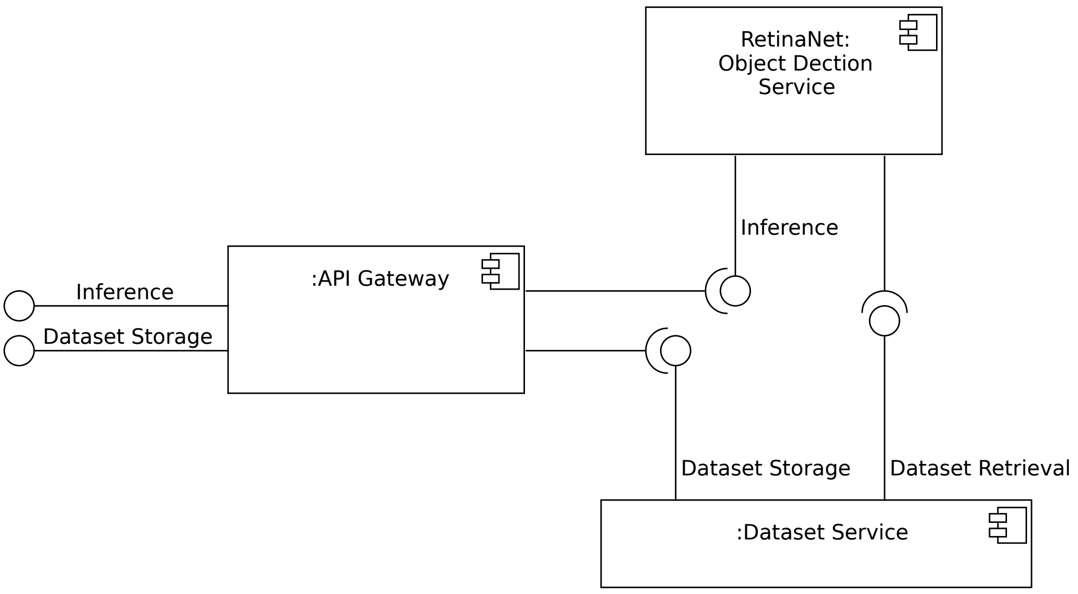


Figure 4.2. Object Detection with RetinaNet - Component Diagram

requirements (see subsection 2.1.3). By using this object detector, all of the object detection requirements defined in subsection 2.1.1 are met instantly.

Since the training considerations exposed in “Object Detection using a One-stage Approach” stay the same, the architecture proposed exposed there that surrounds the object detector is reused leaving the object detector as the only part that changes in this alternative.

Dataset Collection Alternatives

This subsection describes two alternatives that address the challenge of collecting a training dataset. The first alternative constitutes a dataset collection approach by following a crowdsourcing technique. The second one considers a purely automated dataset collection approach. An overall picture of both of the alternatives is outlined in the following paragraphs.

Dataset Collection through Crowdsourcing

Crowdsourcing is a model that allows the solution of problems in a distributed and *outsourced* fashion with the help of the *crowd* [10].

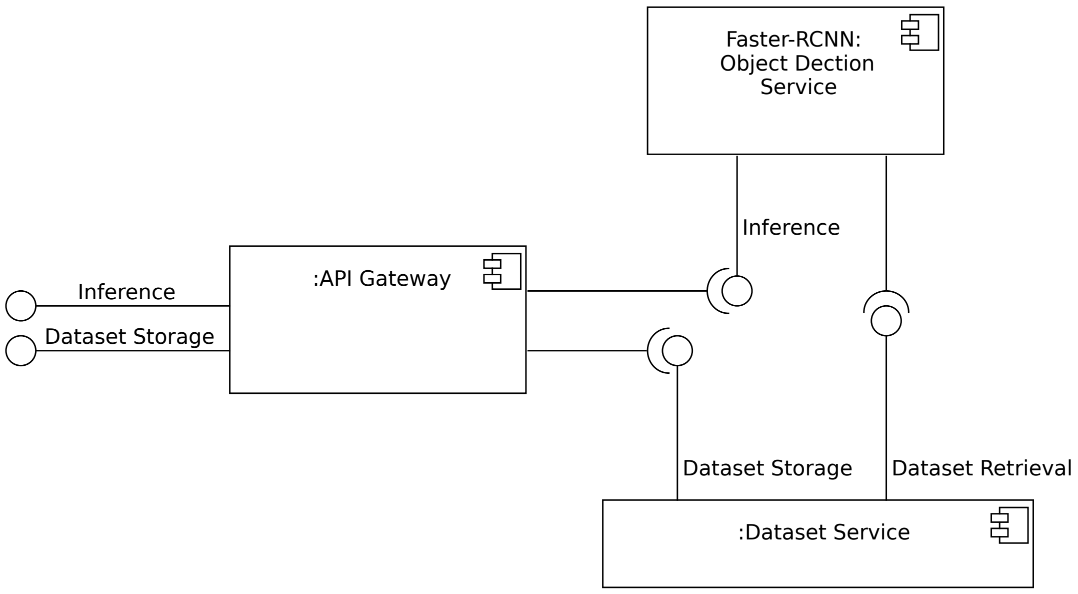


Figure 4.3. Object Detection with Faster-RCNN - Component Diagram

Approaches following this model can vary enormously; however, they share something in common: A description of a task that needs to be performed, and the reward that is given in return for performing the task. In the context of training dataset collection, the overall task description is clear: performing acquisition/labeling of WUIs. This task can be tedious, so there is a need to find a mechanism that enables the labeling experience to be more pleasant for users.

For such reason, the term *Gamification* is introduced. Gamification refers to the use of video game elements in non-gaming systems to improve user experience and user engagement [21]. Such elements can include point scoring, competitions, playing rules, among others.

By having this into consideration, a possible data collection approach is a gamification platform that enables the manual labeling of different WUIs by presenting them randomly to a set of users which perform the labeling and are rewarded with points. Competitions can be built around those points, allowing the winners to exchanging points for products or services.

Taking for granted user acquisitions, the more challenging aspect of implementing this approach becomes controlling the correctness of the labeling that each of the users performs. By having this in mind, it is necessary to bring a mechanism that allows such control.

For such goal, a consensus mechanism is proposed by introducing a **Calibration** component that checks how "calibrated" are the labeling results of different users by (1) presenting the same WUI to be labeled to a set of different and randomly assigned users in order to check similarity in the results and (2) by having a "partial ground truth" that comes from a pre-analysis of the WUI which also allows an additional verification mechanism. It is called *partial* because, given the heterogeneity of how WUIs can be built, it is not possible to construct a 100% accurate ground truth, but instead, an approximation. In the end, labelings are cataloged as correct if their similarity across different users is high, and also surpass a defined similarity threshold against the "partial ground truth".

In summary, the considered solution alternative for data collection consists of:

- A **WUI Analyzer** that is in charge of visiting websites, and analyzing them to store additional data about the WUI which includes "partial ground truths", a corresponding screenshot, and related metadata.
- A **Web Platform** that offers authentication and authorization mechanisms to users, and also provides a friendly UI for labeling WUI parts as well as managing accumulated points for further redemption.
- A **Calibration** component that checks consensus among the labelings performed by the different users to determine if the labelings can be (1) considered for further training and give the respective reward to users or (2) discarded.

An summarized illustration of the components can is exposed in Figure 4.4.

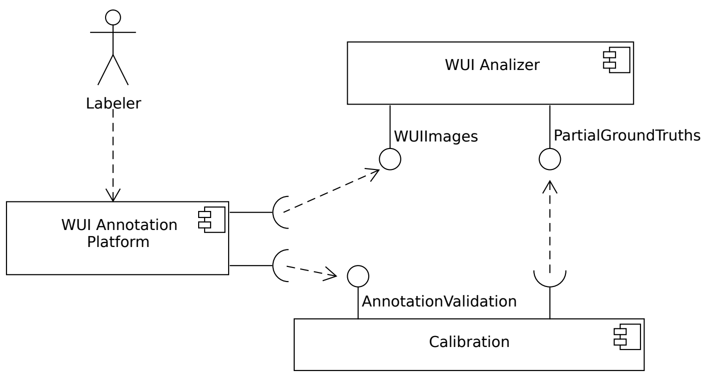


Figure 4.4. Crowdsourcing-based Dataset Collection - Component Diagram

Dataset Collection through WUIs DOM Manipulation

Another feasible data collection approach is one where the dataset is collected right away from automated approaches. This approach allows the acquisition and labeling of a dataset with sufficient size while needing minimum human efforts (see requirements RA5 and RA7).

Automated approaches for collecting training datasets are those in which data is synthetically generated and further labeled by machines. Thus, no human intervention is required.

The applicability of automated approaches for dataset collection depends merely on two different aspects: (1) the domain of knowledge, and (2) the lack/existence of a current dataset. When the new data generated depends on existing data, it receives the name of *Data Augmentation*, and when the data is generated from scratch, it is called as *Data Generation*.

Fortunately, the specific problem domain that is targeted allows both the creation and parsing of WUIs by using well-standardized APIs such as the DOM. In this case, the approach can be cataloged as *Data Generation* because the data is generated from scratch.

However, the most challenging aspect that arises for an automated dataset collection approach is the several ways in which WUIs can be built; said that it is essential to have mechanisms that consider this factor.

For addressing such problem, having different parsers targeting different kinds of WUI frameworks/libraries is a logical approach given the volume of websites built using such tools. However, the problem is not entirely resolved since the spectrum of websites built in customized ways is still huge. Besides that, there are cases where it becomes impossible to identify framework/library-related aspects because of (1) obfuscation techniques applied over website artifacts and (2) The extension of the frameworks/libraries for customized look-and-feels.

To mitigate this, a WUI generator that creates a WUI with a certain distribution and look-and-feel given some randomness seed is considered. Adding this offers the possibility to both generate and label WUIs in a controlled manner.

An overview of the proposed alternative includes:

- A **WUI Generator** that given some randomness seed in a Uniform Resource Locator (URL) through a HTTP GET request, is capable of generating sets of WUIs with look-and-feel variations and different distribution of elements.

- A **WUI Analyzer** that is in charge of visiting URLs, and analyzing their content to delegate their parsing and labeling tasks to the correct parsers and writers. It is also in charge of resolving conflicts across outputs given by different parsers (ground truth overlaps) and computing the statistics that help to control the frequency of distributions' levels.

Such components are illustrated in Figure 4.5.

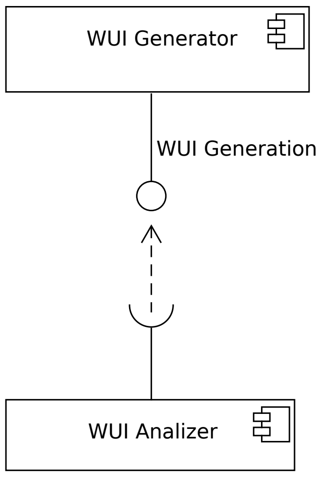


Figure 4.5. Automated Dataset Collection - Component Diagram

The WUI Analyzer consists internally in:

- A **URL Registry** that enables both adding and querying URLs through a registry.
- Several **Writers** that write resulting data about the UI which includes respective labeling (bounding boxes and class labels), a corresponding screenshot, and related metadata in a particular format and structure.
- Different **WUI Parsers** That implement parsing specificities of different types of WUIs e.g framework-specific or generic HTML5 and suggest a set of ground truths i.e. the different elements present within a WUI with their respective locations.
- A **Web Crawler** that browses the web starting from an initial list of URLs and looking for new ones so that it can add them to the URL registry.

4.2.2. Discourse

After defining a set of alternatives, a comparison can be made to select the most appropriate ones. The best combination of alternatives is based on the two particular subpaths that this part of the work demands being (1) A suitable DNN-based object detector for UI objects and (2) A feasible data collection approach. This comparison can be performed thanks to the knowledge acquired in the SotA analysis chapter, especially from the conclusions stated at subsection 3.1.3.

Object Detection using a One-stage Approach

With the RetinaNet approach, all of the object detection requirements are well suited (see subsection 2.1.1) Because it counts with a deep learning architecture capable of detecting objects in an automated fashion given a visual representation and can also be trained for detecting UI objects.

An essential aspect of the RetinaNet object detector is its focal loss function, which allows lowering loss contributions provided by *easy* negative examples that tend to be very high in the case of one-stage object detectors. It means that the detector is capable of learning only from *hard* examples. This fact is significant for accomplishing the result-related requirements since it contributes to its performance in terms of accuracy (see subsection 2.1.3).

Another advantage is that thanks to its FPN, It can have separate feature representation at different scales. It allows the detection of objects at different sizes, and especially the detection of small objects which contributes to having better recall rates. Hence, it is beneficial for meeting the requirement *RA10. High Recall.*

Thanks to its architecture, it can be trained in an end-to-end way using standard techniques such as mini-batch SGD. Additionally, the proposed set of services surrounding the detector itself are also important for allowing the object detector to learn progressively. This becomes critical for fulfilling the requirement *RA8. Avoid Re-training from Scratch.*

Exposing an API facilitates the integration of the object detection system with third-party systems as well. It becomes important having into consideration that this object detector is intended to be used by an object detection aggregation platform analyzed in more detail in section 4.3. Also, it allows encapsulating different pre-processing and post-processing actions before and after the training and inference subprocesses.

Furthermore, given the fact that there are many variables involved in the comparison of object detection techniques, it is hard to tell in advance if the RetinaNet approach performs better than other approaches in the particular domain of detecting UI objects without actually training and testing it with the training dataset. Hence the advantages described are only treated as general guidelines and need to be proved.

For such reasons, this alternative is considered as feasible and is included in the conceptual draft.

Object Detection using a Two-stage Approach

Faster-RCNN object detector, fulfills well all of the object detection requirements (see subsection 2.1.1) Because it also counts with an architecture considered deep that can detect objects given an image and also allows being trained for detecting UI objects.

The most prominent aspect of this approach is indeed its RPN because this network improves enormously the time taken to compute region proposals, the main bottleneck of Fast-RCNN. Furthermore, this network counts with a feature called *invariant translation anchors* that allows the detection of region proposals that are invariant to object translations within an image. It contributes to avoiding overfitting in small datasets [83]. Overfitting dramatically hits the performance of the detector in terms of precision and recall and thus impede the fulfillment of the result-related requirements (see subsection 2.1.3).

Being a two-stage object detector adds the drawback of being slower at the training and inference phases when comparing it with one-stage object detectors. The speed can be increased by lowering the number of RoIs proposed by the RPN, but it can affect the recall of the detection model.

Since this alternative only changes the object detector type, while keeping the surrounding services around it to facilitate its integration and training it progressively, it shares all of the advantages already explained in *Object Detection using a One-stage Approach* that those services add.

Sharing the same position as in *Object Detection using a One-stage Approach* it comes hard to tell if Faster-RCNN are better or worst compared to other object detectors and the only way to prove it is by testing it against the training dataset. Hence this object detector is also taken into account for the conceptual draft of this part of the solution. It is beneficial because it enables the comparison of two different object detectors against the training dataset. Also, both object detectors that can be easily integrated into the object detection aggregation platform explained in section 4.3.

Dataset Collection through Crowdsourcing

Analyzing a crowdsourcing platform that allows the manual labeling of WUIs, it meets the requirement *RA5. Sufficient Training Dataset Size* as well as the requirement *RA6. Sufficient Training Dataset Frequency of Distribution* by having for granted that a considerable amount of people is available and also that there are mechanisms in place that give statistical feedback about the training examples per each of the classes described at Table B.1. However, this solution does not meet the requirement *RA7. Training Dataset Collection Efficiency* because having a bunch of people increases the human effort that is needed having as a consequence a downgrade in the efficiency of the process itself.

Following the *consensus* approach that was proposed still introduces some drawbacks to the whole process since in order to have a calibration mechanism based on checking the similarity of the labeling task accomplished by different users, it means that several users need to label the same sets of UIs, which makes the whole process even more expensive and slower. Nevertheless, this is necessary when it is the only unsupervised and also scalable method to validate the correctness of the ground truth. An interesting idea to minimize such duplicate work can be presenting slight variations of the same visual representation of the user interface to different users by using data augmentation techniques such as color and saturation variations. It can result in users labeling WUIs that are equal in the distribution and location of the different UI parts, but different in terms of color and saturation.

Despite the method employed for validating the correctness of the proposed ground truths, the involvement of people in the process is not desired since it downgrades the efficiency of the process in terms of costs and time compared to automated approaches or the reuse of an existing dataset. Therefore, this approach is discarded for the architectural draft of the concept for a training dataset collection mechanism.

Dataset Collection through WUIs DOM Manipulation

A dataset collection approach based on automated labeling of WUIs by using the DOM fulfills well the requirement *RA5. Sufficient Training Dataset Size* thanks to the automated nature of the whole execution process. It also suits the requirement *RA6. Sufficient Training Dataset Frequency of Distribution* if there are mechanisms in place that at least control minimal threshold values for each of the classes described at Table B.1. Besides, the requirement *RA7. Training Dataset Collection Efficiency* is also fulfilled because it demands a minimum human effort. The only human effort required is related to the development of the software that handles both the acquisition and labeling of the dataset.

Having a flexible architecture where different UI parser implementations can be added, mitigates the problem of having to deal with different user interfaces created in different ways. However, it is also important to consider means to resolve conflicts across different parsers e.g., select the right ground truth when some of them overlap because of similar predictions by different parsers.

Given the facts exposed above, this alternative is the one selected for collecting the training dataset, and its respective architectural details are described later in subsection 4.2.3.

4.2.3. Draft

To solve the previously described problems, an architectural draft that contains (1) an **Object Detection** mechanism and (2) a **Dataset Collection** approach is proposed.

Object Detection

This section of the architectural draft encompasses all details in a conceptual level for an object detector that can learn progressively, i.e., Retains knowledge from previous training sessions. Independently on the object detection approach, this concept contemplates a solution that allows:

- Training an object detector progressively.
- To handle training data that is used for the learning process of the object detector.
- Invoking the inference process in the object detector.

On one side, the training and inference processes of the object detector demand high processing power usually carried out by Graphical Processing Units (GPUs), or by Tensor Processing Units (TPUs). On the other side, handling the training dataset requires a high storage capacity. That is why it becomes ideal that the solution allows those services to be deployed independently. Said that this thesis proposes two separate services; one for the training dataset management, and another for the object detection processes, including both training and inference.

In addition, it is also important to consider that the object detection solution provides a straightforward integration mechanism with third-party systems. The main reason for this is allowing it to be integrated without much effort into the object detection aggregation platform (see section 4.3).

Under the premise that the solution must offer a mechanism that (1) enables exposing separate services with different computing and storage requirements within a unified interface and (2) allows an easy integration from third-party systems, the *API Gateway* pattern is adopted. This pattern defines a service that provides a unique entry point for services with independent lifecycles. This pattern shares similarities with the *Facade* pattern used in object-oriented design, but with the difference of being applied for distributed architectures [20]. It defines the exposure of services following the REST principles which are beneficial in this context for several reasons:

- It offers a *Client-Server* model that allows separation of concerns, enhancing both the server (the whole service described here) and client (third-party applications that integrate with it) to be handled independently.
- Being *Stateless*, the server does not store any state associated with particular clients within invocations.
- It offers *Technology Heterogeneity* between server and clients in a lightweight fashion thanks to the adoption of existing protocols such as HTTP[1]. It is suitable because it does not limit the technologies in which third party systems are implemented for the sake of integrating them with the object detector.

The **Dataset Service** should count with high secondary memory, and expose methods for handling the lifecycle of *training examples*. A training example is a registry that comprises an image of a UI and its associated labels. The image has a specific width and height, and the labels are the set of UI objects that are present in the image with their respective locations encompassed by bounding boxes.

The **Object Detection Service** should count with high primary memory and processing power and expose an API that enables invoking of its inference process. The inference process consists of predicting all the UI objects with their respective locations that are present in a given image. It means that given an unlabeled image, the inference process builds the labels associated with it.

Furthermore, the object detection service should continuously poll training examples from the dataset service to build a batch sufficient in size for starting its training process. The size of the batch must be configurable so that it allows a real *online* training when the size of the batch is 1, or a *mini-batch* training when the size is greater than 1. Having an internal component that handles all the training related tasks is essential to allow the object detection service to learn progressively and thus, contribute to the fulfillment of the requirement "RA8. Avoid Re-training from Scratch". The training process is illustrated in Figure 4.6.

[1]https://tools.ietf.org/html/rfc2616

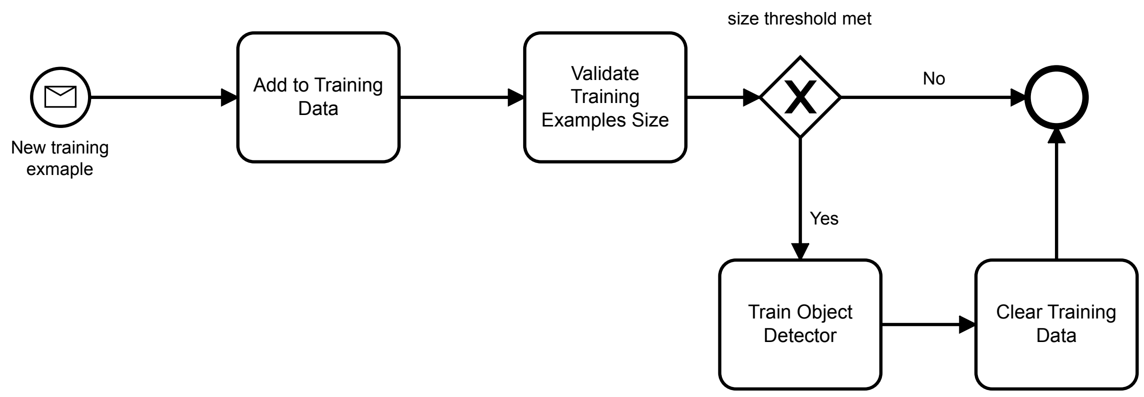


Figure 4.6. Training Process - BPMN Diagram

The **API Gateway** should provide methods for (1) uploading training examples to the dataset and (2) Invoke the inference process with the current model knowledge. A small illustration of the components described can be seen in Figure 4.7.

Additionally, two object detection services are proposed. The former being RetinaNet the latter being Faster-RCNN. The next paragraphs contain a description of both object detectors and an analysis of how they address the requirements.

On the one hand, **RetinaNet** detects objects by receiving an image for which initial future maps are computed with the help of a Residual Network (ResNet). After that, the feature maps go trough a FPN so that a new set of multi-scale feature maps are computed. Finally, each scaled feature map in the set goes through a pair of subnetworks that compute classes within the image and their respective bounding boxes. This is revealed in Figure 4.8 and further internal details can be checked in [57].

On the other hand, **Faster-RCNN** also receives an image and extracts feature maps with a feature extraction network. Having the feature maps computed, the RPN uses them for computing a set of RoIs that consist of bounding boxes with an *objectnes scores* which determine the probability that a bounding box contains an object. Subsequently, the RoIs are projected over the computed feature maps, and with the help of RoI pooling, a vector containing sections of the feature maps encompassed by the RoIs is calculated. From this vector, each feature maps section is employed by a fully connected network that estimates both class probabilities and bounding boxes. This can be visualized in Figure 4.9 and additional details can also be checked in [83].

The requirement “RA1. Detection of the Parts of a UI” is addressed well by RetinaNet and Faster-RCNN in the sense that they can be used in any domain. It means that

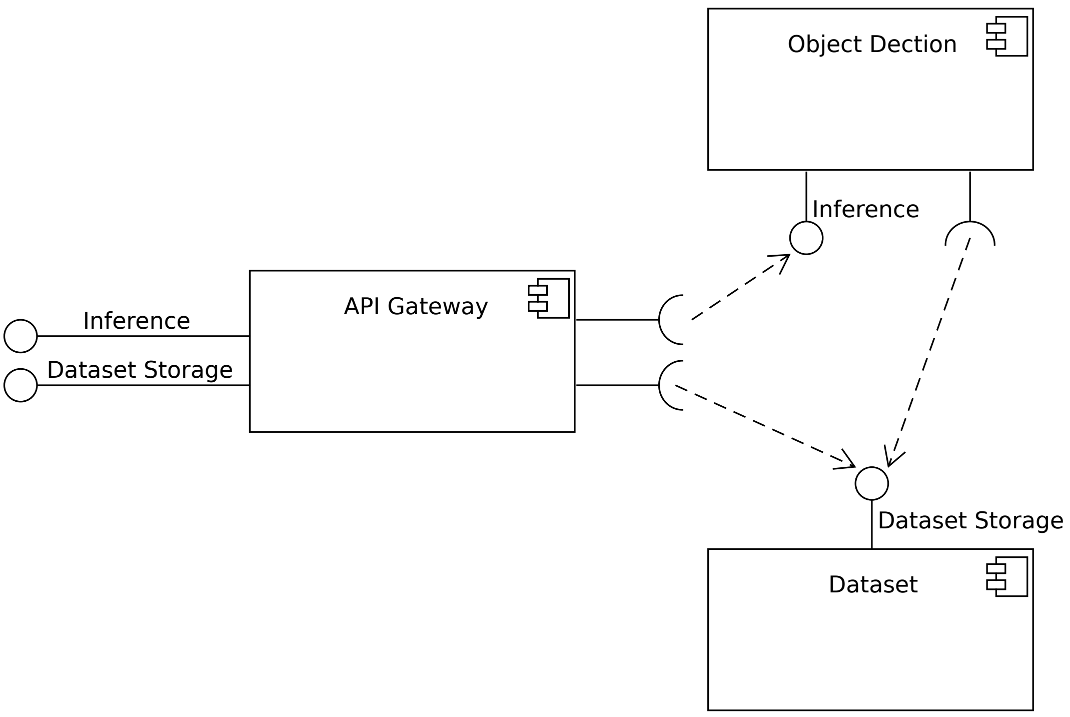


Figure 4.7. Object Detection - Component Diagram

their applicability is feasible without any restriction in this thesis domain, and it is just needed to configure them to detect the desired object types and train them with a proper dataset.

The requirement “RA2. Automation of Object Detection” is met by both RetinaNet and Faster-RCNN because as it can be appreciated in Figure 4.8 and Figure 4.9 respectively, their whole detection processes are automated. They take an image as input and return a set of objects contained in the image with their respective locations. Being an image the input of the process, the requirement “RA3. Visual Input” is also accomplished by both of them.

The requirement “RA4. Deep Learning” is met by RetinaNet since it counts with a “deep” architecture comprised by the ResNet [40] and an FPN [56] that form the backbone of for feature extraction with the additional subnetworks in charge of classifying and detecting objects. Faster-RCNN meets the requirement in a similar way, because of its “deep” architecture comprised by three internal neural networks. The feature extraction network capable of generating feature maps given an image, followed by the RPN that makes region proposals having the feature map as input

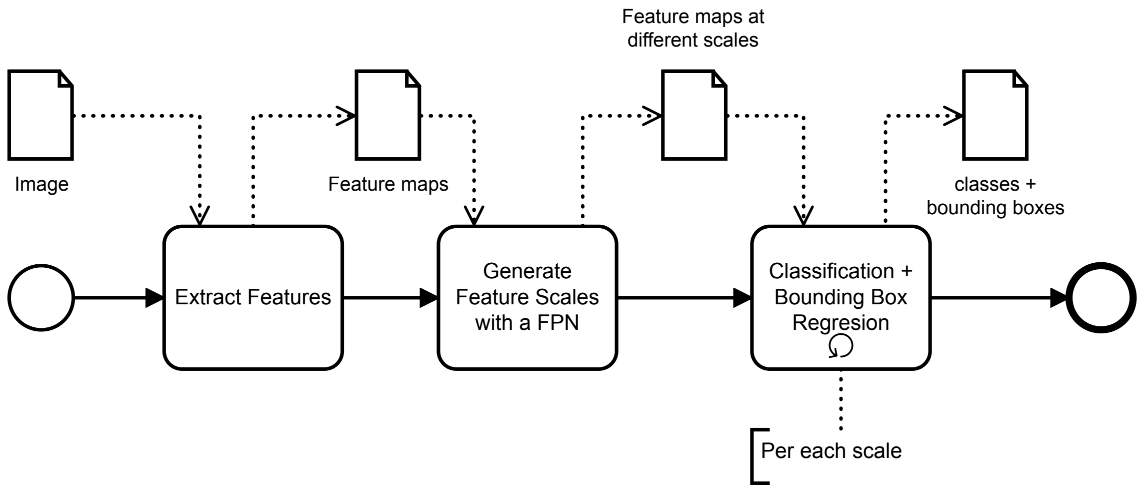

Feature maps at different scales
Image
Feature maps
classes + bounding boxes
Extract Features
Generate Feature Scales with a FPN
Classification + Bounding Box Regresion
Per each scale

Figure 4.8. RetinaNet - BPMN Diagram

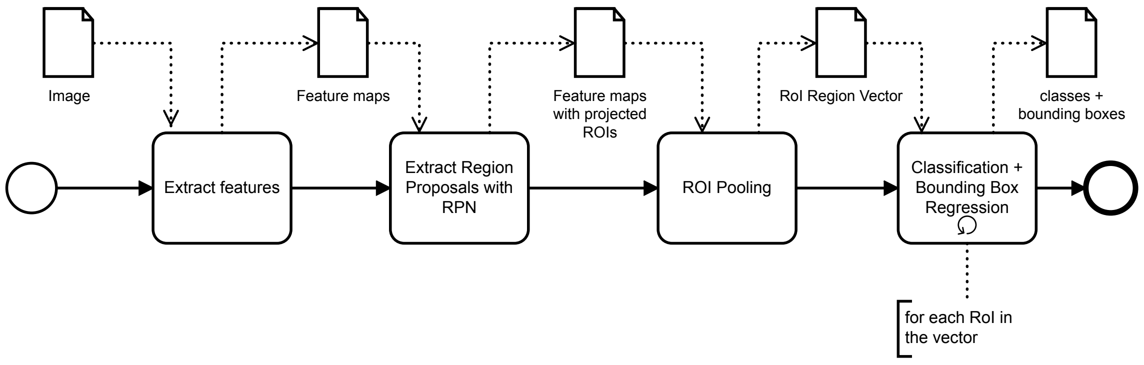


Image
Feature maps
Feature maps with projected ROIs
RoI Region Vector
classes + bounding boxes
Extract features
Extract Region Proposals with RPN
ROI Pooling
Classification + Bounding Box Regression
for each RoI in the vector

Figure 4.9. Faster-RCNN - BPMN Diagram

and the detection network which is just the reuse of its predecessor approach, i.e., Fast-RCNN [31].

For addressing the requirements “RA9. High Precision” and “RA10. High Recall”, RetinaNet and Faster-RCNN are good candidates because they are considered the most accurate object detectors in their respective categories. Nevertheless, this needs further testing since their final performance is highly dependent on the dataset and the hyper-parameters used for training.

Dataset Collection

This section of the architectural draft addresses all the concept details related to an **automated** data collection approach.

One of the key design objectives for this approach is having a system that can be gradually extended to be able to collect and label data in an automated fashion from different sources built by different means. For such reason, an architecture that offers certain flexibility and provides extension mechanisms is proposed in the next paragraphs.

In the automated data collection system, there is a WUI Analyzer that can be seen as the core orchestrator of the whole data collection and labeling process. The WUI Analyzer supports the registration of different WUI Parsers to its parsing pipeline. It supports as well the registration of different Writers that are the ones in charge of storing the labeling in different formats, e.g., the PascalVOC eXtensible Markup Language (XML) format. Besides, it offers the possibility to set a URL Registry that allows the manipulation of a URL repository. Furthermore, the WUI Analyzer has a conflict resolution mechanism for merging the outputs of different parsers.

The goal of each parser is then to find and locate UI objects that follow specific structures imposed by the specificities the frameworks they support. For such purpose, the parsers have access to the DOM of the WUI. The system is intended to provide some implementations, but the developers can also integrate their parsers by just implementing the defined parsers interface and registering them into the analyzer. Having this parsing structure is one of the key aspects for fulfilling “RA5. Sufficient Training Dataset Size” because having more parsers that specialize in certain types of UI, makes the supported WUIs spectrum bigger, and as a consequence, having more labeled data. Additionally, it also contributes to “RA7. Training Dataset Collection Efficiency” because the parsers do not need any human effort to generate the ground truths.

The writers, on the other hand, enable the transformation of the ground truths to any output format they desire. In a similar fashion as with the parsers, the solution offers some default writer implementations for well-known dataset formats. It also supports the extension of new writers, according to the developer needs. Having writers that automate the transformation of the ground truths into different formats contribute as well to the fulfillment of the requirement “RA7. Training Dataset Collection Efficiency”. Also, having a particular writer that specializes in maintaining a record that stores the current dataset size and frequency of distribution across object types contributes to the fulfillment of “RA6. Sufficient Training Dataset Frequency of Distribution”.

It is intended that the solution provides a default URL registry that is replaceable when needed. The registry allows the manipulation of a repository containing URLs to be analyzed. Such configuration makes possible the distribution of the system by having a central repository, and different analyzer instances running on separate machines. The different classes that compose the core architecture of the system are illustrated in Figure 4.10.

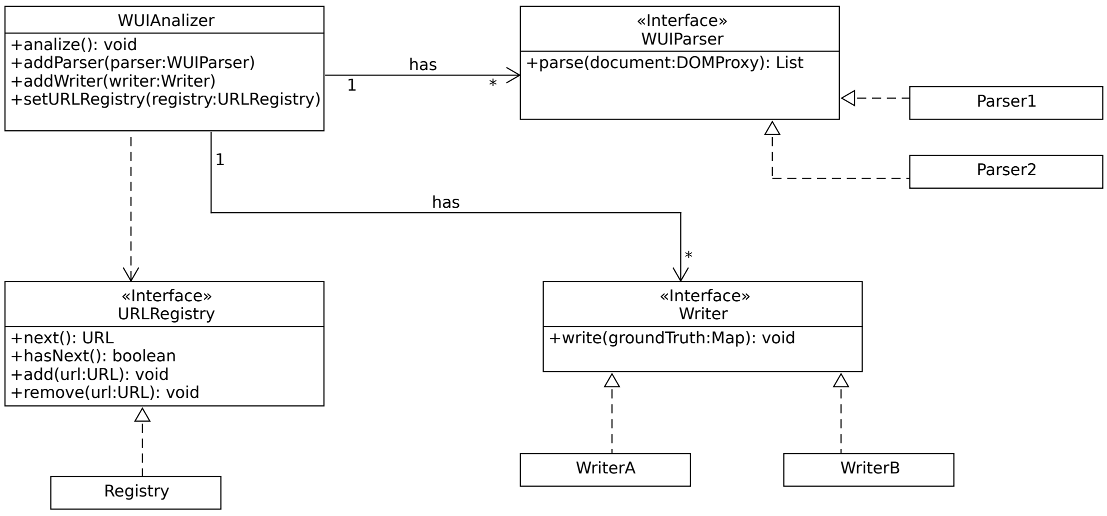


Figure 4.10. Dataset Collector - Class Diagram

In a single iteration, the flow goes the following way: The analyzer gets a new URL to analyze from the URL registry. When the URL is retrieved, the analyzer performs a HTTP GET request to it, and receives a HTTP response containing the corresponding HTML (all the resources associated to the initial request are also considered to be retrieved at this point). It then starts the parsing pipeline processing of the response, by providing to all of the registered Parsers, a DOM representation of the interface. At this point, each of the parsers performs its parsing according to its implementation, i.e., validating if there is something parseable, and then building a structure containing all the UI objects with their respective locations. When the parsing pipeline completes, the analyzer then checks for conflicts in the resulting ground truths, e.g., overlapping of similar UI objects included by different parsers and solves them. Finally, the analyzer takes a screenshot of the WUI and passes the ground truths with additional metadata to the registered writers so that they can adapt the shape of the dataset samples according to their particular implementations. A general overview of the entire interaction between the parts is illustrated in the sequence diagram described in Figure 4.11.

The data collection approach mentioned can work well on its own, but it introduces an additional challenge which is the collection of feasible URLs for further parsing. Since it is desired to enhance the whole data collection mechanism, a WUI generator that creates different WUIs given some randomness factor is proposed. The randomness or *seed* factor contributes deciding at runtime the number of elements per

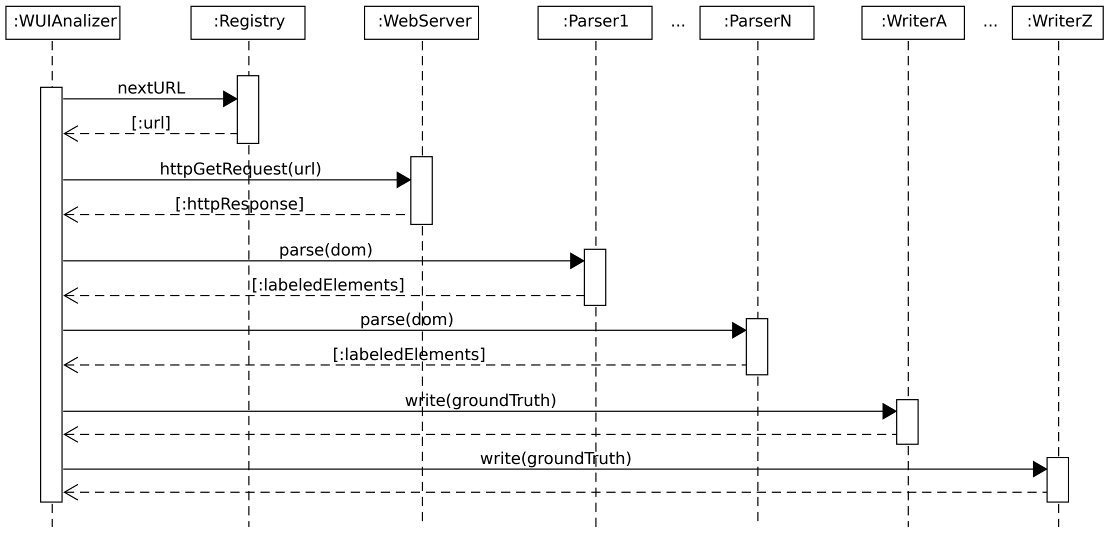


Figure 4.11. Dataset Collector - Sequence Diagram

class and their respective distribution across the view. To support flexibility on the generation mechanisms some default generators are provided and can be extended as well following a similar approach taken from the parsers of the dataset collector.

It is desired that the WUI generator is used through standard means, e.g., through HTTP. Thus it can be easily integrated into the data collection flow. For such reason, a **Web server** that listens to requests and responds with randomly generated user interfaces is needed.

Besides the web server, a **Generation Manager** component decides what generator implementation, and what layout configuration of the elements is applied to the generated WUI.

Each of the implemented **Generators** builds WUIs following the layout configuration specificities that are given by the generation manager.

The description of the flow for a single WUI generation goes as follows: First, a client makes a WUI generation request to the webserver. The server receives the request and parses the incoming "randomness" parameter if present. After the parameters are parsed, it asks the Generation Manager for a new WUI to be generated given the randomness factor. The Generation Manager then builds an abstract WUI structure and selects a random generator implementation to delegate the generation of the WUI given the computed structure. The Generator implementation is in charge of generating the corresponding HTML with a specific look and feel given the structure

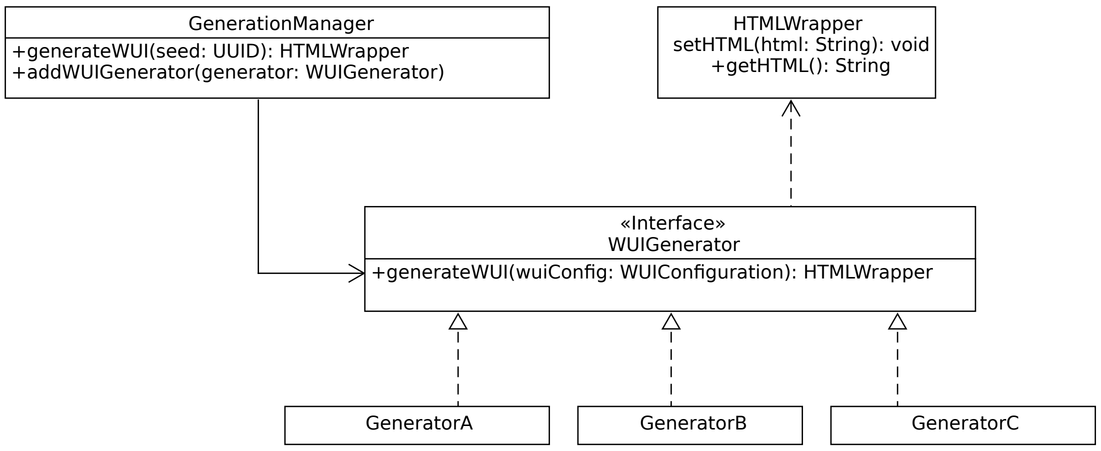

GenerationManager
+generateWUI(seed: UUID): HTMLWrapper
+addWUIGenerator(generator: WUIGenerator)
HTMLWrapper
setHTML(html: String): void
+getHTML(): String
«Interface»
WUIGenerator
+generateWUI(wuiConfig: WUIConfiguration): HTMLWrapper
GeneratorA
GeneratorB
GeneratorC

Figure 4.12. WUI Generator - Class Diagram

of the user interface, and return it once the generation process is completed. Once the Generation Manager receives the generated HTML it forwards it to the webserver which also forwards it back to the client. A sequence diagram showing the interaction between the system objects is illustrated in Figure 4.13.

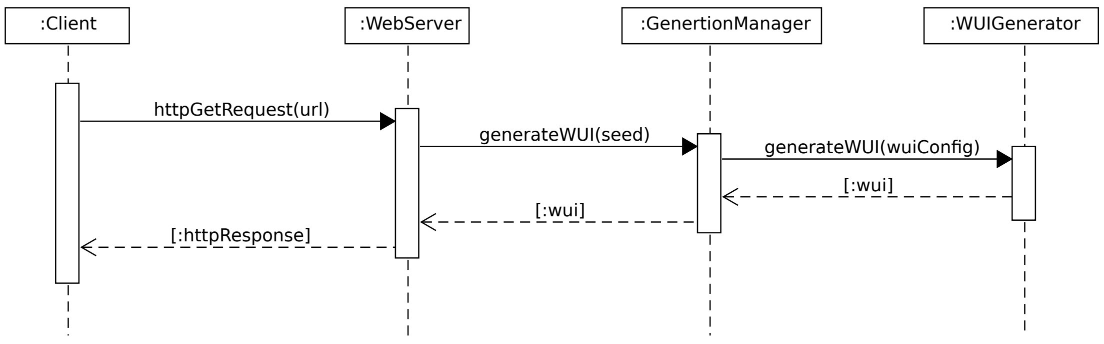


Figure 4.13. WUI Generator - Sequence Diagram

4.3. Combining UI Object Detection Analyzers

Part written by Victor Flamenco

Conclusions in section 3.2 lead to identifying that there is currently no group of approaches that fully solves the problem that this thesis solution part states, and consequently, the concept of an ideal integration platform is to be designed using a combination of these approaches, taking the ideas of each group that fit the best to each requirement. subsection 4.3.1 identifies and describes an overview of potential solution candidates. Subsequently, subsection 4.3.2, submits each alternative to a discussion aiming at choosing the most suitable. The fittest solution is abstractly drafted in subsection 4.3.3.

4.3.1. Possible Solutions

Towards elaborating a concept to combine UI object detectors for an improved detection quality compared to individual detector results, this subsection introduces **three possible solutions**. Solutions presented here are abstract, theoretical concepts which have not yet been implemented up to the full extent detailed in this section. The alternative platform designs include their main components, responsibilities, and interactions. The following list summarizes the alternative platform designs.

- Service-Based Architecture with Meta-Services
- Agents-Based Architecture with (Micro)Service Agents
- Workflow-Based Architecture with (Micro)Services

The three architectural patterns described in subsection 3.2.1 (Workflow-, Agent-, and Service-oriented) are the high-level designs used to identify the three candidate solutions.

In Service-oriented approaches, services collaborate to provide an end set of functions to consumers [71]. Agent-based architectures revolve around the concept of agents, computer systems located in an environment, with encapsulated behavior, that act in the environment flexibly and autonomously [50]. Workflow-oriented platforms support the implementation of diverse distributed execution environments where interrelated tasks are executed and supervised [2]. WFMSs coordinate workflows, computer-readable representations that specify all aspects needed for process execution.

Service- and Workflow-oriented approaches are applicable to the integration platform requirement RB1. Both patterns enable configurable integration of distributed software tools. Agent-based approaches, on the other hand, can integrate tools with agent-organization development efforts. Thus, this is the first indication of three possible high-level alternatives to overcome the requirement. Although agent-based designs provide mechanisms to join distributed software tools, this group obtains its full potential when by implementing reactive agents, since object detection providers should be reusable. When using cognitive agents, the system lacks agents configurability, and this alternative increases system complexity.

Service- and Agent-oriented approaches provide infrastructures for the arrangement of distributed software tools. One possibility with the former group is using *meta-services*, services that encapsulate logic and rules acting upon services [88]. Microservices are also compliant, as one can implement the API gateway pattern. Some object detection tools perform more than one object detection subtask and, thus, one cannot theoretically implement them as microservices.

Agent-oriented approaches present two alternatives: using (1) cognitive agents that can proactively perform a detection action whenever they detect it is necessary; this needs additional programming logic in agents, but is not useful for this work due to the assumption that third-party providers independently create such tools, meaning that tools cannot be modified; and (2) reactive agents that listen passively to environmental changes that trigger their actions; this requires then the environment to handle agents arrangement, increasing development complexity. In contrast, Workflow-oriented approaches use process definitions and process instances, designed to handle the arrangement of distributed tools at both definition and runtime. The most suitable choice is, hence, a workflow-based idea.

By using several tools to execute similar tasks, subsequent results aggregation is desired. An ideal concept should present mechanisms to unify the results of different software components and the ability to output a merged result. None of the researched groups in the previous chapter enables such unification by design; nevertheless, all groups can help partially. In consequence, three potential solutions towards achieving RB3 are extracted, one per SotA group. These alternatives gather the ideas of Meta-services, Agent Environments, and Runtime Control Engines from these groups for constructing a unification controller.

A critical part of results aggregation is conflict resolution (RB4). Possible solutions include rule- or algorithm-based decision-making mechanisms. The three groups researched in the previous chapter provide partial solutions to this requirement. One can implement a service, a reactive agent, or an external workflow task for this purpose. In the end, these three alternatives are in principle equally useful for this requirement, as the goal here is to have one independent software component that deals with conflict resolution and a more prominent orchestration component can consume that. For implementation simplicity, the three upcoming concept alternatives propose a dedicated microservice for conflict resolution.

Ensuring a suitable concept involves evaluating the results of combinations of software components. None of the architectural designs exposed in the previous chapter provides an out-of-the-box solution to do this; thus solving this requirement is worth more attention. The exposed groups may provide partial help, e.g.: (1) a meta-service that generates multiple combinations of services and executes them to evaluate joined results; (2) embedding logic in an agents environment which triggers the execution of multiple agents sequentially towards a single output; and (3) embedding logic in a runtime control for generating multiple process definitions and their subsequent instantiation and evaluation. The upcoming potential architecture solutions consider these three alternatives.

Tools interoperability is achieved by defining uniform communication representation formats for each type of object detection software component, and adequate messaging protocols. All three candidate groups of approaches fully satisfy this requirement. Implementing service descriptions is simpler and more practical using a Web Services Description Language (WSDL) or a REST-based representation, compared to the more complex implementation of FIPA languages. The fact that service-based interoperability aspects are simpler to implement enhances this concept, as they are sufficiently capable of describing services communication.

Based on the above argumentation, the most relevant considerations for extracting alternatives to build a suitable concept are that one:

1. can utilize any high-level architecture studied in the SotA analysis, since the global design only describes the system skeleton, and the three groups may work,
2. has to adopt the concept of process definitions to describe the object detection reference process because they are strictly the only standardized and out-of-the-box solution to RB2,
3. can use any of the three tools-interoperability approaches for system components communication,
4. shall opt for a specialized and independent service -or microservice- to resolve detection conflicts for complying with RB4, and
5. must still explore different possibilities for solving the aspects required by RB3 (results aggregation) and RB5 (tool evaluation) in the upcoming alternative architectures.

Service-Based Architecture with Meta-Services

One can use a high-level architectural pattern following a service-based design, implementing services and a service registry component, and besides, utilize *process* *definitions* as the main artifact type to describe workflows based on a UI object detection reference process. The infrastructure shown in Figure 4.14 considers *two components*, one for building process definitions from user input, and one component to hold detection results.

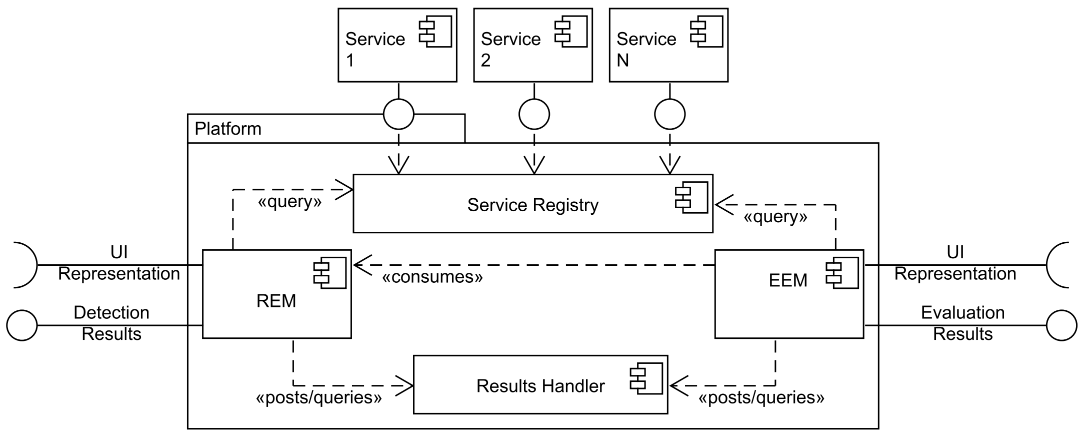


Figure 4.14. Service-Based Architecture with Meta-Services

In addition to these components, one can think of *two key meta-services* that utilize process definitions and play the role of service consumers within the global system architecture. The following list summarizes these aspects.

- the Runtime Engine Metaservice (REM) is in charge of taking user- or system-defined object detection process definitions and finding appropriate services recorded in the registry for each object detection task through queries, and providing an infrastructure to instantiate, run and monitor their execution, to lastly throw a final and unified object detection result,
- a sub-component of the REM that enables process monitoring, or a separate platform component for this purpose,
- the Evaluation Engine Metaservice (EEM) that generates multiple combinations of object-detection services and transforms them into process instances that evaluate joined results; the REM handles this execution,
- a results-holding component, where both the REM and EEM post the detection results; it acts as a bag of results, and
- a component to process user object-detection inputs.

Being the core process execution component, the REM needs to be *fed with appropriate inputs* either processed by another component, or one can think of a sub-component of the REM in charge of this input processing. Furthermore, an essential aspect in this solution is a specific microservice for conflict resolution, which inputs a series of results from several tools, and processes inconsistencies to make decisions towards choosing the most accurate object detections. This microservice is to be introduced to the registry and consumed upon demand by the REM. Communication between services, for simplicity, shall be implemented using, for example, a REST service-description language.

Agents-Based Architecture with (Micro)Service Agents

The architectural design in Figure 4.15 implements an Agent-oriented pattern, consisting of object-detection-tool agents implemented as (micro)services with dedicated functionality, and an environment, an infrastructure where agents reside and interact with each other and with the common environment and share detection results and artifacts. This design includes both the concepts of microservices and; an object-detection tool doing one single subtask is considered a microservice, according to [71]; while tools doing two or more detection sub-tasks are considered regular services.

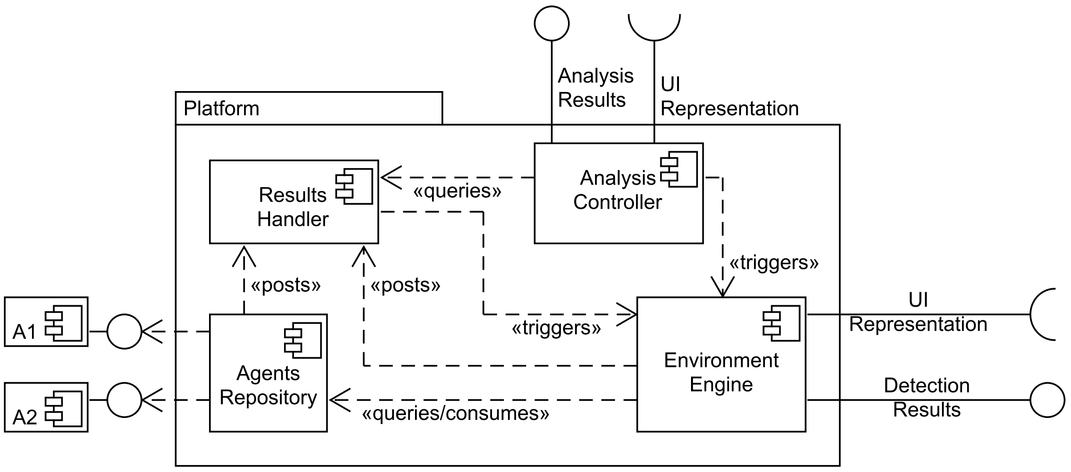


Figure 4.15. Agents-Based Architecture with (Micro)Service Agents

One possible way to design a concept with these characteristics is having *reactive agents*, and *event-based environmental mechanics*. The environment contacts agents with a specific role; it selects one of the available agents by querying an agents repository, requests actions from the selected one, and waits for a results-posting event. These events trigger the environment to continue the detection flow by choosing the following agent. Similarly to the previous solution alternative, this architecture also utilizes process definition artifacts to describe workflows based on a UI object detection reference process. The environment is the most critical component in this architecture, as all agents and detection results reside within, and it has four key components:

- a configurable role-based agents repository, in charge of registering available object detection tools, and facilitating agent discovery,
- an agents-arrangement engine to orchestrate the execution of multiple agents sequentially/parallelly based on process definitions by triggering agent events. It unifies results, and delivers system outputs,
- similar to the previous alternative, a sub-component of the arrangement engine that enables process monitoring, or a separate platform component for this purpose,
- an evaluation engine with embedded logic that generates combinations of agents and transforms them into process instances that are later fed to the arrangement engine in order to evaluate joined results, and
- a component for building process definitions from user input.

The environment needs some logic to process UI object detection inputs, or one can think of an external component in charge of this input processing. In a similar way to the Service-Based Architecture with Meta-Services, this design also requires some specialized service for conflict resolution, with the same characteristics described in the previous alternative; the environment consumes this service on demand. Lastly, communication between agents shall be adopted using the already-existing standardized alternatives for messaging (interfaces and protocols) in the context of agent-oriented platforms, described in section 3.2.1.

Workflow-Based Architecture with (Micro)Services

An almost purely workflow-based architecture is the third alternative (see Figure 4.16). This design takes the main components of a platform oriented to workflows in combination with microservices for solving dedicated tasks. Naturally, it uses process definitions and process instances; it executes instances in an infrastructure provided by a Runtime Engine. This engine controls processes instantiation and monitoring out-of-the-box, it feeds the flow of tasks with appropriate object detection inputs, and at the end, it generates a single output. This design shall consider these critical aspects:

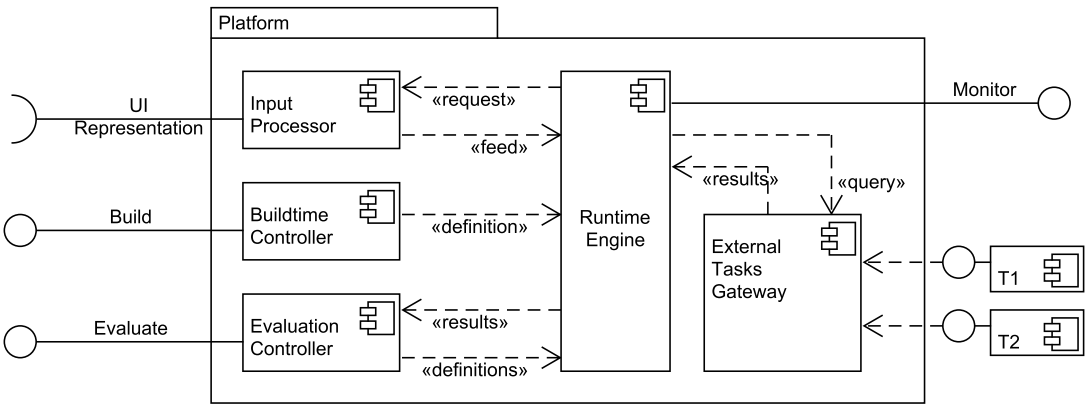


Figure 4.16. Workflow-Based Architecture with (Micro)Services

- a Build-time controller to generate process definitions based on user input,
- a Runtime Engine with embedded logic for running user- and system-generated process definitions; this engine shall include a species of results container where the execution and evaluation components post results,
- a processing component for UI object detection inputs,
- a tool evaluation component with logic for automatically generating multiple combinations of object detection sub-tasks, and the subsequent feeding of such combinations to a build-time controller that turns them into process definitions, to finally feed these definitions to the runtime engine for execution, and
- an external-tools gateway, providing logic for selection of external software tools based on their capabilities.

This alternative includes both microservices and regular services. A platform with these characteristics shall provide three external interfaces for clients to customize detection workflows, request execution of process definitions and tools evaluation. Moreover, this design considers a dedicated microservice for conflict resolution; it receives a series of outputs of other object detection tools and outputs a conflict-free result. The shared infrastructure must call this microservice on demand. Lastly, communication between services shall also adopt a similar approach to the one in the previous alternative solutions.

4.3.2. Discourse

Comparing the three potential solutions is beneficial to maximize the suitability of the concept while simultaneously taking advantage of the drawn conclusions after the SotA analysis pertinent to this second thesis solution part, in subsection 3.2.3.

Service-Based Architecture with Meta-Services

The service-based concept using object detection process definitions complies with all the requirements of an ideal integration platform. Both of its key metaservices are accessible to clients via interfaces and provides an infrastructure to execute processes, and to evaluate the performance of tool combinations. An important advantage to this concept is the number of already existing implementations of this architecture, and a clear path to implementing the registry, services, and their communication aspects. Furthermore, the conflict-resolution and the user input-processing services may also be configured at the registry, bringing *configurability* to implementations to resolve detection inconsistencies and process UI object detection input artifacts. *Communication protocols are very well researched* [35, 42, 78], giving this group an advantage.

Components for results containment and process monitoring shall be further analyzed, as there are two possible paths for each component: either one implements them as sub-components within the REM, or as separate blocks, in which case, it is necessary to describe additional communication between the component and the engines. *This already suggests a higher system complexity* to comply with RB6. Agent- and workflow-oriented approaches consider this component implicitly, meaning that the system complexity of the service-based approach is higher compared to the others. Furthermore, *implementing two metaservices in this alternative increases complexity* too, in the sense that the researched literature does not exemplify these orchestration implementations sufficiently, making RB1 and RB2 possible but complex to achieve. The following two approaches do provide more common infrastructures to achieve the intentions of these metaservices. Consequently, this work *discards the service-based alternative* at this point.

Agents-Based Architecture with (Micro)Service Agents

An agent-based concept complies with all requirements of an ideal platform. A shared agents environment to execute object detection process instances with event-based arranging of role-based agents, and the evaluation engine, and external interfaces for consumption, conform a suitable architecture to customize and automatically generate combinations of diverse UI object detection providers. The environment is an out-of-the-box container for shared detection results, there is no need to consider an additional component for this purpose, and therefore, this *reduces the system complexity* compared to the previous alternative, making RB1 and RB2 feasible.

On the other hand, three custom components for conflict resolution, input processing, and monitoring of processes running in the environment, are still to be considered. Possible solutions to overcome this problem are: (1) to embed these functionalities as sub-component in the environment, and (2) to build separate platform blocks for their purposes, in which case, messaging between the environment and the components should also be included in the design. These arguments *increase system complexity* and require designing such mechanisms that agent-oriented solutions do not consider, therefore RB1 needs special attention.

Furthermore, this potential concept requires the detection tool providers to adopt an agent-oriented messaging protocol; these languages not commonly used, unlike service-based messaging protocols; this is a reason to discourage providers from developing detection services in this manner, limiting the variety of tools that the integration platform may utilize, and consequently, blocking the achievement of this thesis' goal of building a platform as generic as possible for UI object detectors. This alternative limits solutions for RB6, and thus, this work *no longer pursues this alternative.*

Workflow-Based Architecture with (Micro)Services

With this approach, the full potential of a workflow runtime engine can be extracted and combined with the configurability of detection tools and conflict-resolution services, and logic for selection of appropriate object detection tools implemented as services, both features enabled by the external tools gateway. A good indication to decide for this alternative is that the engine holds the status of tasks, and therefore, there is no need to design an additional results container, in contrast to the situation in the previous alternatives; this facilitates RB1. Likewise, the engine is an out-of-the-box solution for process monitoring, enabling, therefore, a reduction in system complexity. The tool evaluation component can potentially reside within the platform at the same level than the runtime engine.

Dedicated microservices for conflict resolution can also be registered at the external tools gateway, providing configurability for resolution of inconsistencies, similar to the previous alternatives, and enabling the achievement of RB2 and RB6. The runtime engine can request this service by asking the gateway. Considering all these factors, this alternative shows *the least complicated architecture*, and avoids re-thinking how to implement some aspects that are already very well addressed in workflow-oriented platforms, specially due to the fact that UI object detection is, in the end, a workflow, and the architectural design of these kinds of platforms is optimized for that purpose. Moreover, combining the idea of external services, and their communication mechanisms is also beneficial. This solution is *the most suitable* to achieve the thesis' objectives, and thus, *this work chooses this alternative for further development* and presents an architectural draft of it next.

4.3.3. Draft

This subsection exposes an architectural draft of the **Thulium Platform**; it first introduces the high-level aspects and a description of the platform's architecture. Next, it proposes the Thulium Object Detection Reference Process, and lastly, it presents a design of the platform architectural components.

High-level Architecture

The core idea behind this concept is a workflow-oriented architecture based on the *Workflow Reference Model.* This framework enables building generic and robust workflow applications and reusing diverse distributed applications to work together, and this is a crucial benefit of workflow platforms. The flow of tasks allows delegating work to different vendors and tools executed in entirely separate environments [43]. These type of systems are called *composite workflow applications.* The Thulium Platform falls within this category.

The specific purpose of this concept is to serve as a workflow platform for UI object detection workflows and with limited types of tasks, and thus, this architecture extracts the characteristics of the reference model that are suitable and useful to achieve this work's objectives. The Workflow Reference Model describes three functional areas: *buildtime*, *runtime control*, and *runtime interactions.* Of these three aspects, the first two are likely to be centralized [2]. One can take the ideas of these three blocks in the following manner:

Buildtime covers workflows definition and modeling, i.e, a process is translated into a computer-readable *process definition* [43]. The definition involves a series of discrete tasks assigned to either human or software entities and rules for tasks execution. For the interests of this work, tasks are only assignable to object detection software tools. This subsection proposes a process definition model suitable for UI object detection considering parser- and vision-based implementations.

During *Runtime Control*, the platform interprets a process definition and creates and executes new process instances, where task coordination and completion happen, by invoking the necessary people and software entities. The Thulium Platform can only invoke software entities. Amid process instance execution in the runtime control, *workflow control data* is persisted to allow execution auditing and recovery from failures, and a *WFM Engine* processes workflow navigation. Alonso and Agrawal propose two components within runtime control: the *storage server* -referred to as WFM Engine in the reference model- and the *navigation server* -known as the workflow control data- [2].

Runtime Interactions enable communication with users and software. User interactions consist basically of a worklist assigned per user; this is commonly concretized through a user interface, e.g., filling a form. Invoked applications, on the other hand, are interfaces that enable execution of remote applications as part of a task [2], e.g., inserting new records in a database, sending a notification. Object detection subtasks can only be delegated to remote software service providers; this concept discards user tasks. Consequently, the Thulium Platform shall only describe the mechanics of working with invoked applications, which are from this point named *Object Detection (OD) Services*. One possibility to overcome this requirement is to define a system component acting as a connector between external object detection providers and the runtime control.

An extension of this framework is still necessary to meet the requirements of this thesis. In addition, to the three building blocks of a workflow-oriented platform following the workflow reference model, the Thulium Platform shall consider two more building blocks: (a) a component for processing suitable UI object detection inputs for parser- and vision-based algorithms, and (b) a tool evaluation component with logic for automatically generating multiple combinations of object detection subtasks, and the subsequent feeding of such combinations to the build-time controller, that turns them into process definitions. This controller can feed these definitions to the runtime engine for execution. A possible design for this component may include collecting the results of executions of all combinations and then computing metrics over these results. Metrics can be analyzed and used as arguments to adjust weights in subsequent object detection processes.

This draft implements an almost purely workflow-based architecture in combination with the ideas of microservices and services for solving dedicated object detection sub-tasks or series of sub-tasks, respectively. Moreover, one can think of a dedicated microservice for conflict resolution: a *CR Service*. This component receives a series of outputs of other microservice tools and outputs a conflict-free object detection result; the shared system infrastructure must consume this service on demand. Lastly, since a (micro)service approach is adopted, communication between services, for simplicity, shall be implemented using, e.g., a REST services description language that includes interfaces and protocols. An overview of the *five building blocks* and (micro)service tools of the Thulium platform are depicted in the diagram of the components in Figure 4.17.

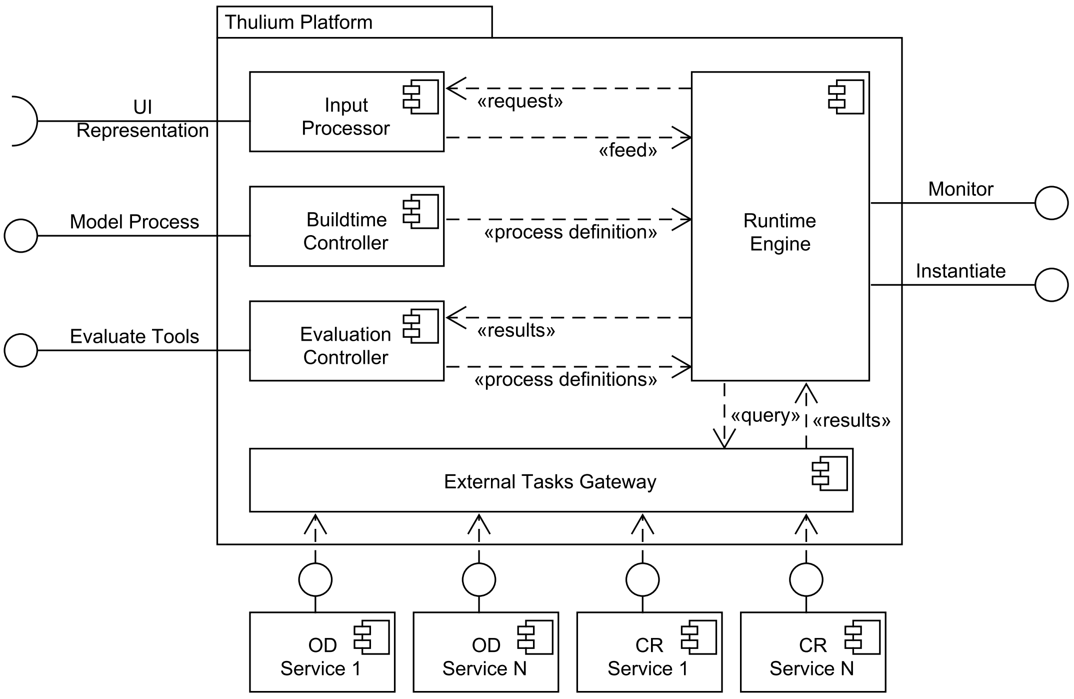


Figure 4.17. Architectural Draft - Combining UI Object Detection Analyzers

This approach extracts the full potential of a Workflow Runtime Engine in combination with configurability of detection tools and conflict-resolution services, and logic for selection of appropriate object detection tools implemented as services, both features enabled by the External Tools Gateway. The runtime engine holds the status of tasks, and therefore, there is no need to design an additional results container. Likewise, the engine is an out-of-the-box solution for process monitoring, enabling, therefore, a reduction in system complexity. The tool evaluation component resides within the platform at the same level as the runtime engine. Custom dedicated microservices for conflict resolution can be registered at the external tools gateway, providing configurability for resolution of inconsistencies in detection outputs. The runtime engine can request registered services by asking the gateway.

This draft implements aspects that are already very well addressed in workflow-oriented platforms, mainly because UI object detection is a workflow, and the architectural design of these types of platforms is optimized for that purpose. Moreover, by combining the idea of external tools and their standardized communication mechanisms, all the process-related requirements in this path of work can be entirely achieved.

Combining parser- and vision-based approaches introduces one crucial step, which is *defining a reference process description* that can fit a wide variety of these techniques and their detection sub-tasks in the most generic possible way.

Thulium UI Object Detection Reference Process

This subsection identifies all conceivable object detection sub-tasks and intermediate artifacts for automated object detection approaches taken from several example implementations and utilizes them as a basis to construct a generic set of object detection steps that can be applied to describe the majority of implementations. The considered **Object Detection Techniques** and their **Strategies** are summarized in Figure 4.18.

Object Detection Techniques	
Parser-based	**Vision-based**
1. End-to-end Strategy	1. End-to-end Strategy
2. Top-to-bottom Strategy	2. Artificial Intelligence Strategy
3. Bottom-to-top Strategy	3. Bottom-to-top Strategy

Figure 4.18. Thulium UI Object Detection Reference Process - Techniques and Strategies

Techniques categorize approaches as parser- or vision-based. Strategies describe the possible series of sequential object detection sub-tasks necessary to detect UI objects. The following subsections identify the subtasks of parser- and vision-based separately, and then finds a suitable combination.

Parser-based techniques

Parser-based techniques divide a page into visually and semantically coherent segments called blocks [85] using UI's underlying structure. Based on their complexity, blocks can be of two types: *atomic* and *composite*. Atomic blocks are indivisible page elements; one cannot divide them into smaller functional and semantically coherent sub-elements. Composite blocks, on the other hand, are divisible into either atomic or smaller composite blocks with visual and semantic coherence. Page segmentation is considered an *object detection* mechanism in the sense that it identifies composite blocks; it recognizes UI blocks to a higher level aiming at semantic elements, while object detection mainly focuses on the atomic blocks, i.e., syntactic elements, although it is not limited to identifying only these as it can also detect composite ones as well.

Specifically for web applications, the input of a parser-based UI object detection process is the DOM tree. Atomic blocks generally refer to HTML tags but are not limited to them, e.g. buttons, links, and input text boxes; while composite blocks are groupings of atomic blocks, such as navigation bars, which may contain a series of links or buttons; or pagination controls, which may include a few buttons and text.

Next, the subtasks of a few *parser-based implementations* are analyzed.

The **Graph-Theoretic approach** for page segmentation uses the structure and properties of the DOM tree, as well as the relationship between its nodes [15]. The approach formulates an optimization problem on weighted graphs, where the weights tell the cost of placing the connected end-points in same/different page segments. It defines a segmentation function that receives a graph node and assigns it to a page segment. This function depends on two cost functions: (a) D_p, the cost of assigning a particular label to the node p, and (b) V_{pq}, the cost of assigning two different labels to the edge end-points (p, q). Three steps summarize the algorithm.

1. Convert the DOM tree of N elements into a graph with a node-set N.
2. Define the segmentation function for the graph.
3. Find the graph weights that minimize the segmentation function. They propose a learning algorithm trained with available labeled data.

The **Bento Page Segmentation algorithm** proposed by Kumar et al. generates a new cleaner and more consistent version of the input DOM tree for a better page segmentation using a four-step algorithm [53] described in the following list. Once the segmentation is complete, all page content is assigned to a leaf node in the DOM tree, and every non-leaf node contains its children nodes in screen space.

1. Wrap inlined elements <*span*> tags.
2. Reshuffle hierarchy to match parent-child relationships in the tree with visual containment.
3. Remove redundant and superfluous nodes.

4. Supplement hierarchy with missing visual structures by computing horizontal and vertical separators and inserting enclosing DOM nodes.

Block-o-Matic uses an improved version of the Bento algorithm in combination with heuristic rules defined by the W3C Web Standards in order to produce a labeled segmentation [87]. BoM performs segmentation without knowledge of the site's content. The algorithm uses the heuristic rules to detect blocks using HTML5 content categories instead of using the tag names. Segmentation is a two-phase process performed at any desired granularity, based on a stop condition:

1. Detect atomic objects.
2. Merge blocks until a stop condition is satisfied.

The **VIPS algorithm** is an effective top-down approach for extracting the semantic structure of UI [13]. It is a *hybrid* approach since it utilizes the DOM tree of a web page, as well as its visual cues. The algorithm extracts the semantic structure of a website, i.e., a hierarchical structure of nodes, each corresponding a block and containing a degree value to describe how coherent the block's content is concerning the visual perception. The algorithm follows these steps:

1. Segment the page by recursively extracting all relevant blocks from the DOM tree.
2. Find optimal division lines that visually separate blocks.
3. Describe the semantic structure based on the found divisions.

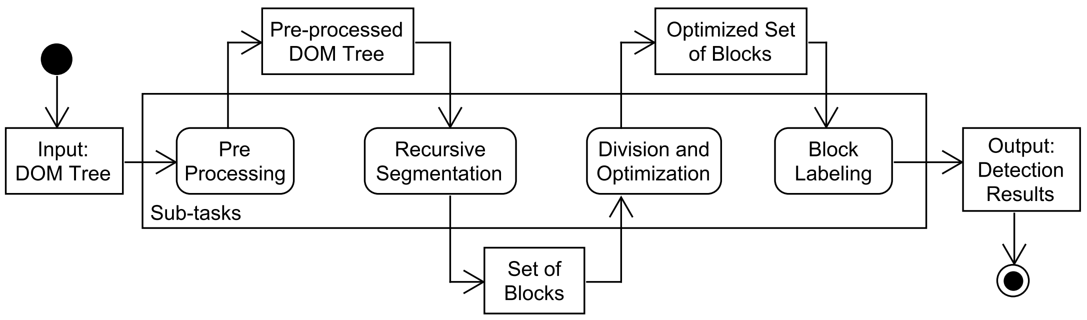


Figure 4.19. Top-To-Bottom Parser-Based Strategy

These example implementations lead to the identification of two important aspects. First, parser-based implementations use the original DOM tree and output a cleaner and more consistent version of the tree, which indicates a good site segmentation. Second, the following three parser-based strategies depict a categorization for these:

- End-to-end strategy,
- Top-to-bottom strategy, and
- Bottom-to-top strategy.

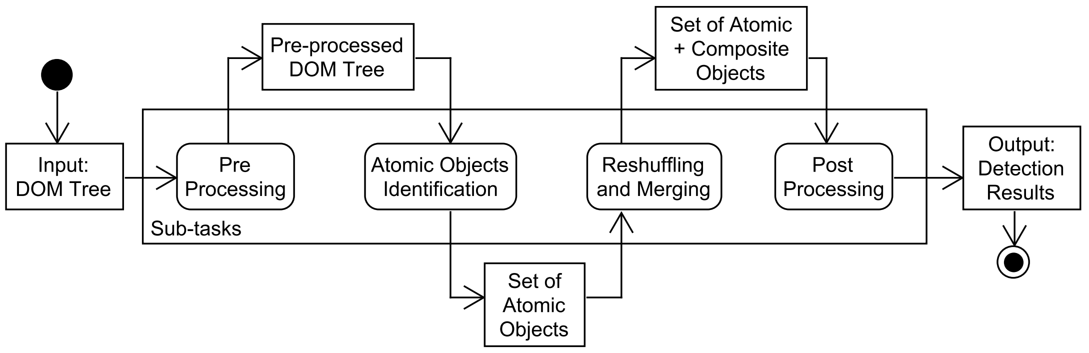


Figure 4.20. Bottom-To-Top Parser-Based Strategy

The **end-to-end strategy** includes implementations that are highly customized and that use intermediate steps and artifacts (e.g., graphs instead of trees) that are inapplicable to any other implementation, and consequently, it is quite challenging to generalize such algorithms. Moreover, this category also fits those indivisible implementations that execute a complete segmentation on their own.

The **top-to-bottom strategy** (Figure 4.19) segments the page by recursively extracting all relevant blocks from the DOM tree, to subsequently find and optimize horizontal and vertical separation lines between blocks, and lastly, it labels each block to describe the semantic structure based on the found divisions.

The **bottom-to-top strategy** progressively finds more complex objects starting from the deepest tree nodes and moving upwards. This strategy, depicted as a four-step process with different intermediate artifacts in Figure 4.20, firstly pre-processes the DOM tree in order to prepare and clean the tree for the following steps. This step may include, as an example, wrapping elements with consistent tags. Secondly, the identification of atomic objects takes place in order to describe their coordinates and sizes. The third step is the core of this strategy and consists of rearranging the positions of objects to fix/improve parent-child relationships, and subsequently merging atomic objects to form composite structures; this is an *iterative step* that scales upwards until finding the root node. The fourth and last step is a post-processing phase that cleans unnecessary tags from the tree.

Constructing a generic parser-based reference process must consider both of these strategies. Later in this section, a merged design considering these strategies in combination with vision-based strategies is presented. The following subsection reviews the former strategies.

Vision-based techniques

Vision-based techniques rely more on a visual representation of the UI, rather than on its underlying structure; this allows computers to understand the semantic structure and detect objects of a UI similar to how a human does. These techniques take advantage of computer vision and image processing in order to segment UIs accurately and detect atomic and composite blocks within them.

The traditional vision-based object detection process consists of detecting instances of semantic classes, i.e., the location and the scale of each one, in an input image, given a predefined set of object classes [92]; these objects can be of any semantic class, e.g., a person, a car, a dog. Moreover, they compute a confidence score that represents how confident the model is that the predicted regions contain an object and how accurate the model thinks the predicted semantic class is [79].

Next, the sub-tasks of a few *vision-based implementations* are analyzed.

Web Interface Interpretation Using Graph Grammars propose a *bottom-to-top* UI segmentation technique based on image analysis and graph grammar, and use the Spatial Graph Grammar (SGG) as the definition formalism for page segmentation [52]. Image segmentation techniques are applied to identify objects in an image of a rendered UI. Object recognition techniques are used to recognize the interface object types. The process consists of two steps:

1. Recognize and classify atomic objects to generate a spatial graph where nodes represent recognized interface objects, and edges indicate significant spatial relations.

2. Apply a graph grammar to the spatial graph to discover the interface semantics and output a tree that represents hierarchical relations among interface objects.

Machine-learning approaches manually identify and extract features for further usage in the learning and prediction phases (*feature engineering*). This strategy identifies interesting rectangular blocks in the image, an then, for each region, it applies a two-step process. First, feature extraction to generate a *feature vector*; and second, image classification. Instances of feature extractors are: (a) the SIFT algorithm, it generates image features invariant to properties such as translation, rotation, scaling, and partially invariant to illumination and three-dimensional projection [62]; and (b) the HOG algorithm, based on the evaluation of well-normalized local histograms of image gradient orientations in a dense grid. For classification tasks, widely-used techniques are, e.g., SVMs and neural networks. Moreover, one can group classifiers according to their outputs (binary and multi-class) classifiers. Three steps summarize the process.

1. Compute region proposals.
2. Feature extraction per region.
3. Classification per region for labeling.

Deep-learning techniques use deep neural networks to perform a three-step end-to-end object detection by progressively extracting and learning complex features and RoIs from images, as well as classifying these regions [60]. This process is indivisible and is, therefore, a one-step strategy. Both *machine-learning* and *deep-learning* techniques can be generically described using *the three same steps.* One must remember, nevertheless, that most region-based CNN implementations perform these steps with one, neural network, meaning that the process is indivisible.

Several incrementally improved versions of **region-based CNNs** have been proposed. *RCNN* was the first one, it integrates AlexNet with a selective-search region proposal algorithm [32]. *SPPNet* combines the traditional SPP with a CNN architecture, obtaining speed improvements while keeping the detection quality [39]. *Fast RCNN* and *faster RCNN* were designed to overcome some of the limitations of RCNN and SPPNet and simultaneously increase the detection quality [31, 83].

These example implementations lead to identifying two essential aspects. First, vision-based approaches generally pre-process images before object detection, e.g., changing the color scales, image sizes, and any other image-processing technique. Second, the following three vision-based strategies depict a categorization for these techniques.

- End-to-end strategy,
- Artificial Intelligence (AI) strategy, and
- Bottom-to-top strategy.

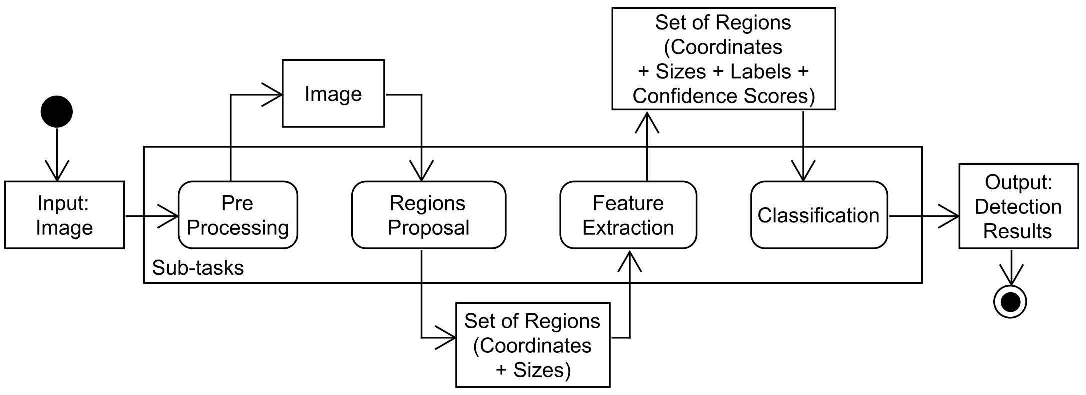

Set of Regions (Coordinates + Sizes + Labels + Confidence Scores)
Image
Input: Image
Pre Processing
Regions Proposal
Feature Extraction
Classification
Output: Detection Results
Sub-tasks
Set of Regions (Coordinates + Sizes)

Figure 4.21. Vision-Based Artificial Intelligence Strategy

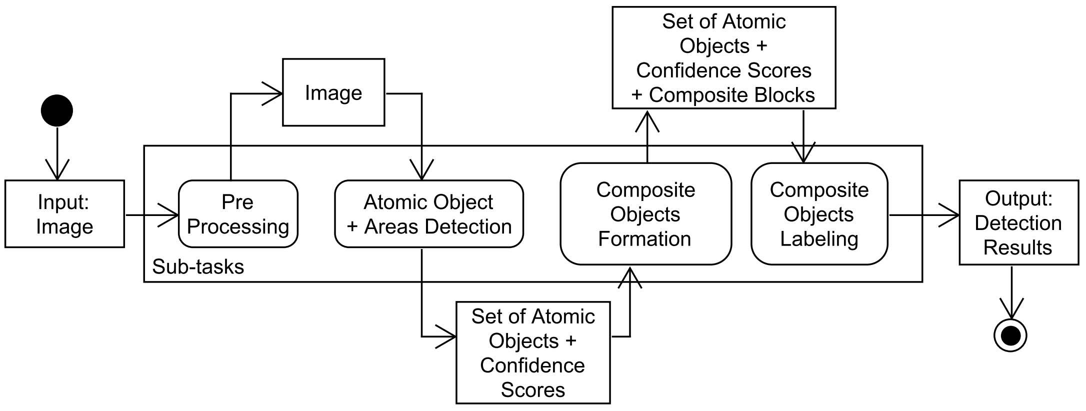

Set of Atomic Objects + Confidence Scores + Composite Blocks
Image
Input: Image
Pre Processing
Atomic Object + Areas Detection
Composite Objects Formation
Composite Objects Labeling
Output: Detection Results
Sub-tasks
Set of Atomic Objects + Confidence Scores

Figure 4.22. Bottom-To-Top Vision-Based Strategy

The **end-to-end strategy** includes those implementations that perform all steps at once and are indivisible. Instances of this group are treated as black boxes, receiving interface images, and outputting object detection results.

The **AI strategy**, shown in Figure 4.21, is formed by processes where image features individually are extracted for each rectangle to determine later the object contained within. Generally, these implementations use artificial intelligence for detection tasks, and thus the strategy name.

The **bottom-to-top strategy**, visually described in Figure 4.22, firstly detects atomic objects and interesting -small- rectangular areas, to later form composite objects, generally using heuristic rules. Subsequently, each composite block is labeled. The output additionally pairs each label with a confidence score.

Constructing a generic vision-based reference process must consider all of these strategies. The upcoming subsection presents a merged design considering these strategies in combination with the parser-based ones.

Combining Parser- and Vision-based Techniques

Having identified all conceivable object detection sub-tasks and intermediate artifacts for automated object detection approaches taken from several example implementations, they are utilized at this point as a basis to identify and build a generic set of object detection sub-tasks that may apply to describe the majority of these techniques.

Interface	Description
IDOMTree	DOM Tree
IScreenshot	Screenshot of rendered UI
IUnlabeledSet	Set (coordinates + size)
ILabeledSet	Set (coordinates + size + label)
ILabeledConfSet	Set (coordinates + size + label + confidence)
IUnlabeledFeatSet	Set (coordinates + size + feature vector)

Table 4.1. Reference Process Interfaces

As a first step, in Table 4.1 all possible input, output and intermediate artifacts for generically describing UI object detection processes are described, namely **Interfaces**. IDOMTree and IScreenshot are the two possible system inputs, while ILabeledConfSet is the unified system output. All interfaces -including these three- can also be used as intermediate artifacts within sub-tasks. Coordinates represent the position, measured in pixels, of a rectangle's top-left corner. Sizes include two measures, also in pixels, for the rectangle's width and height. Confidence scores are percentage values.

Code	Description	Input	Output
TPE1	End-to-End	IDOMTree	ILabeledSet
TPT1	Pre-processing	IDOMTree	IDOMTree
TPT2	Recursive Segmentation	IDOMTree	IDOMTree + IUnlabeledSet
TPT3	Find/Optimize Divisions	IDOMTree + IUnlabeledSet	IDOMTree + IUnlabeledSet
TPT4	Label Objects	IDOMTree + IUnlabeledSet	ILabeledSet
TPB1	Pre-processing	IDOMTree	IDOMTree
TPB2	Identify Atomic Objects	IDOMTree	IDOMTree + ILabeledSet
TPB3	Reshuffle and Merge	IDOMTree + ILabeledSet	IDOMTree + ILabeledSet
TPB4	Post-processing	IDOMTree + ILabeledSet	ILabeledSet
TVE1	End-to-end	IScreenshot	ILabeledConfSet
TVA1	Pre-processing	IScreenshot	IScreenshot
TVA2	Propose Regions	IScreenshot	IScreenshot + IUnlabeledSet
TVA3	Extract Region Features	IScreenshot + IUnlabeledSet	IScreenshot + IUnlabeledFeatSet
TVA4	Classify Regions	IScreenshot + IUnlabeledFeatSet	ILabeledConfSet

TVT1	Pre-processing	IScreenshot	IScreenshot
TVT2	Detect Atomic Objects	IScreenshot	IScreenshot + ILabeledConfSet
TVT3	Form Composite Objects	IScreenshot + ILabeledConfSet	IScreenshot + ILabeledConfSet
TVT4	Label Composite Objects	IScreenshot + ILabeledConfSet	ILabeledConfSet
TXC1	Conflict Resolution	ILabeledConfSet	ILabeledConfSet

Table 4.2. Reference Process - Mapping of Object Detection Sub-Tasks with Input and Output Interfaces

Table 4.2 maps the identified object detection subtasks to their input and output interfaces. A *task code* accompanies each task description. The codes are designed to identify which task it refers to easily, and consist of four characters, where the first one is always a letter *T* meaning that it refers to a **T**ask; the second character points to the technique that the task belongs to, i.e., **P**arser- or **V**ision-based; the third one depicts the *strategy* used by the task, i.e., **E**nd-to-end, **T**op-to-bottom, **B**ottom-to-top, or **A**rtificial Intelligence; and the last character is an integer reflecting the task index, the ordinal position it occupies in sequence of tasks. The only exception is TXC1, as this a conflict-resolution sub-task shared among all techniques.

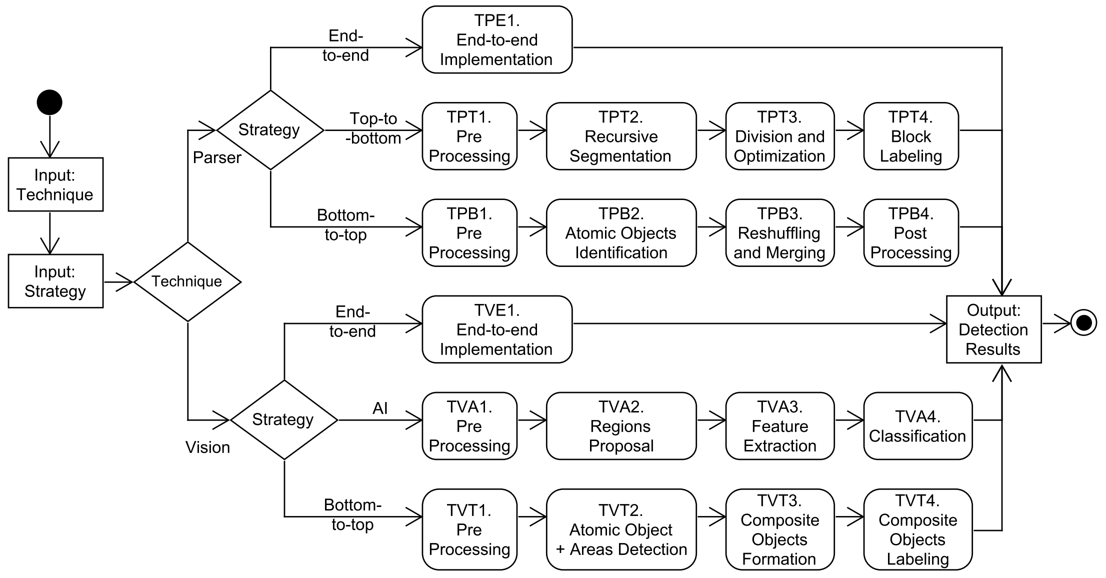

Input: Technique
Input: Strategy
Technique
Parser
Vision
Strategy
End-to-end
Top-to-bottom
Bottom-to-top
AI
TPE1. End-to-end Implementation
TPT1. Pre Processing
TPT2. Recursive Segmentation
TPT3. Division and Optimization
TPT4. Block Labeling
TPB1. Pre Processing
TPB2. Atomic Objects Identification
TPB3. Reshuffling and Merging
TPB4. Post Processing
TVE1. End-to-end Implementation
TVA1. Pre Processing
TVA2. Regions Proposal
TVA3. Feature Extraction
TVA4. Classification
TVT1. Pre Processing
TVT2. Atomic Object + Areas Detection
TVT3. Composite Objects Formation
TVT4. Composite Objects Labeling
Output: Detection Results

Figure 4.23. Thulium UI Object Detection Reference Process

The proposed reference process accepts two input variables in order to choose an appropriate sequence of tasks: the **Object Detection Technique**, and the **Technique's Strategy**. Once the process receives both variables, it directs the execution path towards the set of tasks that match the combination of variables. The diagram in Figure 4.23 describes all evaluation conditions and paths of task sets to generically describe any UI object detection processes, and therefore, it composes the *Thulium*

UI Object Detection Reference Process. This workflow is a key component of the Thulium Platform, as it describes the logic to choose UI object detection sub-tasks for each implementation within a macro-process that combines multiple implementations.

Architecture Components

This subsection describes the following architecture components.

- Input Processor
- External Tools Gateway
- Buildtime Controller
- Runtime Engine
- Conflict Resolution External Tool
- Evaluations Controller

Input Processor

The input-processing component prepares the object detection artifacts to the platform and provides the runtime engine with appropriate inputs for any given object detection implementation. Parser-based techniques use DOMs trees as input, while vision-based ones use screenshots of the target UI. This component includes input-processing logic for both types of techniques.

In order to obtain the DOM tree, the two alternatives are: manually providing the artifact in a suitable file format, or automatically obtaining the tree from the website's URL. On the other hand, for feeding a screenshot of a rendered interface, alternatives are: the component receives an image file, in which case no further input processing is necessary, or the component receives the site's URL and captures the rendered interface.

The proposed workflow in Figure 4.24 generates a suitable object detection input by evaluating two aspects: the type of input artifact, and the technique of object detection in execution by the engine. When the provided input matches the input type required by the engine, no actions follow. Otherwise, it either extracts a DOM tree or generates a screenshot or several screenshots. The screenshot-capturing method expects four input parameters:

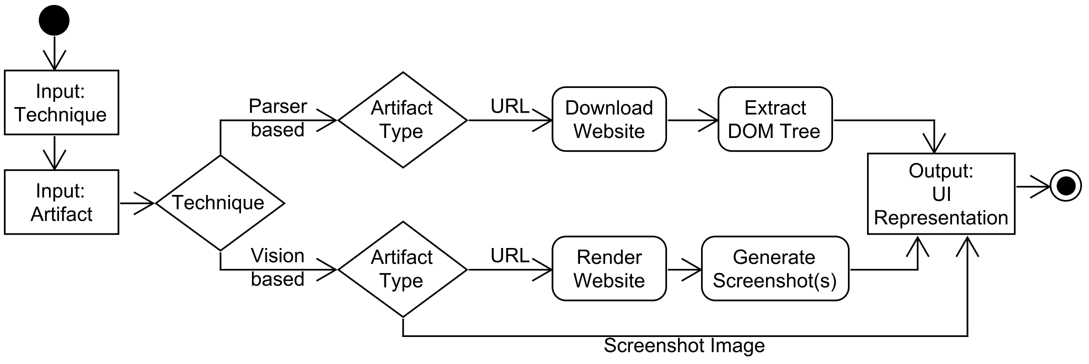


Figure 4.24. Input Processor - Workflow

- Target site's URL,
- Screenshot(s) resolution,
- Vertical scroll step, and
- Maximum number of screenshots to capture.

Screenshot(s) capturing consists of fetching and rendering the site using its URL, and vertically scrolling by the value of the step size until the end of the site is reached, or until the number of taken screenshots equals the maximum number of screenshots parameter, to limit sites with virtually infinite scrolling.

External Tools Gateway

The intention behind *runtime interactions* is to enable communication with users and software [2]. The External Tools Gateway implements the concept of *invoked applications*, interfaces that enable execution of remote applications as part of a task, e.g., inserting new records in a database, sending a notification, or performing an object detection sub-task. Object detection sub-tasks can only be delegated to remote software service providers.

This component describes the mechanics of working with invoked applications, and is designed as a role-based repository of *external services*, from this point, referred to as **External Tools**. The Gateway acts as a connector between the external tool and the platform, specifically, the Runtime Engine and the Evaluations Controller. External tools are categorized as follows:

- **Object Detection Tool**: receives a series of object detection artifacts utilizing the *input interfaces*, executes one or multiple object detection sub-tasks, and returns a series of artifacts using *output interfaces*. Input and output interfaces are described in Table 4.1.
- **Conflict Resolution Tool**: accepts a series of *weighted* detection outputs -a set of artifacts using the *ILabeledConfSet interface*- and returns a single artifact with the same interface. This concept drafts an instance of this type of tools in section 4.3.3.

A **Tool Role** describes the tool capabilities, meaning which object detection sub-task(s) it can perform. A role is defined by the combination of two parameters: *toolType* and *taskCodes*, where *toolType* differentiates *single-task* tools from *multiple-task* tools. Tools of the former type are accompanied by one task code parameter (see Table 4.2), while tools of the latter group have a range of task codes $[T_i - T_j]$, where $i < j$, meaning that task T_i must strictly be before T_j in the sequence of tasks defined at the reference process. An important restriction is that both T_i and T_j must belong to the same *(technique,strategy)* flow.

These fields describe an external tool and its capabilities:

- Tool Name,
- Object Detection Technique,
- Technique's Strategy,
- Initial Sub-Task,
- Final Sub-Task, and
- Uniform Resource Identifier (URI).

The Gateway provides *two interfaces*: (a) tool registration, and (b) tools querying based on their roles. Queries are requested with the parameters: *technique*, *strategy*, *initialStep*, and $finalStep$. The Gateway evaluates whether $initialStep == finalStep$ (a single-task service) or $initialStep \neq finalStep$ (a multiple-task service) shall be returned. The Gateway then searches in the repository for a tool that matches the parameters and returns instructions to consume the service if it finds one.

Buildtime Controller

Build-time covers workflows definition and modeling; it translates a process into a computer-readable *process definition* [43]. The definition covers a series of discrete tasks assigned to either human or software entities and rules for tasks execution. For the interests of this work, tasks are only assignable to object detection software tools.

The build-time controller receives a configuration of object detection implementations provided by a platform user and translates it into a process definition readable to the runtime engine. The controller feeds the Engine with process definitions that the Engine can then instantiate and monitor. Moreover, interactions with the external tools gateway are necessary in order to enable the selection of external tool providers for the execution of specific object detection sub-tasks or series of sub-tasks.

Process definitions have a name, a version number, conditions for start and termination, and additional fields for security and auditing [43]. Workflow versions and security fields are out of the scope of this thesis, while auditing fields, on the other hand, are crucial as they enable process instance monitoring. Nevertheless, auditing fields are excluded from process definitions and are included in process instances instead (see section 4.3.3). The **Thulium Process Definition Artifact** is a simplified version of this fieldset, seasoned with object-detection-specific fields, listed next.

1. Definition name
2. Set of implementations
    a) Object detection technique
    b) Technique's strategy
    c) Weight
    d) Set of steps
        i. Step index
        ii. Initial object detection sub-task
        iii. Final object detection sub-task
        iv. External tool
    e) Class Labels Filter
3. Conflict Resolution Tool

The artifact includes a series of **Implementations**, each of these is an independent **Object Detection Technique** following one **Strategy** exposed in section 4.3.3. Besides, it includes a series of Steps per implementation, which map object detection sub-tasks with tool providers for execution. As per the Thulium reference process, each technique/strategy combination has a specific number of sub-tasks. Each *Step* item describes which of these sub-tasks are performed by the external tool, specified using the range *[Initial SubTask - Final SubTask]*.

A crucial feature for enabling the improvement of detection results is the ability to select which UI object types (see Table B.1) from the output of an Implementation counts towards composing the final detection output. Ideally, an implementation should be first isolatedly evaluated for identifying which object types it detects with high accuracy. The *Class Labels Filter* field contains a set of object types that the Runtime Engine must extract from the implementation's output in order to compose the final detection output.

Among the activities of the build-time controller is the validation of coherence of the step sub-tasks ranges. Each implementation has a weight value, representing the percentage of importance that it is assigned when aggregating results. The last process definition field -*Conflict Resolution Tool*- references an external tool in charge of solving output inconsistencies across implementations and returning a unified detection result.

Runtime Engine

During *runtime control*, workflow-oriented platforms interpret a process definition, and create and execute new process instances, where tasks coordination and completion happen, by invoking the appropriate external tools. Amid process instance execution, workflow control data is persisted to allow execution auditing and recovery from failures, and a WFM engine processes workflow navigation.

The runtime engine takes two inputs: process definitions and a test dataset consisting of DOM trees, interface images, or both types simultaneously. **Four steps** describe the engine's functionality.

1. Interpretation
2. Execution and Monitoring
3. Output Generation
4. Output Evaluation

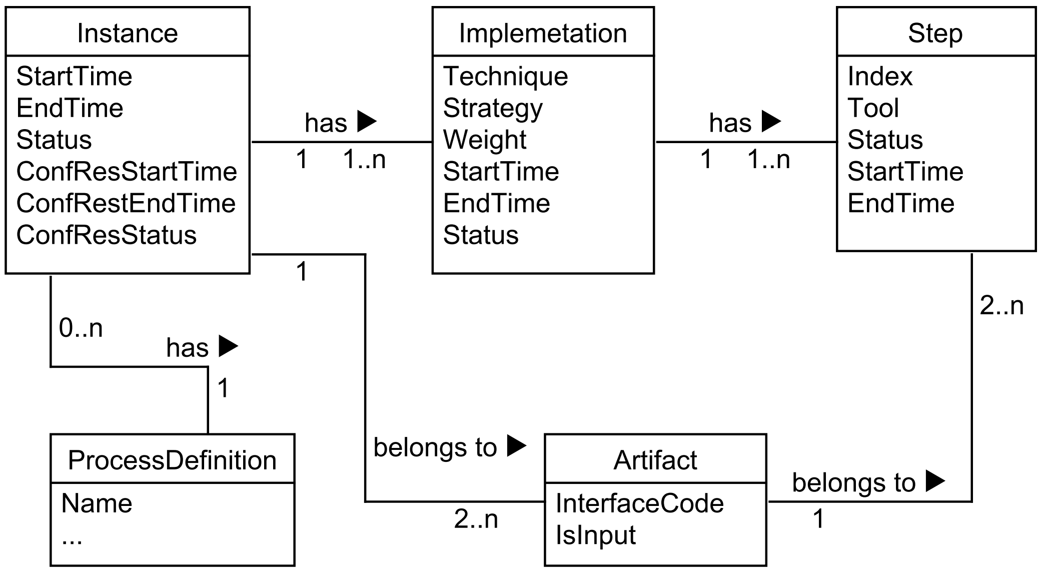


Figure 4.25. Runtime Engine - Entities and Relationships

1. Interpretation: the runtime engine prepares the object detection process execution by creating an instance from a definition, and persisting the series of object detection implementations that the definition contains, and per implementation, persisting the sequence of steps to execute.

The engine initializes each implementation and each of its steps with a *NotStarted* status; it stores a reference to the external tools responsible for steps execution. It additionally stores the inputs to feed the initial steps. The entity-relationship diagram in Figure 4.25 serves as a model for this.

2. Execution and monitoring: the engine starts the instance and its implementations in parallel and changes their statuses to *Executing*; it feeds and triggers the initial step of each implementation with inputs that depend on the implementation's technique. Triggering a step involves querying the Gateway for the external tool in charge of the step execution, and subsequently, requesting the tool execution.

Communication between the engine and external tools is achieved using Webhooks, an "elegant, clean, and RESTful solution for bi-directional Web eventing" [91], based on the publish/subscribe messaging pattern [25]. Webhooks enable data reception from applications through callbacks over HTTP. Upon events, data is sent to the callback URLs specified at subscription time.

The engine feeds the external tool with an appropriate payload, i.e., possible input interfaces for the step's initial sub-task, described in section 4.3.3. Step start- and end-time are persisted, and the step status is changed to *Executing*. The external tool then notifies the engine once step execution is complete. The response payload is persisted, and the engine populates the audit fields. If the task is completed successfully, the engine pipelines the payload to the next step in the sequence. Otherwise, the engine allows manually triggering a retry action on the failed step. The new possible step statuses are therefore: *Completed* and *Error*.

This iteration happens until the last step in the sequence is completed. At this point, the engine closes the execution in the current implementation, marks it as *Completed*, and filters the set of detected objects from the implementation's output based on the set of class labels to consider, configured when modeling the process definition (see section 4.3.3).

Subsequently, the engine checks whether all implementations in the process instance are complete. If that is the case, the engine handles results aggregation through conflict resolution. Unification consists of taking the outputs of the last step of each implementation, i.e., a series of *weighted* artifacts with the ILabeledConfSet interface, and executing the conflict-resolution external tool with these artifacts as input. Once the tool achieves a resolution, it returns a single ILabeledConfSet artifact containing the conflict-resolution output. The engine persists the unification result at the instance level, and as a final step, it populates the instance's audit fields and marks the instance as *Completed*.

3. Output generation: the instance's output detection annotations and the input UI screenshot pass through an engine's sub-component in charge of graphically annotating the image by drawing a rectangle for each detected object, and labeling each rectangle with the predicted class name and confidence score. The drawings are returned on top of the original interface image.

4. Output evaluation: a critical feature that enables comparing the instance's output detection annotations to a ground truth annotations file. For this, the assistance of the Evaluations Controller (section 4.3.3) is required. The controller computes the **Average Precision (AP)** per object type; it is the mean Precision at a set of eleven equally spaced recall levels, and it corresponds to the evaluation method used in the PASCAL VOC Challenge [28]. Average Precision is the most commonly used metric for measuring the performance of object detection algorithms [60]. This component adopts **Mean Average Precision (mAP)** as the final measure of performance, and computes it by averaging all the APs per object type.

Conflict Resolution External Tool

The Conflict Resolution External Tool is in charge of results unification. It receives a series of *weighted* ILabeledConfSet outputs from diverse object detection implementations, and returns a single ILabeledConfSet artifact. This output is intended to be a clean and ideally conflict-free UI object detection result.

This component must be registered at the External Tools Gateway, providing configurability for custom conflict resolution implementations, and it must be available for consumption upon demand by the runtime engine once it has orchestrated the execution of all instance implementations. This tool implements the communication mechanisms described in section 4.3.3.

Heterogeneous object detectors, especially vision-based ones since they are probabilistic, can predict different results (RoIs, class labels, and confidence scores) for the same input UI representation. Conflict resolution refers to deciding which implementation within a process instance has the most accurate detection output, by examining two **conflict dimensions**: **bounding-box** and **classification**.

The bounding-box dimension studies overlapping detections with matching class labels. Two detections are considered conflictive in this dimension if they are labeled with the same object class, and the Intersect Over Union value[2] (Figure 4.26) $IOU(bBox_1, bBox_2) \geq \varepsilon$, where $bBox_1$ and $bBox_2$ are the bounding boxes of the two detections, and ε is a threshold value. The classification conflicts dimension, on the other hand, studies cases where algorithms predict different semantic classes for matching bounding boxes, predictions are marked as conflictive in this dimension when they are labeled with different object classes, and the ratio $IOU(bBox_1, bBox_2) \geq \varepsilon$.

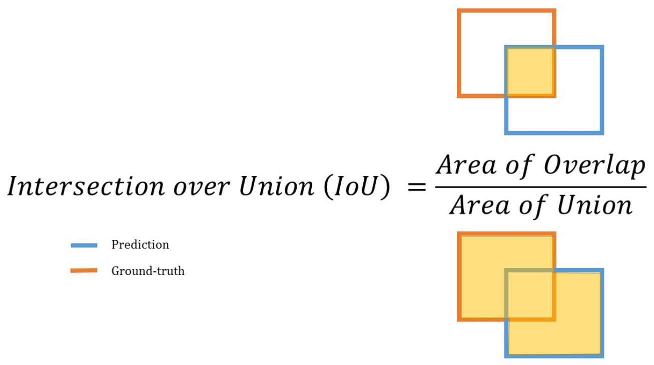

Intersection over Union (IoU) = Area of Overlap / Area of Union
Prediction
Ground-truth

Figure 4.26. Intersection Over Union

[2]https://www.kaggle.com/c/3d-object-detection-for-autonomous-vehicles/overview/evaluation

```
function thuliumSoftNMS(detections, iouBoundingBox, iouClassification
    , confidenceThreshold, sigma)
     for (detection in detections)
         detection.confidence = detection.confidence * detection.
    weight
     groupsPerObjectType = groupByObjectType(detections)
     subSets = []
     for (group in groupsPerObjectType)
         subSet = softNMS(group, iouBoundingBox)
         subSets.add(subSet)
     return softNMS(subSets, iouClassification)
```

Listing 4.1 Pseudocode - Thulium Soft-NMS Algorithm

The *NMS algorithm* is an integral part of the object detection pipeline, and it aims at removing overlapping detections by recursively finding detections with significant overlaps, using a pre-defined threshold and suppressing the ones with lower confidence score [8]. Bodla et al. propose *Soft-NMS*, which, instead of completely discarding the overlapping detections with lower confidence, penalizes their confidences by applying a Gaussian function. In their work, they show that this soft version obtains consistent improvements over several commonly used datasets, such as PASCAL VOC and MS-COCO. Consequently, the Conflict Resolution component is based on this extension of NMS.

Soft-NMS uses three hyperparameters: (a) *Intersection over Union (IoU) threshold*, the minimum overlap ratio between two bounding boxes that are to be marked as conflictive, (b) *confidence threshold*, the minimum confidence score value for suppressing detections, and (3) σ, the confidence-penalization parameter. The **Thulium Soft-NMS algorithm** (Listing 4.1) extends these hyperparameters:

- Bounding-Box IoU Threshold
- Classification IoU Threshold
- Confidence Penalization
- Confidence Threshold

The algorithm penalizes detection confidences in two ways. First, using the confidence penalization parameter in the Gaussian softening function; and second, using the *implementation weight* defined in the process definition. This algorithm resolves first the bounding-box dimension conflicts by grouping all the detections by object type, and per group, applying Soft-NMS with a **Bounding-Box IoU Threshold** value. Subsequently, the resulting subsets are joined, and the classification-dimension conflicts are resolved by applying Soft-NMS over the entire set, with a **Classification IoU Threshold** to suppress conflicts of detections predicting different object types. Having one hyperparameter per conflict dimension makes the algorithm more dynamic for its intended use.

Evaluations Controller

The Evaluations Controller facilitates automatedly testing the detection quality of isolated external tools, as well as combinations of tools, against configurable datasets. With this, improving object detection quality becomes an iterative process designed with the following use case in mind.

First, one can set up a testing dataset, consisting of several UI screenshots, and one ground truth labels file per image. Second, one can model a process definition where one external tool is isolated, or several external tools are combined. Next, the process definition is instantiated with each dataset element as input, and the detection results are used for computing object detection metrics. One can analyze these metrics and adjust the definition parameters -implementation weights and class labels filter-, and conflict-resolution parameters (see section 4.3.3). After parameters adjustment, evaluation metrics can be recomputed. This loop happens until the detection results are sufficiently good.

Although the Build-time Controller allows customizing definitions with isolated or combined and weighted implementations of detectors, this is not scalable since the number of possible combinations of tools can grow exponentially as the number of registered external tools increases. Testing all combinations involves manually modeling a high number of process definitions. For this reason, the Evaluations Controller also assists in automating the modeling of process definitions. Its **four main tasks** are therefore:

1. Generation of combinations of tools,
2. Management of a repository of datasets,
3. Evaluation of process definitions against entire testing datasets, and
4. Evaluation of individual process instance outputs.

The **first task** of the Evaluations Controller is to generate multiple combinations of object detection flows based on the Thulium reference process, and considering detection tools registered and available at the External Tasks Gateway, that are capable of performing one or several sub-tasks.

Towards generating all possible combinations of object detectors (see Listing 4.2), the controller has one function that takes: (a) the set of available external tools, and (b) the Thulium reference process. The function iterates over all defined pairs of *(technique, strategy)* in the reference process, and for each pair, the possible sub-tasks are extracted and provided as input to a helper function that generates all possible series of sub-tasks. Given an example strategy with the sequential tasks: $[T1, T2, T3, T4]$, the possible combinations of sub-tasks are:

- $[T1, T2, T3, T4]$
- $[[T1 - T2], T3, T4]$
- $[[T1 - T2], [T3 - T4]]$
- $[[T1 - T3], T4]$
- $[[T1 - T4]]$
- $[T1, [T2 - T3], T4]$
- $[T1, [T2 - T4]]$
- $[T1, T2, [T3 - T4]]$

The process filters the series to leave only the ones for which there exists a registered external tool for each of its composing items. As an example, for the series $[[T1 - T2], T3, T4]$, the algorithm queries the Gateway for three tools: one that simultaneously executes both tasks 1 and 2, i.e., $[T1 - T2]$, one for executing $T3$, and one for $T4$. If the three tools are found, the series is added to the set of possible combinations, the function's output. Moreover, if multiple suitable tools are found for each item, then each alternative represents a new combination of tools, which are therefore added to the function's output as well.

```
function generatePossibleCombinations(refProcess, availableTools)
    combinations = []
    objectDetectionTools = availableTools.getObjectDetectionTools()
    conflictResolutionTools = availableTools.
   getConflictResolutionTools()
    foreach (technique in refProcess.techniques)
        foreach (strategy in technique.strategies)
            subTasks = getSubTasks(refProcess, technique, strategy)
            seriesOfSubTasks = generateAllPossibleSeries(subTasks)
            foreach (series in seriesOfSubTasks)
                seriesIsPossible = true
                stepTools = []
                foreach (step in series)
```

```
                    toolsForStep = toolsForStep(objectDetectionTools, step)
                    **if** (toolsForStep is empty)
                        seriesIsPossible = false
                    else
                        **foreach** (tool **in** toolsForStep)
                            stepTools.add({ step, tool })
                **if** (seriesIsPossible)
                    **foreach** (confResTool **in** conflictResolutionTools)
                        combinations.Add({series, stepTools, confResTool})
    **return** combinations
```

Listing 4.2 Pseudocode - Evaluations Controller - Generate Tools Combinations

Once all possible combinations are formed, the Evaluations Controller clones each combination n times, where n represents the number of available conflict resolution tools. This final set of combinations is provided to the Build-time Controller that turns them into process definitions.

The **second task** of this controller consists of managing a repository of testing datasets. It enables persisting pairs of input UI representations with their corresponding ground truth annotation files. Datasets are then replicable whenever an evaluation is triggered in the next task.

When running evaluations (the controller's **third task**), one process definition and one testing dataset are taken as inputs. The controller calls the Runtime Engine to instantiate the definition one time per dataset item and collects the results from all instances once completed.

The **fourth task** consists of measuring the quality of detection results for the entire the dataset, by comparing the detection outputs of the process instances to the dataset's ground truth annotation files. Three criteria can be used to evaluate the performance of detection algorithms: detection speed, precision, and recall [60]. This thesis discards the detection speed, utilizing, therefore, only the two remaining criteria: (precision and recall). Based on the formula presented in section 2.1, obtaining these values requires calculating the number of True Positives (TPs) and False Positives (FPs).

A predicted detection is marked as TP if:

- the predicted class label is equal to the ground-truth label,

- the IoU between the predicted bounding box and the ground truth bounding box is greater or equal than an IoU threshold ε.

Otherwise, the detection is flagged as FP.

AP per object class can be derived from the TP and FP. Moreover, **mAP** is adopted as the final measure of performance. Evaluation results are presented ordered by the quality of their detections. With this data, one can infer valuable information about the most suitable combinations of sub-tasks that they may use to model and parameterize subsequent object detection processes. Such parameterization includes adjusting the definition's weights and class labels filters, and the conflict-resolution parameters.

As a final feature, to facilitate iterative improvements of detection quality, this controller allows recomputing the evaluation metrics when any parameters have been modified, without re-executing the external tools against the same dataset, a task that can be both resource- and time-consuming.

4.4. Summary

This chapter defines a conceptual description of a solution capable of combining different UI object detection strategies in order to improve overall detection results. At start, it carries out the description of a top-level architecture outlining an overview of a *unified* solution which addresses all the thesis objectives (see *Objective A* and *Objective B* in section 1.4).

The concept considers potential solution candidates and their respective advantages and disadvantages; it includes an abstract design of the most suitable candidates. For *Objective A* it encompassed two object detectors able to learn progressively as well as an automated dataset collection approach. For *Objective B*, it consists of a workflow-based platform that allows the integration of different object detection strategies and offers means for conflict resolution and aggregation of results.

The next chapter goes further and describes implementation considerations concerning the solutions drafted here; it discusses the most challenging aspects from an implementation point of view of the solution and explains the different paths to take for addressing them.

5. Implementation

The previous chapter proposes a conceptual solution capable of combining different UI object detection strategies in order to improve overall detection results. The current chapter describes the implementation considerations aligned with such concept; it discusses the most challenging aspects from an implementation point of view of the solution and explains the different paths exercised to address them.

The achievement of thesis Objective A requires implementing a DNN-based UI object detector able to learn progressively; it requires the usage of object detectors that define deep architectures using CNNs that support receiving visual representations of UIs and predicting their contained UI objects in an automated way. Additionally, it requires the acquisition of a sufficient dataset suitable for training the object detector.

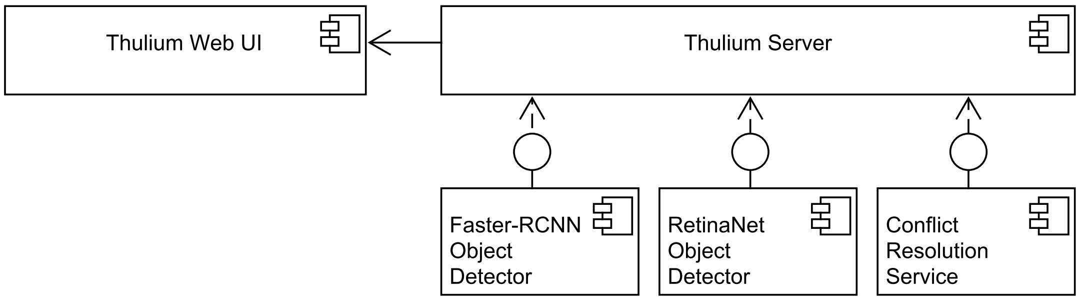

Thulium Web UI
Thulium Server
Faster-RCNN Object Detector
RetinaNet Object Detector
Conflict Resolution Service

Figure 5.1. Implementation of Solution Architecture

Achieving thesis Objective B needs the implementation of a software platform capable of integrating multiple implementations of UI object detection techniques. Requirements for this include: the design of an integration platform architecture guaranteeing support for multiple UI object detection tools, handling the results aggregation and conflict resolution, counting with uniform inter-component communication interfaces. Moreover, requirements describe the necessity to evaluate different combinations of tools for figuring out the most suitable weight distribution mechanism for components.

The implemented solution shown in Figure 5.1 concretizes the overall thesis concept. It includes two vision-based object detectors -Faster-RCNN and RetinaNet implementations-, and a web application for integrating multiple object detection tools. The web application consists of a web UI and an application server. The former component allows users to register external object detection and conflict resolution tools, configure process definitions and execute and monitor process instances, review and download instance artifacts and results, and request evaluation of existing tools.

The following two sections detail the implementation of each of the two parts of this solution. section 5.1 examines the concretization of the DNN-based object detectors, while section 5.2 exposes the fulfillment of the Thulium Platform.

5.1. UI Object Detection Using DNNs

Part written by Juan Sebastian Quintero

This section describes the most challenging implementation parts of a prototype that follows the design aspects sketched in subsection 4.2.3. It mentions implementation considerations derived from an object detection system, as well as gathering a sufficient dataset to train it properly. For each of them, the most challenging parts are described in terms of technologies used, internal architectures, components, and algorithms.

Object Detection

This subsection shows some of the most challenging aspects that are taken into account in the prototype implementation of the object detection mechanism. At first, an architectural overview of the prototype is described. Following the architectural description, key functionalities considered at the training and inference phases are further discussed. For the prototype implementation, the *Python*[1] language is selected given the variety of tools, libraries, and frameworks that are available on top of it in the field of deep learning and object detection. All the code snippets and listings in this subsection are shown with the python syntax for consistency reasons.

Architecture

The architecture of the implemented prototype follows an API gateway pattern. Such architectural decision allows having different services with different hardware require-

[1]https://www.python.org/

ments deployed on machines with different specifications. Additionally, it allows injecting extra logic before/after services execution and encapsulates the location of the services to the clients. The API gateway is implemented by using a *NGINX*[2] server, and it exposes the following services:

- A service that offers an inference operation for the detection of objects in UI images and also encapsulates all training related functionalities i.e., the **Object Detection Service**. It requires high computational requirements given the cost of the operations related to the training and inference processes.
- A service that supports storage operations of training data and internally allows fetching this training data on demand by the object detection service i.e., the **Dataset Service**.

Two different object detectors (RetinaNet and Faster-RCNN) are considered for this prototype implementation. Such detection models are implemented in this occasion using the *Tensorflow Object Detection API*[3], since it provides out-of-the-box models that follow both of these meta-architectures. This tool also counts with a *Model Zoo* which has pre-trained models on common datasets such as COCO, Open Images, among others and can be used right away for inference or for *Transfer Learning* [90]. It becomes convenient in this case because such pre-trained models can be used as starting points that are fine-tuned afterward for the detection of UI objects, enabling a much cheaper training procedure. More details on the object detection framework can be found in [46].

Over the models, a service allowing the encapsulation of the training and inference processes performed by them is put in place. It is called the **Object Detection Service**. This service has two key components. The *Training Client* and the *Inference Service*. The former is in charge of orchestrating all training related aspects. The later exposes an operation via REST that allows invoking the inference process on unlabeled images of UIs.

Additionally, the **Dataset Service** represents a dataset that can grow in size across time and becomes the primary source for the training client of the object detection service. A deployment diagram with some more details is illustrated better in Figure 5.2. The internals of both training and inference processes are exposed below.

[2]https://www.nginx.com/
[3]https://github.com/tensorflow/models/tree/master/research/object_detection

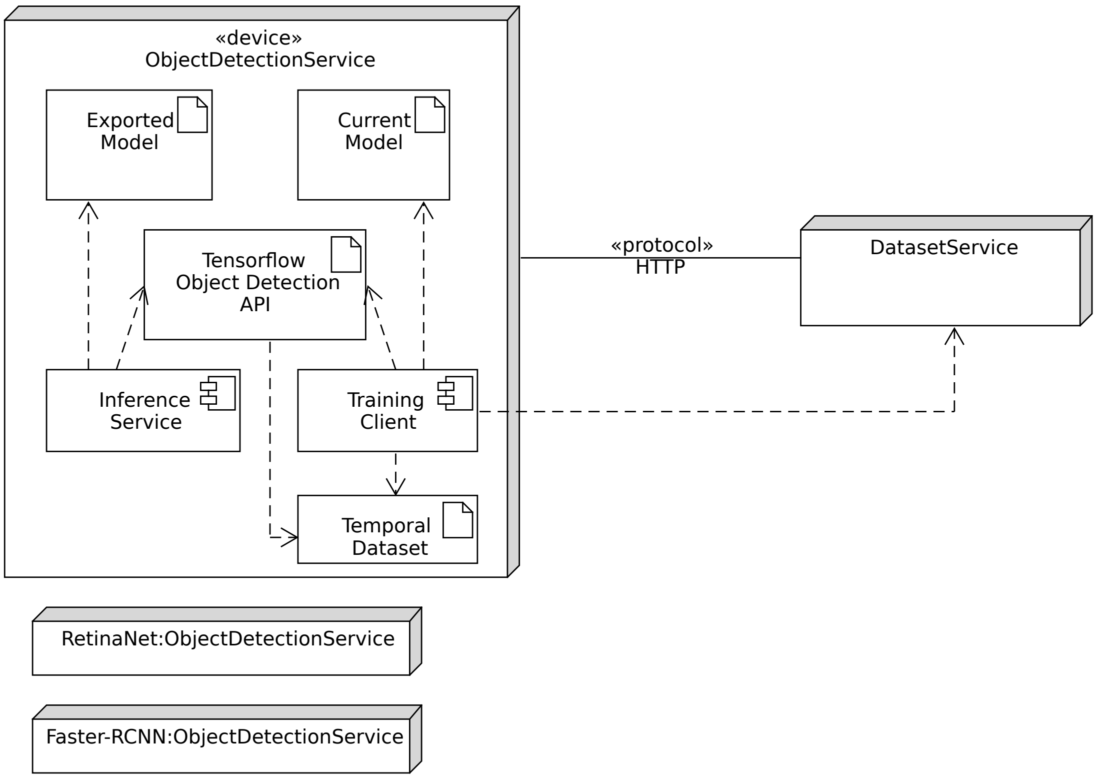


Figure 5.2. Object Detection - Deployment Diagram

Training Process

In the training process, the core parts are the **Training Client** and the **Dataset Service**. The training client is the one that enables a progressive training of the model. Such progressive training is possible thanks to its support for regularly fetching training examples from the dataset service that further conform a *Temporal Dataset*. It then orchestrates all training related processes using this gathered dataset. Such processes include pre-processing the training data, calling external training scripts, and exporting the trained model afterward for inference.

For gathering training examples, the training client fetches new ones from the dataset service by following a *HTTP Long Polling*[4] technique. It is convenient in the sense that long polling allows the training client to control the flow of the requests. The long polling technique enables gathering training examples on-demand, avoiding bottlenecks at the client side if there is a massive flow of new training examples registered

[4]https://tools.ietf.org/html/rfc6202

in the dataset service. Another advantage is that it is easily consumed as a normal REST API operation from the clients becoming transparent for them.

For each received training example from the dataset service, the training client invokes a pre-processing phase. The pre-processing of training examples is important because:

- It provides means for data augmentation, which is always a good alternative when training DNN-based models.
- It overcomes precision and recall degradations that can be introduced by existing models that execute auto-resizing on input images because such models require the images with smaller sizes and different aspect ratios.
- It allows the object detection service to be trained with images of different sizes.

This pre-processing phase goes as follows: Having an input image I of size $W \times H$ with its respective labels L, a sliding window SW of size $W_{sw} \times H_{sw}$ is sliced over the image with a stride of S_{sw} in order to generate N cropped-images of size $W_{sw} \times H_{sw}$. For each of the cropped images, N labels that contain information strictly related to each of them are generated. After the generation of the labels is completed, the ones that do not contain any objects are filtered out. When finishing the pre-processing phase, the model can be trained with N images and labels (cropped ones). The values for W_{sw}, H_{sw} and S are fully customizable according to the underlying model's needs. The implementation of the sliding window method is shown in Listing 5.1.

```
def slide_window(width, height, window_width, window_height, stride_x
    , stride_y):
    frames = []
    y = 0
    y_ended = False
    while not y_ended:
        if y + window_height >= height:
            y -= (y + window_height) - height
            y = max(0, y)
            y_ended = True
        x = 0
        x_ended = False
        while not x_ended:
            if x + window_width >= width:
                x -= (x + window_width) - width
                x = max(0, x)
                x_ended = True
            frames.append({
                'x': x,
                'y': y,
                'width': window_width,
                'height': window_height,
            })
            x += stride_x
        y += stride_y
    return frames
```

Listing 5.1 Code Snippet - Sliding Window Method

After the pre-processing phase is completed, the training client stores the resulting augmented data with the help of a dataset storer. For this prototype implementation, the dataset storer uses a PascalVOC structure to store the temporal dataset that is used for training. Following such structure is ideal because it allows reusing ready-to-use scripts provided by the Tensorflow Object Detection API that operate upon this dataset convention. When the training client detects that the storer counts with a sufficient temporal dataset for training, it stops the fetch process, builds the dataset with the required format (TFRecords) and initializes the training process. In Listing 5.2, the implementation code is shown in detail.

```
    # Initialization of objects is omitted
    while True:
        while not storer.has_enough_data():
            _id, image, annotations = fetcher.fetch()
            to_store = preprocessor.preprocess(image, annotations)
            for idx, item in enumerate(to_store):
                image, annotations, _ = item
                storer.store(_id + "-" + str(idx + 1), image, annotations)
        storer.build_sets()
        tfrecords_gen.build_tfrecords()
        trainer.train()
        exporter.export()
        tfrecords_gen.clean()
        storer.clean()
```

Listing 5.2 Code Snippet - Fetch and Train

Inference Process

In the inference process, The most crucial component is the **Inference Service**. It is the one in charge of executing the detection of objects given unlabeled UI images. The inference process is exposed as a REST API operation, allowing it to be called on-demand. The implementation receives an image from an inference client, pre-processes it, and feeds it to a model that is ready for inference. Such a model has been previously trained and exported by the training client. After receiving the responses from the model, the outputs are post-processed and returned to the inference client.

The inference service also follows a very similar pre-processing stage as the one described in the training process with some slight variations. Having an input image I of size $W \times H$, a sliding window SW of size $W_{sw} \times H_{sw}$ is sliced over the image with a stride of S_{sw} in order to generate N cropped-images of size $W_{sw} \times H_{sw}$. After finishing the pre-processing phase, the inference model is invoked N times. The values for W_{sw}, H_{sw} and S correspond to the same-ones parametrized for the training phase for consistency reasons.

Since one inference call to the inference service results in N calls to the model, it is also needed to have a post-processing stage that *unifies* all the different model outputs for each of the cropped images. For this purpose, The inference service uses the NMS algorithm provided by the *OpenCV*[5] library.

Dataset Collection

This subsection shows the most prominent parts of the implementation of a prototype that follows the design considerations explained in the concept chapter for an automated collection of a dataset. At first, an architectural overview of the prototype is described. Following the architectural overview, key parts of the internal components and tools are emphasized. For the prototype implementation, the *Python*[6] language is selected. All the code snippets and listings in this subsection are shown with the python syntax for consistency reasons.

Architecture

The architecture of the data collection solution follows a *Client-Server* architecture and is comprised of two different applications. On the client-side, there is the **WUI Analyzer** which is in charge of the examination of web user interfaces and producing their respective annotations in an automated fashion. On the server-side, there is a **WUI Generator** that synthetically generates web user interfaces with random look-and-feels and distribution of elements. The analysis process of the WUI Analyzer includes web user interfaces that are synthetically generated on demand by the WUI Generator. Additionally, it also incorporates web pages from different online sources

[5]https://opencv.org/
[6]https://www.python.org/

with the help of a *Web Crawler* [51]. An illustration of the architectural solution, including both applications, is shown in Figure 5.3.

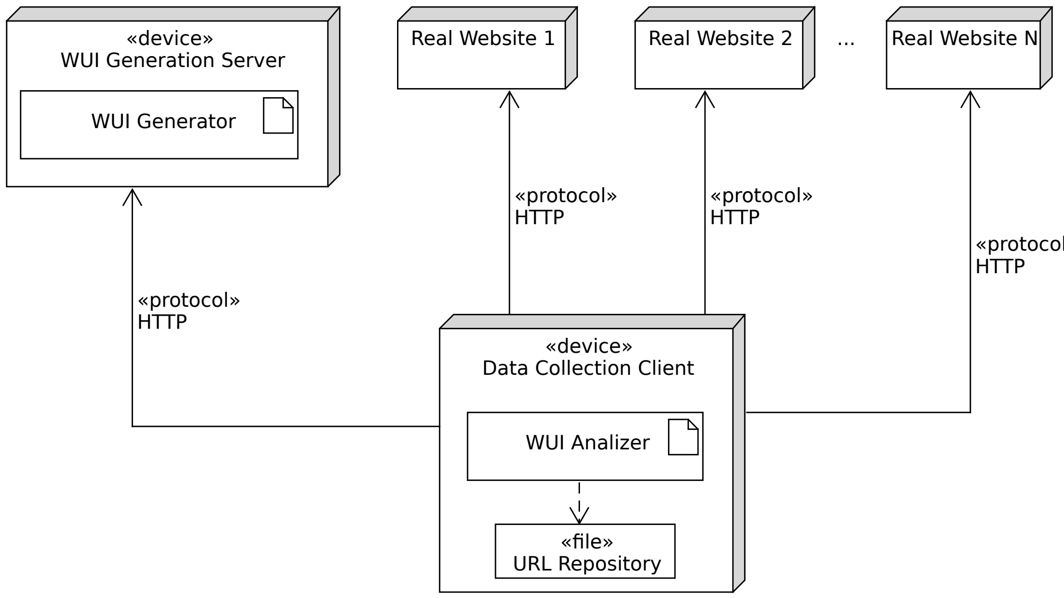


Figure 5.3. Dataset Collection - Deployment Diagram

In the following subsections, some of the core aspects of both applications are outlined.

WUI Generator

The prototype implementation of the WUI Generator consists in a web application developed using the *Tornado*[7] framework. Such a framework is selected because it has built-in functionality for developing web-based applications, and its templating system becomes ideal for the WUI generation strategy approached. The WUI generation strategy constitutes the core part of this application, and for such reason, it is explained in detail in the following paragraphs.

The WUI generation strategy has two key elements. The first one is the **GenerationManger**, which encapsulates and orchestrates all the WUI generation process. The second one is the **BaseWUIGenerator**, which is an abstraction of a generator implementation that generates WUIs from a set of configurations provided by the GenerationManager. The BaseWUIGenerator is extended to several concrete implementations that use different WUI frameworks.

[7]https://www.tornadoweb.org

The GenerationManager orchestrates the WUI generation process as follows: It first builds a generic randomizer using a seed parameter in a UUID4 format. If no seed is provided, it auto-generates one. Using the built randomizer, the GenerationManger picks a generator implementation from the ones registered and initializes it. It then generates the WUI configuration that represents the hierarchical structure of WUI elements and finally delegates the generation process with such structure to the generator implementation selected. The code snippet for this functionality is presented in Listing 5.3

```
async def generate_WUI(self, seed=None, root=None):
    if len(self.__generators) == 0:
        raise WUIGenertionException('There must be at least one
    generator implementation registered')
    try:
        seed = uuid.UUID(seed).int if seed else uuid.uuid4().int
        randomizer = Random(seed)
        generator = randomizer.choice(self.__generators)
        generator = generator(randomizer)
        wui_config = await self.__build_structure(randomizer, root)
        return generator.generate_WUI(wui_config)
    except ValueError:
        raise WUIGenertionException('The provided seed does not
    follow the UUID format')
```

Listing 5.3 Code Snippet - WUI Generation Orchestration

For building the WUI configuration, two kinds of elements are considered - composite elements and atomic elements -. Composite elements are those types of WUI elements that can contain elements of other types while atomic elements are those that are indivisible.

The structure that contains metadata about atomic elements has the following representation:

- **name**: The name of the atomic element
- **label**: The label that accompanies the atomic element if applies.
- **placeholder**: The placeholder that accompanies the atomic element if applies.
- **text**: The text of the atomic element.

The structure that represents composite elements is defined as follows:

- **name**: The name of the composite element

- **header**: The set of sub-elements that constitute the header of the composite element. They can be composite or atomic as well
- **content**: Contains the content of the composite element. Represented as an array of arrays. The outer array represents the rows of the content. Each inner array represents the set of sub-elements that are present in a single row. They can be composite or atomic.
- **footer**: The set of sub-elements that constitute the footer of the composite element. They can be composite or atomic.

The *__build_structure(randomizer, root)* method builds a random distribution of elements wrapped in the structured mentioned above. For building such structure, it counts with a predefined set of rules that define specific structure templates that contemplate possible layout variations. There are predefined rules for both atomic and composite WUI elements. Having control over the possible layout variations is a better solution compared to a purely random generation strategy since it avoids the problem of generating a hierarchy of elements that is far from reality.

In the case of atomic elements, the rules define the maximum text, label, and placeholder words that an element can have. Based on those rules, the structure of atomic elements is built with the help of the randomizer by generating the random values of the word-lengths for the text, label, and placeholder properties. For the random generation of the different texts that the atomic element can have, the Manually Annotated Sub-Corpus (MASC) of American English is used as a text repository [47]. It was selected for the prototype implementation because it allows having total control of the text repository. A dependency on a third party service was not desired since it usually imposes usage limit rates that can impede or delay the whole WUI generation process.

For composite elements, the rules define a maximum content size which represents the total number of rows of elements it can have, and several templates that describe layout variations for the header, content, and footer of the composite element when applicable. The structure of composite elements is built with the help of the randomizer by generating a random value for the content size and randomly choosing a setup of sub-elements from the available ones for the header, content, and footer respectively. Since the header, content, and footer refer to other composite or atomic elements, the process is executed *recursively* for each entry on those properties.

WUI Analyzer

The prototype implementation of the WUI Analyzer is an application with a very simple Command Line Interface (CLI). It includes implementations of *Parsers* that match the specificities of the WUI frameworks considered by the WUI Generator prototype. Additionally, it also includes two *Writers*. One writer implements the conventional structure of PascalVOC. The other writer aggregates statistics about the generated ground truths in a Comma Separated Value (CSV) file.

For the analysis of WUIs, it uses as a primary base *Selenium*[8], a portable framework that automates browsers and is widely used to test web applications. The WUI Analyzer retrieves URLs from a file-based URL registry, and for each of those URLs conducts an analysis that includes making a request to the URL, taking a screenshot of the rendered WUI that results from making the request, generating annotations by accessing its DOM with the help of the parser implementations, and writing those annotations to the formats supported by the implemented writers.

The file-based URL is alimented by two entities. The first one is a *Web Crawler* that collects URLs from linked websites on the internet. The second one is a URL generator that builds URLs with random seeds pointing out to the WUI Generator.

5.2. Combining UI Object Detection Analyzers

Part written by Victor Flamenco

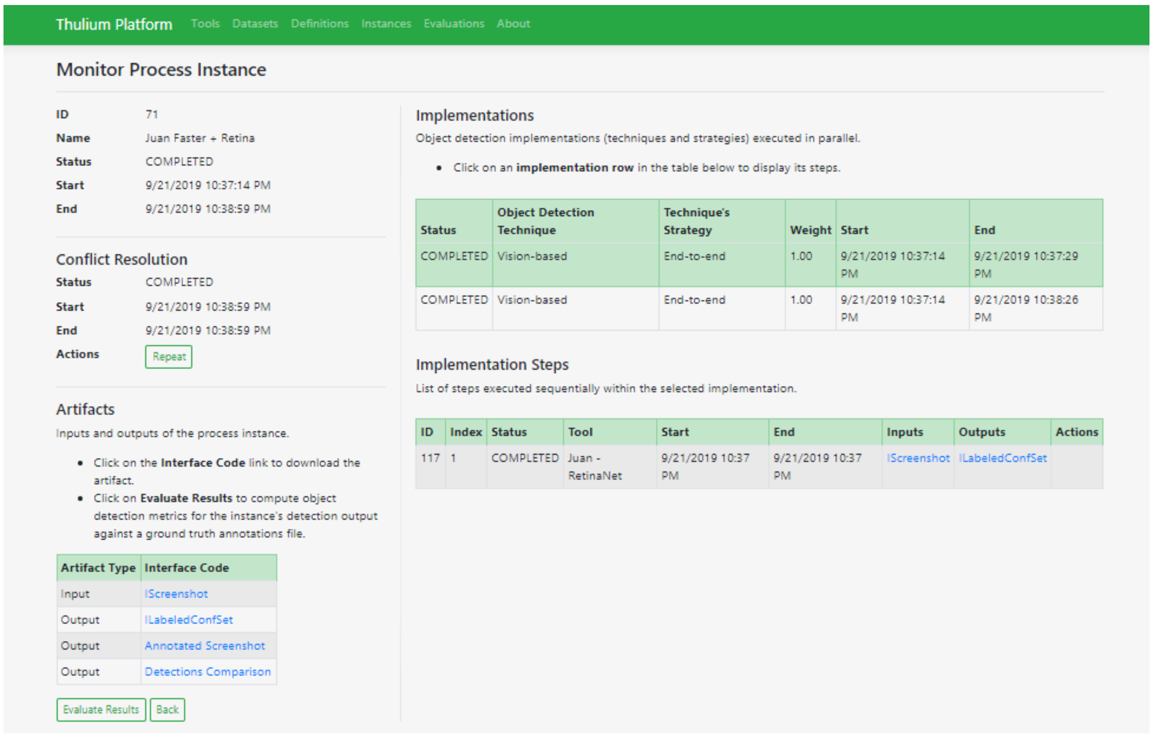
This subsection examines the concretization of the Thulium Platform, implemented as a web application that consists of a platform server and a web UI; it presents the platform architecture and exposes details about the most crucial components.

Architecture

The Thulium Platform is implemented as a layered web application using a *multi-tier* architecture, as shown in Figure 5.6. The guiding principle behind employing this pattern is the paradigm of loose coupling and separation of concerns. The platform layers evolve independently of each other as inter-layer interactions are well-defined interfaces that change seldom. Each layer provides a higher level service to the next one, adding specific functionality.

[8]https://www.seleniumhq.org/

Technology-wise, the platform utilizes the *Dotnet Core*[9] framework, an open-source, cross-platform, general-purpose development platform maintained by Microsoft and the .NET community[10].



Monitor Process Instance

ID	71
Name	Juan Faster + Retina
Status	COMPLETED
Start	9/21/2019 10:37:14 PM
End	9/21/2019 10:38:59 PM

Conflict Resolution

Status	COMPLETED
Start	9/21/2019 10:38:59 PM
End	9/21/2019 10:38:59 PM
Actions	Repeat

Artifacts

Inputs and outputs of the process instance.

- Click on the **Interface Code** link to download the artifact.
- Click on **Evaluate Results** to compute object detection metrics for the instance's detection output against a ground truth annotations file.

Artifact Type	Interface Code
Input	IScreenshot
Output	ILabeledConfSet
Output	Annotated Screenshot
Output	Detections Comparison

Evaluate Results Back

Implementations

Object detection implementations (techniques and strategies) executed in parallel.

- Click on an **implementation row** in the table below to display its steps.

Status	Object Detection Technique	Technique's Strategy	Weight	Start	End
COMPLETED	Vision-based	End-to-end	1.00	9/21/2019 10:37:14 PM	9/21/2019 10:37:29 PM
COMPLETED	Vision-based	End-to-end	1.00	9/21/2019 10:37:14 PM	9/21/2019 10:38:26 PM

Implementation Steps

List of steps executed sequentially within the selected implementation.

ID	Index	Status	Tool	Start	End	Inputs	Outputs	Actions
117	1	COMPLETED	Juan - RetinaNet	9/21/2019 10:37 PM	9/21/2019 10:37 PM	IScreenshot	ILabeledConfSet	

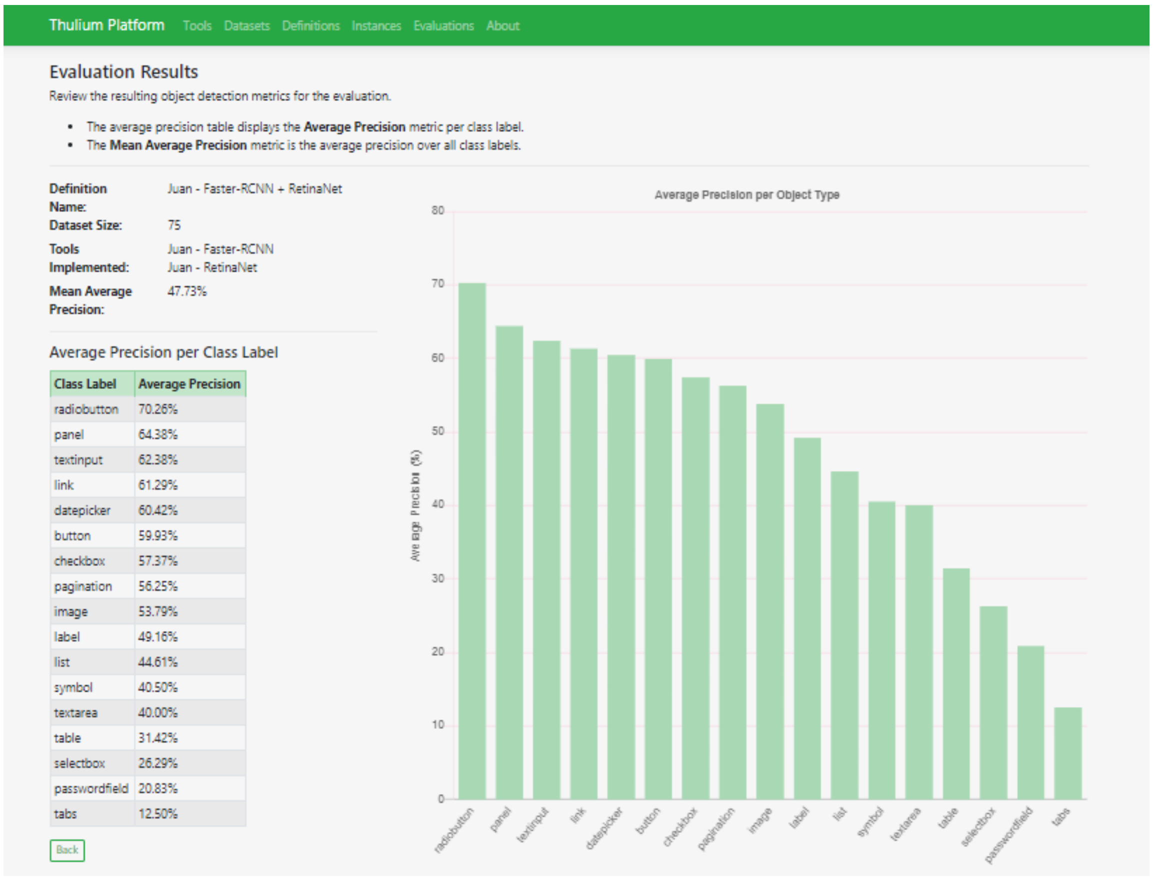
Figure 5.4. Process Instances Monitoring

Users interact with the *Views layer*, which implements server-side-rendered UIs using the *Razor syntax*[11]. The five components in this layer are:

- **External Tools Management**: a catalog of external object detection and conflict resolution tools. Users can register tools and configure the object detection sub-tasks they can execute. Moreover, for the Conflict Resolution tool external tool implemented in this thesis, parametrization is possible in this component.
- **Datasets Management**: a configurable catalog of testing datasets.
- **Process Definitions Management**: an interface for customization and instantiation of object detection workflows.

[9]https://github.com/dotnet/core
[10]https://docs.microsoft.com/de-de/dotnet/core/
[11]https://docs.microsoft.com/de-de/aspnet/core/mvc/views/razor

- **Process Instances Management**: an interface for monitoring the execution, audit data, and artifacts of running process instances (Figure 5.4), as well as evaluation of instance detection results.
- **Evaluations Controller**: allows evaluating process definitions against entire testing datasets, by comparing the detection results to the dataset's ground truth annotation files, using object detection metrics; it visually presents evaluation metrics in the form of tables and charts (Figure 5.5).



Evaluation Results

Review the resulting object detection metrics for the evaluation.

- The average precision table displays the **Average Precision** metric per class label.
- The **Mean Average Precision** metric is the average precision over all class labels.

Definition Name:	Juan - Faster-RCNN + RetinaNet
Dataset Size:	75
Tools Implemented:	Juan - Faster-RCNN Juan - RetinaNet
Mean Average Precision:	47.73%

Average Precision per Class Label

Class Label	Average Precision
radiobutton	70.26%
panel	64.38%
textinput	62.38%
link	61.29%
datepicker	60.42%
button	59.93%
checkbox	57.37%
pagination	56.25%
image	53.79%
label	49.16%
list	44.61%
symbol	40.50%
textarea	40.00%
table	31.42%
selectbox	26.29%
passwordfield	20.83%
tabs	12.50%

Figure 5.5. Visualization of Evaluation Metrics

The Views layer connects to the *Controllers layer*. Controllers implement the logic for processing user inputs and HTTP requests and returning appropriate responses. Similarly, the *API Controllers layer* provides interfaces for processing HTTP responses from external tools and for process definition instantiation. Both of these layers interact with the *Services layer*.

The Services layer implements a series of components that are automatically provided to the controllers and API controllers layers by the framework's dependency injection

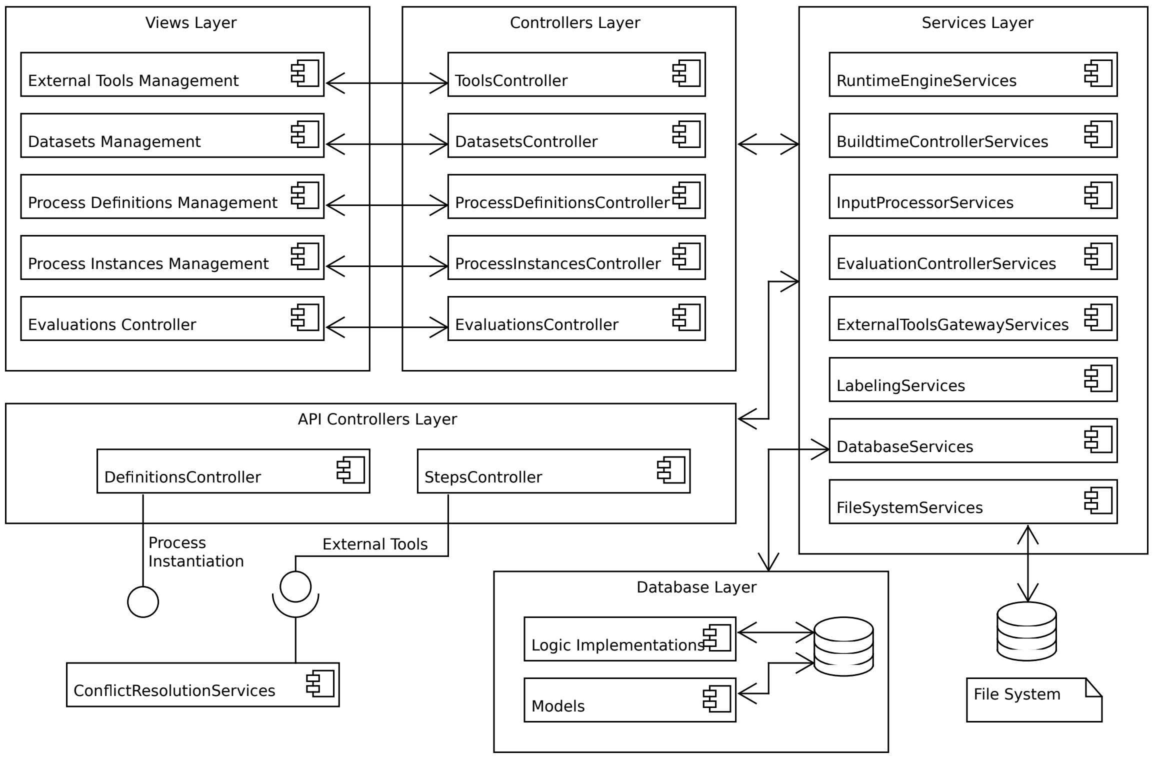


Figure 5.6. Thulium Platform Architecture

engine. These services implement the core logic of the Thulium platform, and this layer interacts with the two remaining system layers: the *File System layer* and the *Database layer*. The former one handles storage and retrieval of inputs and outputs, i.e., Portable Network Graphics (PNG) images, XML documents, and others; while the last layer establishes communication with the platform's database.

The database layer persists data from external tools, process definitions, instances, and tools evaluations. It utilizes *Entity framework Core*, a lightweight, extensible, open-source and cross-platform version of Entity Framework, a data access technology[12]; it serves as an Object-Relational Mapping (ORM) framework. Moreover, this layer implements a *SQL Server* connector to intercommunicate with the platform's database.

[12]https://docs.microsoft.com/en-us/ef/core/

Components

The upcoming list depicts the most challenging architecture components of section 4.3.3; then, the following subsections explain the strategies that this work follows to address their implementations.

- Input Processor
- Buildtime Controller
- External Tools Gateway
- Runtime Engine
- Evaluations Controller

Input Processor

The input-processing component is implemented in the **InputProcessorService** component. It prepares the artifacts that users input to the platform to provide the runtime engine with appropriate artifacts for any given object detection implementation. The engine requires either a DOM tree using the *PascalVOC* format stored as XML files, or screenshots of the rendered UI in PNG format. The image resolution is 1024x768 pixels as the experimentation in this work shows that this resolution provides sufficient amount of detail to identify small UI objects, and additionally, this resolution generates files light enough to avoid affecting the image processing speed.

This component offers built-in functionality to obtain a DOM tree from a website's URL. The website is fetched using a *WebClient* and its HTML content is downloaded as a string, and stored in the file system in XML format. On the other hand, for feeding a screenshot of a rendered interface, alternatives are: the component receives an image file (or multiple files), in which case no further input processing is necessary, or the component receives the site's URL and performs processing to obtain a screenshot or a series of screenshots of the rendered page.

Screenshots are obtained using the *Selenium*[13] package, a tool for automating web applications for testing purposes. The input processor uses this plugin to render a given website using its URL and a browser's driver -in this case the Chrome driver [14]- and to capture a screenshot with resolution 1024x768. If the site's height is greater than 768, then multiple screenshots are taken by scrolling the site down in steps of

[13]https://www.seleniumhq.org/
[14]https://chromedriver.chromium.org/

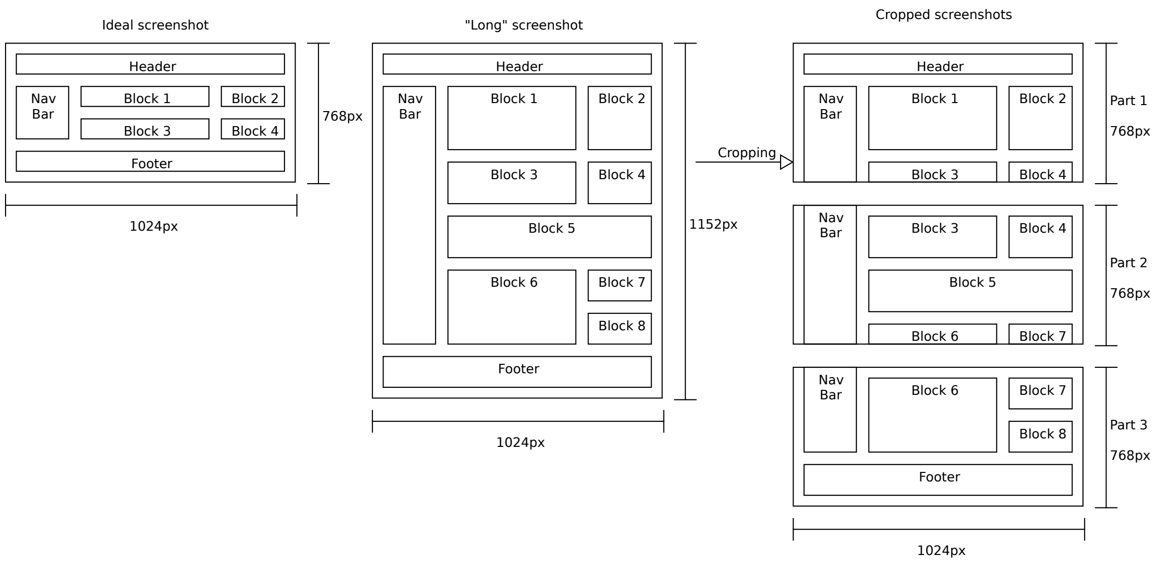


Figure 5.7. Input Processor - UI Screenshots Generation

768/2 pixels, which guarantees that the majority of the objects can be fully included in at least one screenshot (see Figure 5.7). For handling dynamic websites that allow virtually infinite vertical scrolling, a configurable parameter is used to limit the number of screenshots to take per website.

Buildtime Controller

The build-time controller, implemented in the **BuildtimeControllerService** component, receives a configuration of object detection implementations provided by a platform user, processes this configuration by translating it into a process definition understandable by the engine. The platform's database layer persists this definition tables dedicated to this controller. A diagram of this part of the database is displayed in Figure 5.8.

Besides a definition name for identification purposes, process definitions include a series of implementation configurations, where each of these is a fully-independent object detection technique following one of the strategies exposed in section 4.3.3. Also, definitions describe a series of steps per implementation, which map object detection sub-tasks with tool providers for execution. According to the Thulium reference process, each technique/strategy combination has a specific number of sub-tasks; in the process definition, each step describes which sub-tasks are performed by the external tool, specified using the range *[FromSubTask - ToSubTask]*.

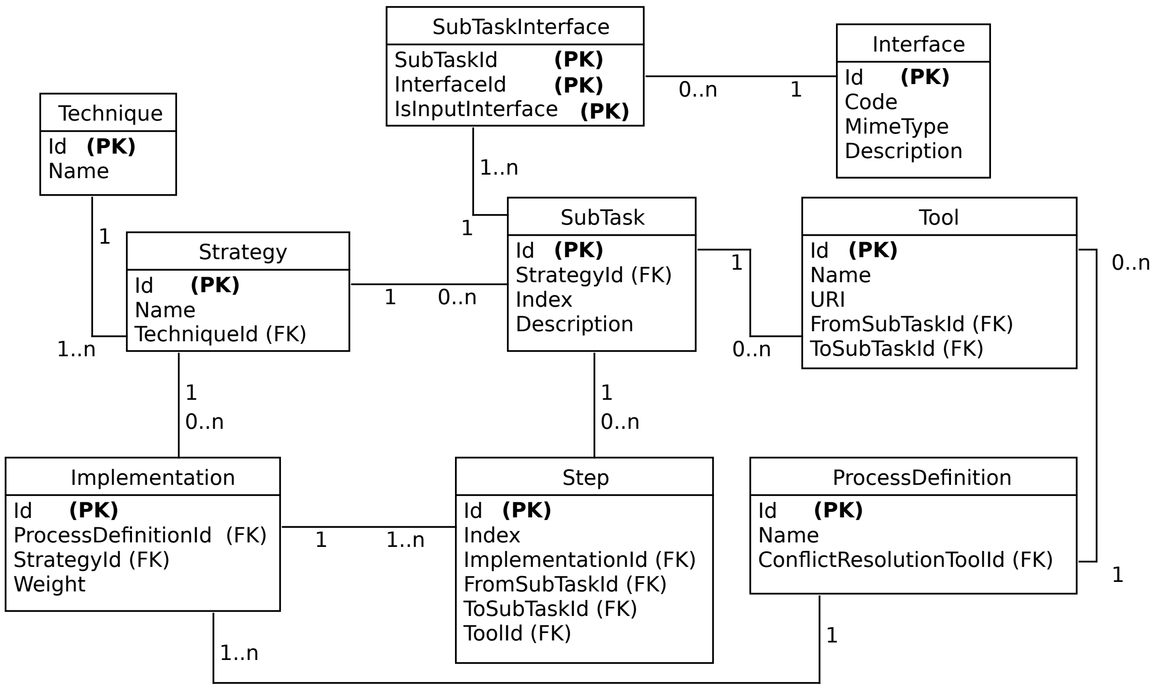


Figure 5.8. Buildtime Controller - Database Diagram

Among the activities of the build-time controller is the validation of coherence of the step sub-tasks ranges. Each implementation includes a weight value, representing the percentage of importance that it is assigned when aggregating results of all implementations. The last process definition field (conflict resolution tool) references an external tool in charge of solving output inconsistencies across implementations.

External Tools Gateway

The **ExternalToolsGatewayService** component implements a role-based repository of services; it describes the mechanics of working with external object detection services and acts as a connector between external object detection providers and the platform, specifically with the Runtime Engine and the Evaluations Controller. The Gateway interacts with two application services: the *ToolsService* and the *SubTasksService* (see Figure 5.9). The former service is used to query, and filter persisted external tools, and it uses the second service for querying possible sub-tasks.

The core functionality of the gateway service, besides registering external object detection and conflict resolution tools (*RegisterObjectDetection(...)* and *RegisterConflictResolution(...)*, respectively), is to execute them. The *ExecuteTool(...)* procedure is called by the Runtime Engine whenever a step requires execution. This method

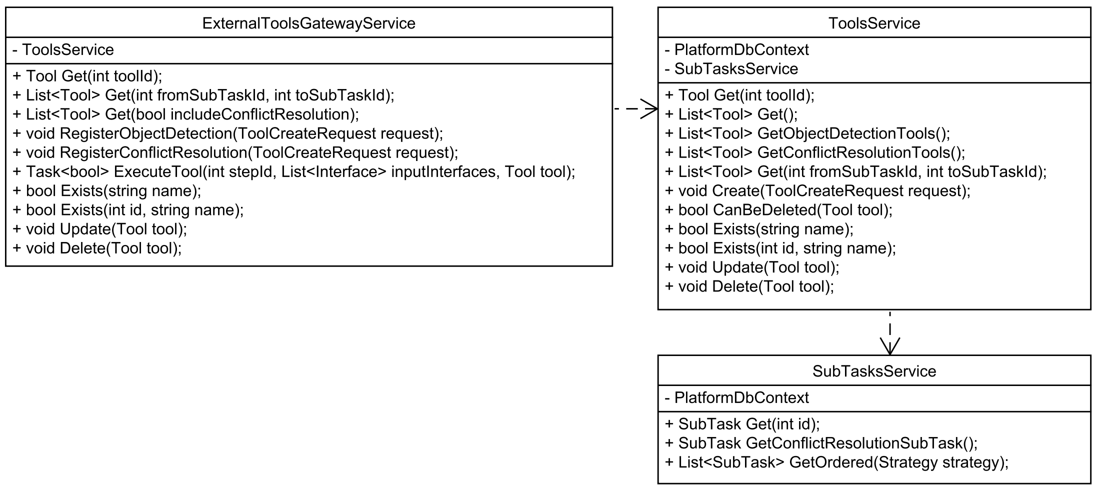

ExternalToolsGatewayService
- ToolsService
+ Tool Get(int toolId);
+ List<Tool> Get(int fromSubTaskId, int toSubTaskId);
+ List<Tool> Get(bool includeConflictResolution);
+ void RegisterObjectDetection(ToolCreateRequest request);
+ void RegisterConflictResolution(ToolCreateRequest request);
+ Task<bool> ExecuteTool(int stepId, List<Interface> inputInterfaces, Tool tool);
+ bool Exists(string name);
+ bool Exists(int id, string name);
+ void Update(Tool tool);
+ void Delete(Tool tool);
ToolsService
- PlatformDbContext
- SubTasksService
+ Tool Get(int toolId);
+ List<Tool> Get();
+ List<Tool> GetObjectDetectionTools();
+ List<Tool> GetConflictResolutionTools();
+ List<Tool> Get(int fromSubTaskId, int toSubTaskId);
+ void Create(ToolCreateRequest request);
+ bool CanBeDeleted(Tool tool);
+ bool Exists(string name);
+ bool Exists(int id, string name);
+ void Update(Tool tool);
+ void Delete(Tool tool);
SubTasksService
- PlatformDbContext
+ SubTask Get(int id);
+ SubTask GetConflictResolutionSubTask();
+ List<SubTask> GetOrdered(Strategy strategy);

Figure 5.9. External Tools Gateway Service - Class Diagram

sends an HTTP POST request to the tool's URI and adding the Step ID as a path parameter. Listing 5.4 shows an example request. *ExecuteTool(...)* additionally sets the runtime engine as a response subscriber. An appropriate payload is attached to the request, based on the input interface required by the step's initial sub-task.

```
POST /tools/5 HTTP/1.1
Host: vsrstud05.informatik.tu-chemnitz.de:5000
Content-Type: multipart/form-data; boundary=----
    WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="input"; filename="image.png"

------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

Listing 5.4 HTTP Request - External Tool Execution Example

Runtime Engine

During runtime control, the engine, implemented in the **RuntimeEngineServices** component, interprets a process definition and creates and executes new process instances, where task coordination and completion happen, by invoking appropriate external tools. Amid process instance execution, workflow control data is persisted to allow execution auditing and recovery from failures. The runtime engine takes two inputs: process definitions and inference datasets consisting of DOM trees or screenshot images. The implementation of the Engine (described in section 4.3.3) consists of **four steps**.

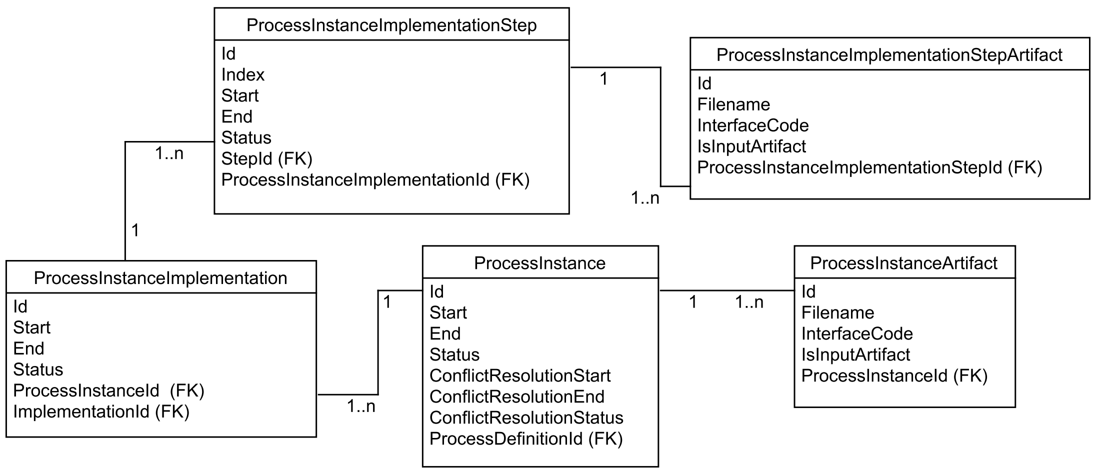
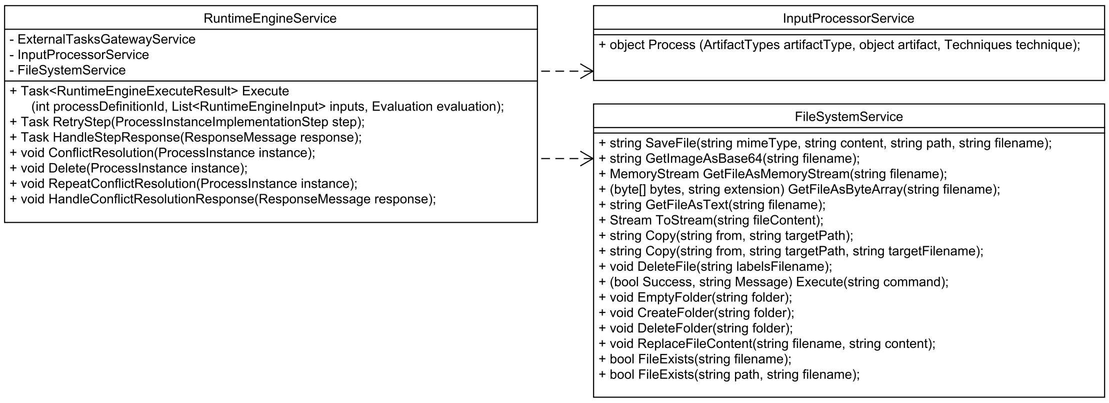


Figure 5.10. Runtime Engine - Database Diagram

1. Interpretation: the Runtime Engine prepares the process execution by creating a process instance from it and persisting the series of object detection implementations that it contains, and per implementation, persisting the sequence of steps to execute. The engine initializes each implementation (and each of its steps) with a *NotStarted* status; it stores references to tools registered at the gateway, that are responsible for steps execution. Moreover, the engine stores the input to feed the implementation's initial step. The diagram in Figure 5.10 serves as a model for the part of the database related to the runtime engine.


```
RuntimeEngineService
- ExternalTasksGatewayService
- InputProcessorService
- FileSystemService
+ Task<RuntimeEngineExecuteResult> Execute
    (int processDefinitionId, List<RuntimeEngineInput> inputs, Evaluation evaluation);
+ Task RetryStep(ProcessInstanceImplementationStep step);
+ Task HandleStepResponse(ResponseMessage response);
+ void ConflictResolution(ProcessInstance instance);
+ void Delete(ProcessInstance instance);
+ void RepeatConflictResolution(ProcessInstance instance);
+ void HandleConflictResolutionResponse(ResponseMessage response);

InputProcessorService
+ object Process (ArtifactTypes artifactType, object artifact, Techniques technique);

FileSystemService
+ string SaveFile(string mimeType, string content, string path, string filename);
+ string GetImageAsBase64(string filename);
+ MemoryStream GetFileAsMemoryStream(string filename);
+ (byte[] bytes, string extension) GetFileAsByteArray(string filename);
+ string GetFileAsText(string filename);
+ Stream ToStream(string fileContent);
+ string Copy(string from, string targetPath);
+ string Copy(string from, string targetPath, string targetFilename);
+ void DeleteFile(string labelsFilename);
+ (bool Success, string Message) Execute(string command);
+ void EmptyFolder(string folder);
+ void CreateFolder(string folder);
+ void DeleteFolder(string folder);
+ void ReplaceFileContent(string filename, string content);
+ bool FileExists(string filename);
+ bool FileExists(string path, string filename);
```

Figure 5.11. Runtime Engine - Class Diagram

2. Execution and monitoring: the Engine starts the instance with the *Execute(...)* method (see Figure 5.11). It starts all implementations in parallel and changes their statuses to *Executing*. For each implementation, it feeds the initial step with the appropriate sub-set of items from the test dataset and triggers the initial step.

Starting an instance step implies sending a request to the external service that is assigned for the step, being assisted by the external tools gateway (see section 5.2).

The engine registers the start time and setting the step status to *Executing*. The message's payload includes a reference to the unique identification number of the step and appropriate input content. The gateway subscribes the runtime engine to the completion event on the service. Once the external service has processed the task, it notifies the engine back for further processing, by sending an HTTP POST request, attaching a result content, and in a path parameter, a reference to the Step ID provided at subscription time (see Listing 5.5). In case of error, the message is formatted as described in Listing 5.6.

```
POST /api/v1/steps/5 HTTP/1.1
Host: vsrstud05.informatik.tu-chemnitz.de:5000
Content-Type: multipart/form-data; boundary=----
    WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="Files"; filename="detections.
    xml"

------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

Listing 5.5 HTTP Request - External Tool Success Example

```
POST /api/v1/steps/5/error HTTP/1.1
Host: vsrstud05.informatik.tu-chemnitz.de:5000
Content-Type: application/json

{
    "errorMessage": "error message"
}
```

Listing 5.6 HTTP Request - External Tool Error Example

The *HandleStepResponse(...)* method receives the response payload from the external tool, persists the response artifacts, and logs audit data. If the task completes successfully, the method pipelines the payload to the next step in the implementation's steps sequence, using *ExecuteNextStep(...)*. Otherwise, the engine allows manually triggering a retry action on the failed step (*RetryStep(...)*). The new possible step statuses are *Completed* and *Error*. When the engine finds no further steps in the sequence of the current implementation, the *CompleteImplementation(...)* method populates audit information for completion of the implementation. This method additionally checks whether all instance implementations are already processed. In a positive case, it triggers the conflict resolution step. Finally, once the resolution tool responds with a unified detections result, the *HandleConflictResolutionResponse(...)* method is fired. Instances with one and only one implementation omit conflict resolution.

3. Output generation: the engine utilizes *LabelingService* to visually annotate the input UI screenshot using the instance's output annotations. The service draws rectangular blocks with their respective positions and sizes and writes the object types and confidence scores above each rectangle. The engine allows downloading these drawings in PNG format. Besides, the engine allows the inspection of an annotated image that contains all rectangles and labels from the instance's implementations before the resolution of conflicts, enabling to visualize the heterogeneous detection predictions easily.

4. Output evaluation: the engine utilizes a user-uploaded ground truth labels XML file in PascalVOC format and assesses the instance detection outputs based on the **AP** per object class and the overall **mAP**. These metrics are calculated with assistance of the Object Detection Metrics library[15].

The library expects a special labeling format, stored as a plain text file, in which, each line represents a detected object. The format for each detected object line in ground truth labels files is: *objectClass left top width height*, e.g., *button 36 111 198 416*. On the other hand, the line format in detector prediction labels files is: *objectClass confidence left top width height*, e.g., *button 0.85 35 110 200 421*.

The engine converts, therefore, the instance's detection outputs and the ground truth PascalVOC labels to this format for triggering the metrics calculation. Metric calculation results include a mAP overall object classes and a list of average precisions per object class.

Evaluations Controller

The **EvaluationControllerService** component implements functionality for automatically testing the detection quality of isolated external tools, as well as combina-

[15]https://github.com/rafaelpadilla/Object-Detection-Metrics

tions of tools, against configurable datasets, enabling iterative improvements in the object detection metrics.

For the **first task** of the controller, conceptualized in section 4.3.3, the *GenerateProcessDefinitions()* method (see Figure 5.12) iterates over the sub-tasks of each strategy in the Thulium Reference Process, and queries available external tools at the gateway. The algorithm generates all possible combinations of series of sub-tasks and conflict resolution tools. Generated combinations are turned into process definitions with the assistance of the *BuiltimeControllerService*. Each definition contains only one implementation for isolated tool evaluation, having a default *weight of 1.0*.

The *DatasetsController* (Figure 5.6) implements the controller's **second task**: a repository of testing datasets. It enables persisting pairs of input UI representations with their corresponding ground truth annotation files. Datasets are then replicable whenever an evaluation is triggered in the next task.

Running an evaluation (**third task**) requires three inputs:

- a process definition,
- a testing dataset, and
- an IoU threshold ε parameter used for the computation of metrics as described in section 4.3.3.

The Evaluations Controller requests the Runtime Engine to execute one instance of the process definition per element in the testing dataset.

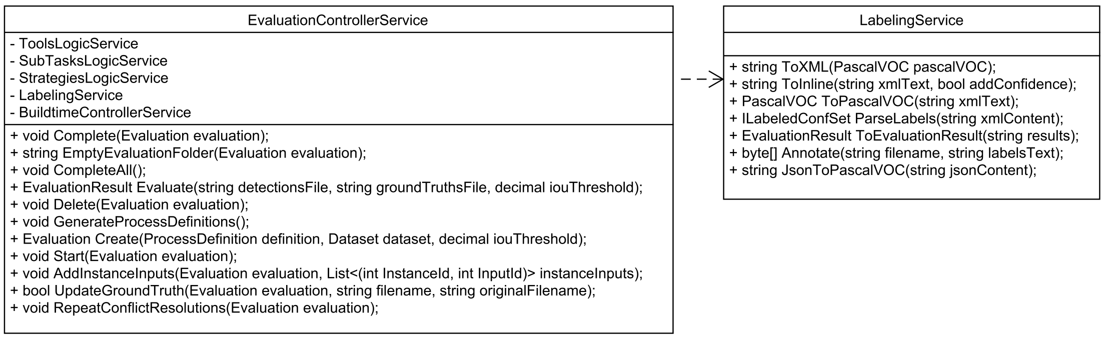

```
EvaluationControllerService
- ToolsLogicService
- SubTasksLogicService
- StrategiesLogicService
- LabelingService
- BuildtimeControllerService
+ void Complete(Evaluation evaluation);
+ string EmptyEvaluationFolder(Evaluation evaluation);
+ void CompleteAll();
+ EvaluationResult Evaluate(string detectionsFile, string groundTruthsFile, decimal iouThreshold);
+ void Delete(Evaluation evaluation);
+ void GenerateProcessDefinitions();
+ Evaluation Create(ProcessDefinition definition, Dataset dataset, decimal iouThreshold);
+ void Start(Evaluation evaluation);
+ void AddInstanceInputs(Evaluation evaluation, List<(int InstanceId, int InputId)> instanceInputs);
+ bool UpdateGroundTruth(Evaluation evaluation, string filename, string originalFilename);
+ void RepeatConflictResolutions(Evaluation evaluation);
```
```
LabelingService
+ string ToXML(PascalVOC pascalVOC);
+ string ToInline(string xmlText, bool addConfidence);
+ PascalVOC ToPascalVOC(string xmlText);
+ ILabeledConfSet ParseLabels(string xmlContent);
+ EvaluationResult ToEvaluationResult(string results);
+ byte[] Annotate(string filename, string labelsText);
+ string JsonToPascalVOC(string jsonContent);
```

Figure 5.12. Evaluations Controller - Class Diagram

The Engine notifies the Controller once all process instances have ended. At this point, the Controller starts the preparation for evaluation. As a first step, it pipelines the outputs of all process instances as input to the evaluation algorithm. Similarly to the Runtime Engine, this Controller also utilizes the Object Detection Metrics library for this computation.

The inputs for the evaluation are all the resulting detections from the process instances (of type **ILabeledConfSet**), and the ground truth annotation files from the testing dataset. The output metrics are: **AP** per class and the overall **mAP**.

As a final feature, to facilitate iterative improvements of detection quality, the *RepeatConflictResolutions(...)* method recomputes the evaluation metrics after users modify any definition or conflict-resolution parameters, without re-executing the external tools against the same dataset, a resource- and time-consuming task. This process consists of three steps. First, the Controller clears any previous evaluation results and metrics; second, it re-triggers the *Conflict Resolution* step (and strictly only this step) for each process instance related to the current evaluation; and third, it recomputes the evaluation metrics once all instances have ended.

5.3. Summary

This chapter explains the aspects to build a prototype of a solution capable of combining different UI object detection strategies to improve overall detection results. At the start, it outlines a *unified* solution which addresses all the conceptual aspects exposed in chapter 4. Subsequently, a zoom-in on the most challenging aspects of the implementation following the design considerations stated in section 4.2 and section 4.3 is carried out separately per solution part.

The first solution part details implementation considerations of a solution including two object detectors that can learn progressively as well as an automated dataset collection approach. The second solution part covers the implementation details of a workflow-based platform that allows the integration of different object detection strategies and offers means for conflict resolution and results aggregation.

The upcoming chapter consists of an assessment of the implementation prototypes described in this chapter. It includes the description of an evaluation methodology, as well as the different results obtained when following the evaluation methodology to assess the prototypes implemented.

6. Evaluation

The solution implementation detailed in chapter 5 concretizes the overall thesis concept. It includes a pair of DNN-based object detection tools and the Thulium Platform, a web application for integrating multiple object detection tools for improved object detection results. The current chapter assesses these implementation prototypes in order to discover whether they satisfy the thesis' requirements and to test whether the thesis hypotheses are accepted ultimately.

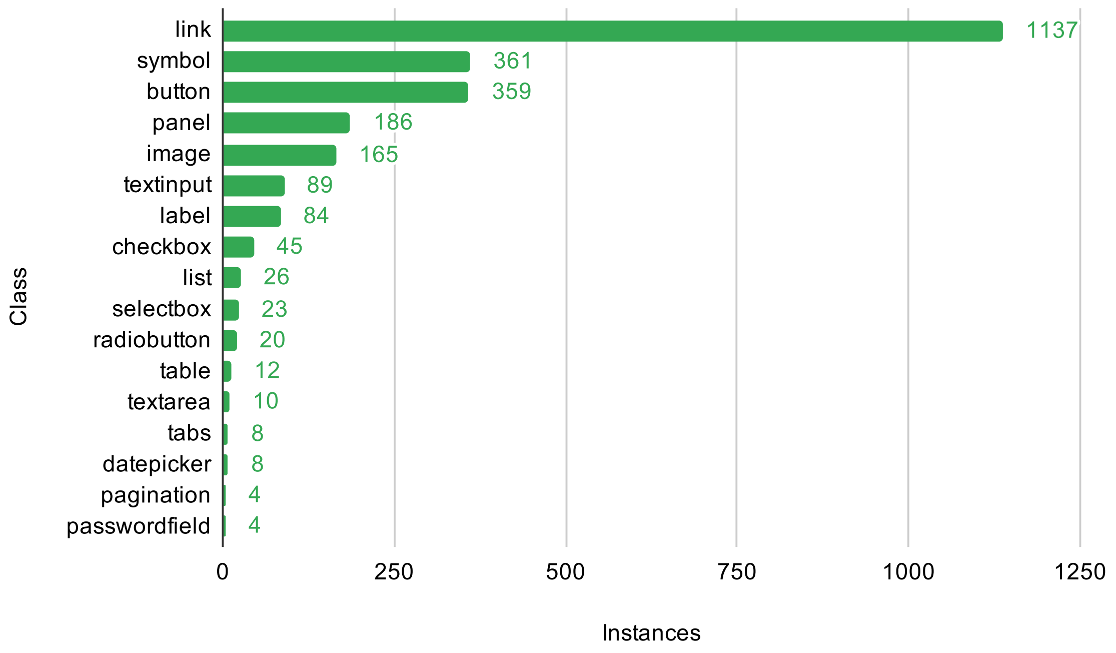


Figure 6.1. Evaluation Testing Dataset - Frequency of Distributions

For both of the thesis solution parts, evaluations utilize a common **testing dataset**, consisting of 75 randomly-selected UI screenshots and their corresponding ground truth annotation files, with the object types distribution displayed in Figure 6.1. The testing dataset domain covers newspapers, wikis, educational (colleges, universities, research), non-profit organizations, and form-based data collection (user registration and survey) sites. Furthermore, the dataset is entirely derived from real websites, meaning that it includes no artificial UIs.

The following subsections individually describe the solutions assessment for each thesis solution part; they present the evaluation environments, measurements, and procedures first, and subsequently, expose the evaluation results and compare the two proposed solutions to the existing SotA groups described in chapter 3.

6.1. UI Object Detection Using DNNs

Part written by Juan Sebastian Quintero

This section evaluates the implementation of two isolated object detectors to assess whether they satisfies its purpose and to finally test Hypothesis A.

6.1.1. Setup

The evaluation setup for this part of the thesis describes the environmental aspects in which the evaluation of the prototype implementation exposed in section 5.1 is conducted. The first subsection outlines the evaluation configuration regarding the dataset collection strategy. The following subsection describes the whole training environment comprising configuration parameters and outcomes from the training process on each of the object detection models. Moreover, the evaluation procedure, as well as the different measurements considered for the assessment, are also outlined.

Data Collection

The data collection evaluation procedure consist in the assessment of approximately 23000 training examples containing around 808983 UI objects generated using the **WUI Analyzer** in combination with the **WUI Generator**. Each of the training examples is comprised of an image with a resolution of $1024px \times 768px$ and its respective annotations in the PascalVOC XML format. From the total amount of training examples, 17000 training examples are generated from real websites using a repository of URLs fed by a web crawler. The other 6000 are generated from a repository of URLs pointing out to the WUI Generator.

The 50% of the training examples are generated using the *Firefox WebDriver* and the other 50% is created using the *Chrome WebDriver*. Following this approach has the advantage of introducing an additional level of variability in the data that results from the rendering process of standard WUI elements in different browsers.

For such an assessment, the frequency of distribution of object instances per class is computed. A threshold of 400 instances per class is defined. This number is considered after the evaluation of the frequency of distribution of the PascalVOC dataset.

Object Detection

For the evaluation of the **Faster-RCNN** and **RetinaNet** implementations provided (see section 5.1) Three incremental training stages are conducted separately on each of them using the training dataset collected. For the training process, the training examples are further subdivided randomly in a *train set* consisting of the 90% of the training examples and a *test set* consisting of the remaining 10%.

Both object detection implementations are trained using pre-trained models on the COCO dataset as a starting point for fine-tuning in the first training stage. Such models are provided by the model zoo of the *TensorFlow Object Detection API*. For the Faster-RCNN implementation, the *faster_rcnn_resnet101_coco* pre-trained model is used. In the same order, the *ssd_resnet50_fpn_coco* pre-trained model is used for the RetinaNet implementation.

The second and third training stages use as a base the models that result from the previous stages and thus, allow incrementally testing the whole training process. Additional details regarding training are stated in Table 6.1 and Table 6.2.

	1st Iteration	**2nd Iteration**	**3rd Iteration**
Training Time	12h32m	18h18m	27h5m
Batch Size	4	4	4
Training Steps	29215	36000	36000
RPN - Localization Loss	0.48	0.44	0.45
RPN - Objectness Loss	0.27	0.27	0.28
Classifier - Localization Loss	0.25	0.20	0.08
Classifier - Classification Loss	0.29	0.20	0.04
Total Loss	1.29	1.11	0.89
mAP	0.66	0.85	0.94

Table 6.1. Training Details - Faster-RCNN

	1st Iteration	**2nd Iteration**	**3rd Iteration**
Training Time	11h52m	6h11m	20h9m

Batch Size	4	4	4
Training Steps	29215	36000	36000
Classifier - Localization Loss	0.12	0.05	0.06
Classifier - Classification Loss	0.30	0.16	0.17
Regularization Loss	0.23	0.09	0.09
Total Loss	1.29	1.11	0.89
mAP	0.45	0.73	0.86

Table 6.2. Training Details - RetinaNet

Having the two object detection models trained and ready for prediction, the evaluation phase is carried out using the ***Thulium Platform*** (see section 5.2). The evaluation consists of assessing the two object detection models *separately* against a ground-truth comprised of 75 different examples. The results of the evaluation process contain individual AP values per class and a mAP that gives a final score to the object detector in a summarized manner. Performing the evaluation stage through the Thulium Platform is important since it allows having comparable performance results given by each object detector against their combined output driven by the platform. The mAP scores appearing in Table 6.1 and Table 6.2 are the final metrics computed by the TensorFlow evaluation process on the *test set* during training. These metrics should only be considered as benchmarks. A more detailed version of such metrics can be checked in Table C.1 and Table C.2.

6.1.2. Results

This subsection shows the different results obtained when assessing both the data collection strategy and the object detection implementations trained with the dataset generated.

Data Collection

For the data collection strategy, a frequency of distribution representing all the different object instances per class in the collected dataset is computed. Figure 6.2 illustrates the results of such computation.

On on hand, the classes that register most object instances across the dataset correspond to “link”, “button” and “symbol” with 191406, 69455 and 66147 instances respectively. Such first three classes represent approximately the 68% of the total of object instances comprised by the whole dataset.

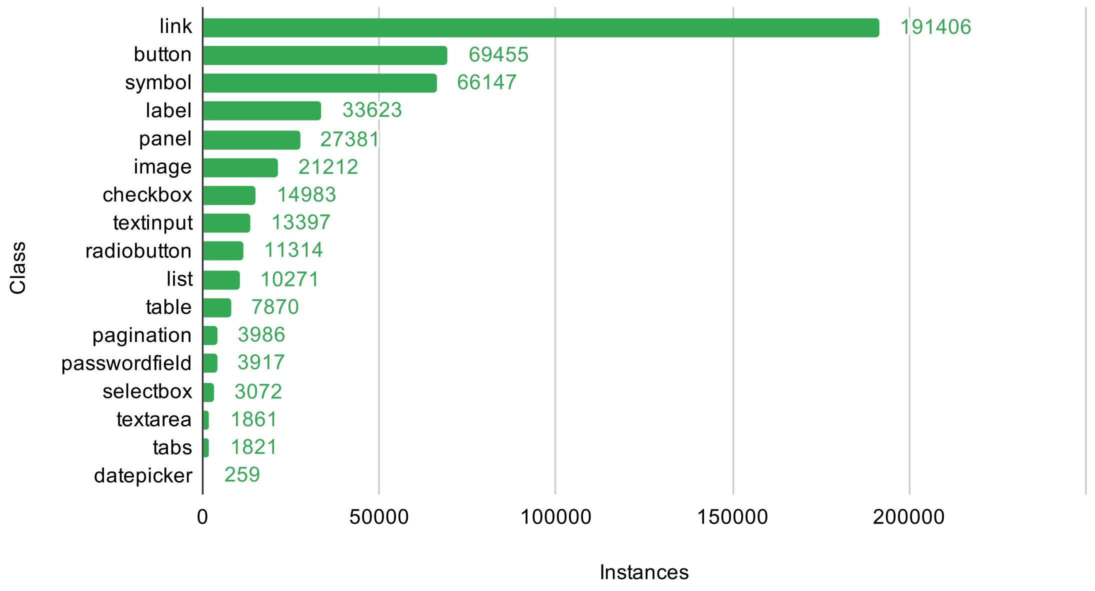


Figure 6.2. Training Dataset - Frequency of Distributions

On the other hand, the classes with fewer object instances correspond to “datepicker”, “tabs”, and “textarea” with 259, 1821 and 1861 respectively. From all the considered classes it can be observed that the “datepicker” class is the only one unable to reach the defined threshold. Analyzing the reason that leads to insufficient date-pickers in the dataset, one can infer that it happens because date-pickers are nowadays rarely inline elements in WUIs and instead, need previous actions from users in order to be rendered such as clicking on a text box, or a button. Such behavior is even standardized by the HTML5 specification with date-type inputs [1].

Object Detection

For the evaluation of the Faster-RCNN and RetinaNet implementations, both object detection implementations are assessed against a test dataset comprising 75 different entries. Figure 6.4 shows the mAP scores reached by both implementations. Also, Figure 6.4 and Figure 6.5 illustrate the different per-class APs obtained by them.

From the mAP scores, it can be seen that Faster-RCNN outperforms RetinaNet with a score of 42.19% against a score of 31.12%. Such results are also in concordance with the evaluation results produced during the training phase.

[1] https://html.spec.whatwg.org/multipage/input.html

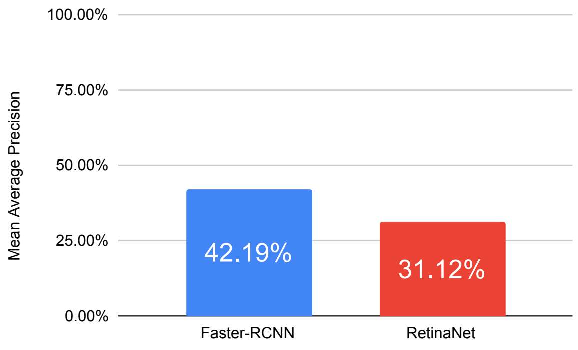


Figure 6.3. mAP Metrics - Faster-RCNN and RetinaNet

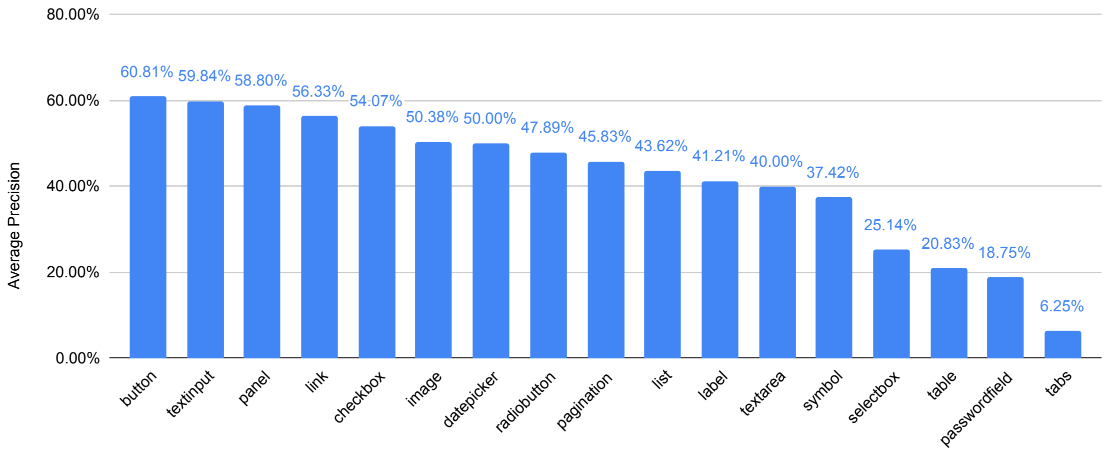


Figure 6.4. AP Metrics - Faster-RCNN

One the one hand, the results of per-class APs for Faster-RCNN show it has its best performance on detecting buttons with an AP value of 60.81%. Additionally, the results show a deficient performance on detecting tabs with an AP value of just 6.25%.

On the other hand, the results of per-class APs for RetinaNet evidence an acceptable performance on date-pickers with an AP value of 62.50%. Similarly, as with the Faster-RCNN, the results also evidence a deficient performance on tabs with an AP value of 0.60%.

By comparing the results of both models, it can be noticed that the RetinaNet implementation is only better at predicting date-pickers. Additionally, both models share text-inputs as their second-best predictions and password-fields as their second-worst predictions. Moreover, both models have very low AP rates on tabs; it can occur due

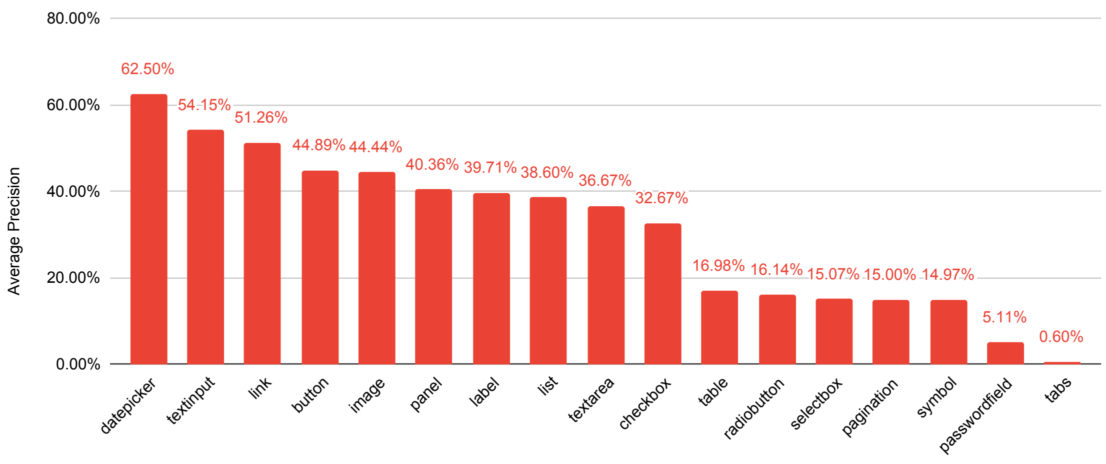


Figure 6.5. AP Metrics - RetinaNet

to miss-labeled data since there are limited ways to identify tabs in an automated fashion. It can also be observed that there is not a strict relation between per-class AP values obtained and the number of class object instances used for training.

The AP values are calculated having as a base isolated precision/recall curves for each of the classes. The AP value is the resulting area under the curve. The Appendix D contains all the precision/recall curves for Faster-RCNN and RetinaNet in merged charts. An important finding in those charts is that in general Faster-RCNN gets higher precision values, but RetinaNet has better recall values; nevertheless, contrary to precision values recall values never reach 1.0. Figure 6.6 and Figure 6.7 are some representatives evidencing this finding.

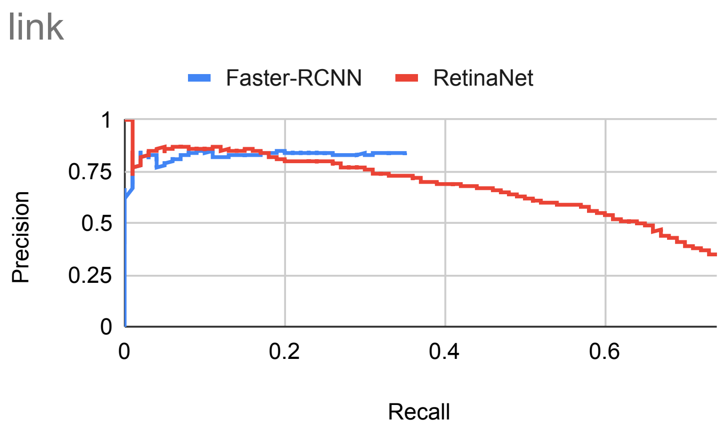


Figure 6.6. Precision/Recall Curve - Link

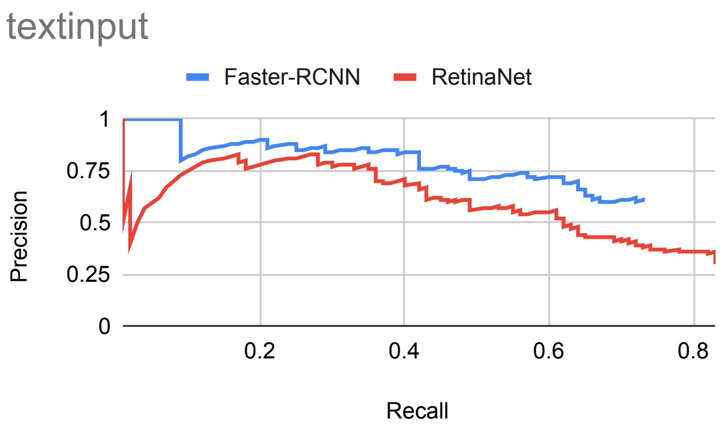


Figure 6.7. Precision/Recall Curve - Textinput

6.1.3. Discourse

This subsection assesses the final solution proposed against each of the requirements defined at section 2.1. A summarized version of this assessment can be found in Table 6.3 and Table 6.4.

RA1. Detection of the Parts of a UI

The solution offers two different models trained to detect each of the 17 non-OCR-related UI object types previously defined by this requirement. After the execution of the assessment and further observations on the evaluation results, the following aspects are identified:

Both models respond well regarding training (see Table 6.1 and Table 6.2). It means the models can learn specificities and patterns of the different object types considered in general. By comparing the mAP scores of the two models during training, it can be observed that both object detectors increase their scores after each training iteration. Faster-RCNN has slightly higher mAP values for each of the training iterations performed in relation to the mAP scores of RetinaNet. Furthermore, another good indicator is the reduction of the total loss rates in both of the models.

The evaluation results on Faster-RCNN and RetinaNet show that the models have some difficulties in differentiating objects of some particular classes given the low inter-class variability introduced by the domain. Some of the most notable findings include:

- Buttons with gray-scaled backgrounds and inner-text vs. text inputs.
- Small buttons, including a symbol vs. symbols.
- Checkboxes vs. radio-buttons vs. symbols.
- Small images vs. symbols.
- text inputs vs. password fields vs. select-boxes.
- panels with background-images vs. images.

Additionally, both models have better performance when detecting objects in clean user interfaces, e.g., user interfaces with white backgrounds, low visual clutter, and good contrast between UI objects and background. One of the reasons of this behavior might be the percentage of "clean" WUIs making part of the training dataset. As a consequence, those WUIs would have a more significant influence in the training process and condition the models to learn most from them.

Despite those facts, both models support prediction considering the 17 UI object types, an thus, fully satisfying the requirement.

RA2. Automation of Object Detection

The solution offers a REST-based API that encapsulates the whole prediction process, allowing an end-to-end execution in an automated fashion. It receives an image of a UI as the only input means and returns a PascalVOC-like format including the different predictions for the objects contained in the UI with their respective bounding boxes. This approach easies the whole prediction process by encapsulating most of the additional steps required by raw detectors. Furthermore, it allows for straightforward integration with third-party systems. Thus, the solution fully satisfies the requirement for process automation.

RA3. Visual Input

The solution considers images as the only input means for the detection of objects. Additionally, it counts with a sliding window approach that normalizes the input sizes and allows the model to receive images of varying sizes. By considering those aspects, this requirement is cataloged as fully satisfied by the solution proposed in this thesis.

RA4. Deep Learning

The solution proposed includes two different object detection mechanisms (RetinaNet and Faster-RCNN). One is a one-stage object detector, and the other one is a two-stage object detector. Both of the meta-architectures include CNNs for extracting features, detecting objects and detecting bounding boxes. Additionally, they are cataloged as supervised learning. For this reason, the requirement is assessed as fully satisfied by this solution.

RA5. Sufficient Training Dataset Size

The data collection mechanism proposed by the solution follows an end-to-end automated process. This approach is feasible thanks to the possibility to access the DOM of WUIs, which can bring valuable information for the different labeling tasks. This technique allows acquiring a dataset with a sufficient size in much less time when comparing it to a manual technique.

The dataset acquired consists of 23000 training examples containing around 808983 UI objects. This value surpasses by far the numbers registered by the PascalVOC dataset, and for such reason, this technique fully satisfies the requirement of a sufficient dataset in size.

Nevertheless, it is necessary to mention that following this approach introduces some noise in the data given by incorrect auto-generated labels from websites on the internet. It is a limitation introduced by the different ways in which website interfaces can be built, but it is a price worth to pay for collecting a dataset with more variability.

RA6. Sufficient Training Dataset Frequency of Distribution

The data collection mechanism proposed by the solution includes several features that allow fulfilling this requirement. The WUI Analyzer supports an execution configuration that allows specifying a set of required elements in user interfaces for further labeling. It also counts with a component that exports statistical data about the auto-generated ground truths. Additionally, the solution includes another tool that reads the statistical information exported by the WUI Analyzer and shows a visual representation of such statistics. Having this visual representation easies the process of checking the frequency of distributions per-class. Such considerations are enough for fully satisfying the requirement.

RA7. Training Dataset Collection Efficiency

The data collection mechanism proposed by the solution is comprised of a set of tools that facilitate the whole process. With the tools in place, labeling a single UI frame takes around 2 seconds, which is much faster compared to a crowdsourcing approach. Additionally, the automated nature of this mechanism requires much less human efforts. For such reason, this technique fully satisfies the requirement in mention.

RA8. Avoid Re-training from Scratch

The followed approach enables different object detection models to train progressively in a straightforward manner because it provides an architecture that includes on one hand, a dataset service capable of storing training examples, and on the other hand, a set of object detectors that fetch training examples from this dataset service and subsequently trigger the training process when they count with a batch sufficient in size (threshold configurations can be changed as desired). When the training process finishes, it exports a model from which further training processes can start. That is why the requirement is considered as fully satisfied.

RA9. High Precision

Regarding precision, the followed approach is highly dependent on the object-detection mechanisms reused. The evaluation is measured in terms of mAP and per-class APs metrics which unify precision and recall rates in a single value. Being a 100% mAP score the maximum score reachable, it can be appreciated in Figure 6.3 that both Faster-RCNN and RetinaNet produce scores below the middle. Analyzing why this can happen, it is concluded that there are two main factors:

- Inter-class similarity: There are several cases in which the detector can be confused by objects of different classes given their shared similarities. This confusion can lead to a double penalization in the score, since deciding the wrong class affects the AP values of the two classes involved.
- Prediction vs. Ground-truth IoU: Another case that can contribute negatively to the metric score is the IoU between the bounding boxes predicted by the model and the ones from the ground-truth. The evaluation considers an IoU threshold of 0.5, which is the default value used for computing the PascalVOC metrics. This value demands bounding boxes from predictions vs. ground-truths with a low error margin; if a prediction contains a correct class but its

bounding box does not match the one of the ground-truth sufficiently, it is considered as a false positive, downgrading the precision of the detector.

It can also be appreciated that Faster-RCNN outperforms RetinaNet in almost every class, leaving it as the most suitable object detection implementation for the domain encompassed by this thesis. Considering mAP score of 80% as a sufficient value, Faster-RCNN is only able to achieve half of the score, meaning it only partially satisfies the requirement.

RA10. High Recall

The evaluation of recall is measured in terms of mAP and per-class APs metrics. It is relevant to mention that recall values have the same influence as precision values on the final per-class AP scores since it represents the average of the all the possible precision vs. recall rates in the model for a class. Having a low recall will impact negatively on the AP score.

An additional cause for a low recall can be wrong objectness score predictions made by the object detectors. An incorrect prediction on the objectness score can be traduced in a potential object being filtered out by a predefined confidence threshold in a post-processing stage of the detection.

By having the same considerations for assessing this requirement as with precision, it is rated as partially satisfied.

	GA1.1. Two-Stage	**GA1.2. One-Stage**	**Solution**
RA1. UI Parts	++	++	++
RA2. Process Automation	+	+	++
RA3. Visual Input	++	++	++
RA4. Deep Learning	++	++	++
RA8. Avoid Re-training	+	+	++
RA9. High Precision	-	-	+
RA10. High Recall	-	-	+

++ fully satisfied, + partially satisfied, - not satisfied

Table 6.3. Requirements Assessment - Object Detection - UI Object Detection Using DNNs

	GA2.1. Existing	**GA2.2. Manual**	**GA2.3. Auto**	**Solution**
RA5. DS Size	-	++	++	++
RA6. DS Freq Dist	-	++	++	++
RA7. DS Coll Eff	++	-	++	++

++ fully satisfied, + partially satisfied, - not satisfied

Table 6.4. Requirements Assessment - Dataset Collection - UI Object Detection Using DNNs

Hypothesis A

Regarding the Hypothesis A, it becomes difficult to be tested, since until now there is no appropriate data to which the results in this thesis can be compared. The closest way to comparing the performance of object detection strategies across the object detection domain of this thesis and a general object detection domain is grabbing the mAP scores of similar detection techniques in datasets such as PascalVOC and making a comparison with the mAP scores of the detectors provided by this solution on the dataset collected here. However this is not the fairest methodology, because there can be several variables introduced by the dataset that can end up affecting the mAP score of the detectors. Some dataset can contain many more classes than others, and the particularities introduced by each class in the dataset can also be crucial. Particularly speaking about the PascalVOC and the dataset collected here, they have similar sizes according classes. However, One can notice that the dataset collected here has much less inter-class variability, causing a serious dropdown to the metrics. For such reasons, it is considered unfeasible comparing object detection techniques trained and evaluated over different datasets.

A much more feasible approach, could be to count on existing metrics of other techniques different to deep learning on a common dataset that addresses the domain of this thesis. However there are no public datasets nor official metric registries of other techniques across compatible datasets that can be used as a basis. Such fact ends up limiting the test scenario of the hypothesis.

6.2. Combining UI Object Detection Analyzers

Part written by Victor Flamenco

This section evaluates the implementation of the Thulium Platform to assess whether it satisfies its purpose and to test Hypothesis B ultimately.

6.2.1. Setup

The main objective pursued in this setup is to iteratively improve the quality of detection results provided by external UI object detection tools. Setting this evaluation up in the Platform demands the experimentation with all its features, and consequently, by testing hypothesis B, the Platform's concept is implicitly evaluated. The testing dataset consists of 75 randomly-selected and labeled UI screenshots with sufficient heterogeneity. As a first step, the dataset is uploaded utilizing the *Datasets Management* interface.

Two available vision-based end-to-end UI object detection tools are considered: a Faster-RCNN and a RetinaNet implementations, both drafted in subsection 4.2.3 and implemented as described in section 5.1. The evaluation procedure consists of two parts. First, testing *isolated tools*, and second, testing *combined tools*. Results of both parts are lastly contrasted.

Tools Isolation

Both detectors are registered at the External Tools Gateway, via the *ToolsController*, and subsequently, one process definition is modeled per detector, using the following configuration:

- Technique: Vision-based
- Strategy: End-to-end
- Weight: 1.0
- Class Labels Filter: all checked (see Table B.1)

Since both detectors are end-to-end implementations, each process instance implementation has one single step, and both share the same initial and end sub-tasks: *end-to-end implementation.* Moreover, since the goal is tool isolation, the conflict resolution external tool is omitted in this part.

Naturally, the next step is running the Evaluations Controller with both definitions and the testing dataset as inputs. This evaluation maintains a constant IoU threshold of 0.5, similar to the PASCAL VOC Challenge [28]. The evaluation's metrics (AP and mAP) are extracted. Towards evaluating conflict resolution, measurements to include, per detector, are the total number of detected elements and the number of detected elements per object type. These measurements serve as results and are compared to the results of the next experiment: tools combination.

Tools Combination

Modeling a new process definition follows. It uses both detectors combined, and must, therefore, contain two implementations, one per detector. Each implementation is initially built in the following manner:

- Technique: Vision-based
- Strategy: End-to-end
- Weight: 1.0
- Class Labels Filter: all checked.

The evaluation is executed in **three iterations**. The first two iterations serve as a basis to adjust the definition's parameters (weights and class labels filters), as well as the conflict-resolution parameters (bounding-box conflicts IoU, classification-conflicts IoU, and confidence penalization σ). The evaluation's output metrics, i.e., AP and mAP, are analyzed after each iteration and used as a basis to interpret for which class labels each implementation provides better results, in order to adjust conflict-resolution and the definition's parametrizations in the next iteration. Finally, the resulting metrics from the third iteration are considered as final.

Besides, conflict resolution is measured and analyzed by comparing the total number of detected objects, as well as the number of detections per object type, before and after results unification. The conflict-resolution parameters setup is determined by this work's experimentation, leading to identify the values that provide the most consistent conflict-resolution outputs.

6.2.2. Results

Similarly to the previous subsection, this one presents and analyzes first, results for tools isolation, and afterward, results for tools combination.

Tools Isolation

Details on the evaluation metrics for tools isolation are exposed in section 6.1.2. Therefore, this subsection focuses on measurements that facilitate the evaluation of conflict resolution. Table 6.7 summarizes the number of detected UI objects per detector and per object type. The second and third columns show the number of objects detected in the testing dataset by **isolated tools**; the fourth column aggregates these numbers. This subsection compares later these numbers against the number of detected objects by the combination of tools.

Object Type	**Faster-RCNN**	**RetinaNet**	**Total Object Type**
link	1,332	2,360	3,692
button	477	1,107	1,584
symbol	460	1,207	1,667
panel	234	326	560
image	202	619	821
textinput	108	248	356
label	87	327	414
list	55	169	224
checkbox	51	111	162
radiobutton	39	44	83
selectbox	37	120	157
table	21	89	110
passwordfield	16	71	87
tabs	12	28	40
textarea	10	26	36
pagination	6	15	21
datepicker	4	11	15
Total Detector	**3,151**	**6,878**	**10,029**

Table 6.5. Tools Isolation - Number of Detected Objects

Tools Combination

Results of the three iterations concerning tools combination are exposed here. The histogram in Figure 6.8 reveals the progressive improvement in metrics through each iteration. Table 6.6 details the AP per object type and mAP scores for the three evaluation iterations.

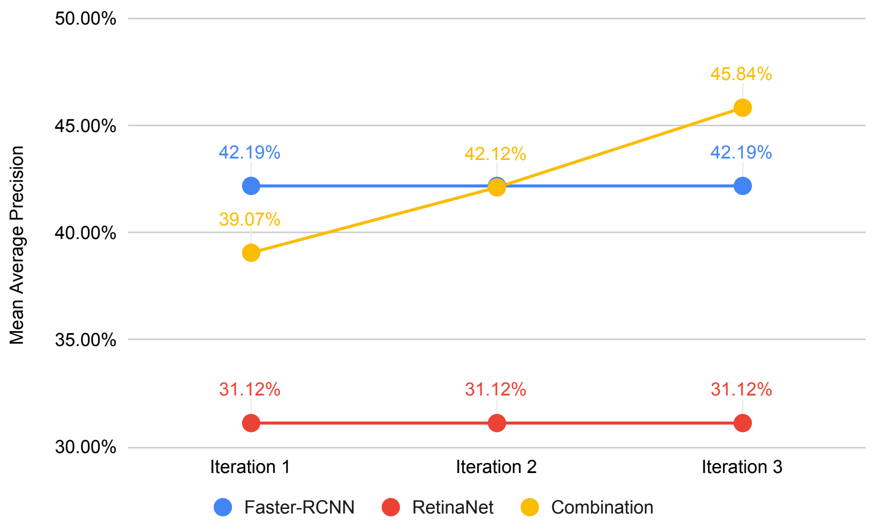


Figure 6.8. Iterations Results - Mean Average Precision

Iteration 1

In pursuit of an initial estimation of the most suitable parameter values, this iteration assigns both Faster-RCNN and RetinaNet implementations a weight of 1.00; for the sake of an objective conflict-resolution evaluation, this iteration filters no class labels from either implementation.

The following list presents the four **parameter values of Iteration 1**. Conflict-resolution is intended to be more sensitive at identifying bounding-box conflicts, and less sensitive at classification instances. Experimentation shows that IoU threshold values of 0.20 and 0.50, respectively, are convenient for obtaining a good initial sense of the behavior of the Thulium Soft-NMS conflict-resolution algorithm. This iteration replicates the confidence penalization value of 0.50 used in [8] to feed the Gaussian weighting function.

- Bounding-box IoU threshold: 0.20
- Classification IoU threshold: 0.50
- Confidence penalization: 0.50
- Confidence threshold: 0.10

The above configuration provides a mAP value 7.95% higher than RetinaNet isolated. Nevertheless, this result is 3.12% lower than the value of Faster-RCNN isolated. Consequently, adjustments are required.

Based on AP values per object type, Faster-RCNN is better than RetinaNet at identifying particularly *radiobuttons*, *paginations*, *passwordfields*, and *tabs*. On the contrary, RetinaNet excels at detecting *datepickers*. Overall, the former tool provides higher scores at 16 out of the 17 identified object types, leading to the necessity of reducing the latter tool's weight in the process definition. Furthermore, class labels filtering is adjusted to enhance these results in the upcoming iteration.

Object Type	**Iteration 1**	**Iteration 2**	**Iteration 3**
link	59.17%	60.99%	63.60%
panel	58.06%	60.10%	61.92%
checkbox	53.05%	56.25%	61.80%
textinput	52.95%	56.10%	61.20%
button	52.04%	54.31%	57.14%
image	49.60%	54.17%	56.25%
label	47.28%	51.17%	54.07%
radiobutton	45.29%	47.70%	50.00%
list	42.85%	46.87%	49.29%
textarea	40.00%	42.75%	48.30%
symbol	39.05%	40.00%	44.43%
pagination	37.50%	39.61%	40.65%
datepicker	25.00%	37.50%	40.00%
table	19.05%	24.17%	29.58%
selectbox	18.25%	19.40%	25.98%
passwordfield	12.50%	12.50%	22.50%
tabs	12.50%	12.50%	12.50%
Mean Average Precision	**39.07%**	**42.12%**	**45.84%**

Table 6.6. Iterations Results - Average Precisions and Mean Average Precision

Iteration 2

This iteration reduces RetinaNet's weight in the process definition, as the previous iteration shows that this tool provides lower AP scores for most object types, and a lower mAP score when compared to Faster-RCNN. Class labels filtering is additionally adjusted to take advantage of the results from the previous iteration.

The following list presents the **adjusted parameter values of Iteration 2**. The definition configuration suppresses the object types for each implementation, where the tool's results are not as good as the results of the other tool. Adjustments to the conflict-resolution parameters are necessary for two reasons. First, to stronger penalize lower-confidence detections overlapping with the higher-confidence detections, hence the penalization parameter is increased by 0.20. Second, to reduce the number of suppressed detections with a confidence score lower than the confidence threshold, which is potentially correct, thus the confidence threshold is reduced to the half.

- Faster-RCNN weight: 1.00
- Faster-RCNN filtered objects: datepicker
- RetinaNet weight: 0.80
- RetinaNet filtered objects: passwordfield, tabs, pagination, radiobutton
- Confidence penalization: 0.70
- Confidence threshold: 0.05

The resulting mAP value of 42.12% in this iteration shows an improvement regarding the previous iteration, however the score is still slightly below Faster-RCNN. Besides, two key aspects are identified. First, by reducing the weight of RetinaNet's implementation, the output misses several correct predictions from this detector. Second, a revision of the resulting conflict resolution outputs helps to identify that there are still several resolutions that can be further improved and that a relevant number of correct detections are suppressed. The next iteration aims at optimizing these parametrizations.

Iteration 3

The following list presents the **adjusted parameter values of Iteration 3**. For optimizing conflict resolution, the confidence penalization is increased to its maximum, as the experimentation repeatedly indicates that this value throws the most accurate resolutions. Moreover, this iteration minimizes the confidence threshold value to reduce the number of omitted detections compared to the previous iteration. As the last adjustment, the weight of RetinaNet's implementation is restored to its initial value in order to restore the correct predictions suppressed from this detector in the second iteration.

- Confidence penalization: 1.0

- Confidence threshold: 0.001
- RetinaNet weight: 1.0

The resulting mAP value of 45.84% in this iteration represents an improvement of 3.72% over the previous iteration, as well as, an improvement of 3.65% and 14.72% against Faster-RCNN and RetinaNet, respectively. These results are considered final.

Conflict Resolution

Comparing the number of objects detected by isolated tools to the number of detected objects after combining them and applying results unification using Thulium Soft-NMS, is a key step towards understanding the improvement in the evaluation metrics. Table 6.7 summarizes the comparison. The second column presents the sum of objects that Faster-RCNN and RetinaNet identify isolatedly; the third column represents the number of detected objects by the combination of both tools; and lastly, the fourth column depicts the percentage of reduction in the detected objects that conflict resolution achieves.

Object Type	**Total Isolated**	**Total Combined**	**Reduction (%)**
link	3,692	3,590	2.76%
button	1,584	1,442	8.96%
symbol	1,667	1,596	4.26%
panel	560	530	5.36%
image	821	729	11.21%
textinput	356	299	16.01%
label	414	390	5.80%
list	224	206	8.04%
checkbox	162	162	0.00%
radiobutton	83	39	53.01%
selectbox	157	108	31.21%
table	110	94	14.55%
passwordfield	87	16	81.61%
tabs	40	11	72.50%
textarea	36	24	33.33%
pagination	21	6	71.43%
datepicker	15	9	40.00%
Total Analyzer	**10,029**	**9,251**	**7.76%**

Table 6.7. Reduction of Detected Objects With Thulium Soft-NMS

6.2.3. Discourse

This subsection is organized by process- and results-related requirements defined in section 2.2. The compliance of the Thulium Platform implementation against each requirement is described in this direction. Table 6.8 summarizes the evaluation and compares the proposed implementation in this thesis with the SotA groups of approaches.

Process-related Requirements

The analysis of the implemented integration platform against process-related requirements follows.

RB1. Integration Platform

The Thulium Platform offers a common infrastructure for reusing distributed object detection tools. In its multi-tier architecture, platform layers perform various tasks and provide services to other layers. Inter-layer communication is achieved through well-defined APIs. Thulium is categorized as an *Integration Platform* since it virtually joins a set of distributed services into a shared infrastructure.

A key platform feature is *configurability* because external tools have inherently plug-and-play nature, and become dynamically accessible via defined interfaces. Thulium is a suitable integration platform because of its ability to connect distributed software components in a configurable fashion. Thus, the requirement is *fully satisfied*.

RB2. Supporting a Wide Variety of Tools

Although this work is scoped to parser- and vision-based techniques, the platform is not limited to them. Object detection techniques and their strategies are configurable thanks to the Thulium UI object detection reference process, a workflow that defines the arrangement of tools execution. A variety of tools is supported: (a) end-to-end tools, executing complete object detection virtually in a single step using, and (b) divisible implementations, consisting of several sequential steps. The External Tools Gateway provides mechanisms to configure external tools and their capabilities.

The Process Definition artifact facilitates configurable isolation and combination of tools via so-called *implementations*, enabling parallel execution of various algorithms.

Implementation Steps realize the execution of sequential sub-tasks. Criterion No. 1 is positive since the platform organizes external tools at *definition time*.

Moreover, Thulium organizes the execution of external tools through the Runtime Engine. It triggers instances of process definitions, where the arrangement of sub-tasks execution occurs. Interactions with the External Tools Gateway enable the establishment of communication with object-detection and conflict-resolution tools through standard interfaces. Therefore, the platform complies with criterion No. 2, as the organization of software components at *runtime* is possible.

Thulium *supports* a wide variety of object detection tools as it complies with both of these evaluation criteria. The platform consequently *fully satisfies* this requirement.

RB3. Results Aggregation

The Runtime Engine processes instance implementations parallelly. Once all instances end, a subsequent aggregation of results takes place. Results of diverse end-to-end and divisible implementations are aggregated into one single system output with the ILabeledConfSet interface, containing RoIs, class labels, and confidence scores. Results aggregation is achieved by resolving inconsistencies in the outputs of implementations and merging their outputs into a single process instance output. The derived requirement RB4 discusses more in this direction.

Besides, a crucial aspect for aggregating results is counting with uniform input and output formats for UI object detection implementations. The derived requirement RB6 analyzes this further. Thulium *partially satisfies* this requirement because it only fully satisfies two out of its three derived requirements; more details are included in each requirement.

RB4. Conflict Resolution

The Conflict Resolution component receives a series of *weighted* ILabeledConfSet outputs from diverse object detection implementations, and returns a single ILabeledConfSet artifact. This output is intended to be a clean and ideally conflict-free UI object detection result. The component defines conditions to identify conflicts in either of two dimensions: *bounding box* and *classification*, and an algorithm that solves instances of both.

The Thulium Soft-NMS conflict-resolution algorithm proved to be crucial for improving metrics. It managed to reduce by 7.76% the total number of detected objects by a combination of tools to 9,251 when compared to the outputs of isolated tools, where

10,029 objects were detected. Such performance leads to significant improvements in Average Precision values, for example:

- Concerning *radiobutton* objects, reducing outputs by 53.01% leads to an enhancement to the best isolated AP score from 47.89% to 50.00%
- Solving conflicts for *tabs* objects by reducing the number of detections by 72.50% caused the best isolated AP score to double from 6.25% to 12.50%.
- Reducing the detected *table* objects by 14.55%, it managed to increase the best isolated AP score by 8,75%.

As Thulium solves both of the studied conflict dimensions, it *fully satisfies* this requirement.

RB5. Weight Distribution Analysis

In addition to assisting in the generation of tool combinations, datasets management, and the automatic testing of process definitions against entire datasets, the Controller enables iterative improvement of detection metrics. One can compute an initial set of metrics, analyze the results, tune conflict-resolution and process-definition parameters, and subsequently re-run the metrics computation. The iterative evaluation procedure followed in subsection 6.2.1 and subsection 6.2.2 shows the practical potential of the Evaluations Controller.

AP is computed per object class. For comparing performance overall object classes, mAP is adopted as the final measure of performance. With these metrics, one can infer valuable hints and information about what can be the most suitable parametrization for upcoming object detection processes in order to improve detection quality.

Nevertheless, experimentation in this work aided to identify that penalizing the entire set of detected object types with a shared weight leads to suboptimal results. One alternative to enhance this is using object-type-specific weights, maximizing the flexibility in evaluations and conflict-resolution mechanisms. In this direction, improvements are still required, Thulium *partially satisfies* this requirement.

RB6. Uniform Communication Interfaces

This work concretizes a complete set of uniform communication interfaces considering two aspects: *representation formats* and *messaging protocols*. The former aspect is described by the Thulium UI Object Detection Reference Process through *Interfaces*, standard representation formats of all possible input, output, and intermediate artifacts for generically describing UI object detection processes. The reference process guarantees tools interoperability, and therefore, the platform to complies with criterion No. 1.

The latter aspect is described by the External Tools Gateway and the Runtime Engine. The Gateway is responsible for requesting tools execution using lightweight and standard protocols; while the Engine accepts tool responses in the same fashion. Counting with suitable messaging protocols enables Thulium to comply with criterion No. 2, and in consequence, to *fully satisfy* this requirement.

	GB1. Service	**GB2. Agent**	**GB3. Workflow**	**Thulium**
RB1. Platform	++	+	++	++
RB2. Variety	+	+	++	++
RB3. Aggregation	-	+	+	+
RB4. Conflicts	+	+	+	++
RB5. Weights	+	+	+	+
RB6. Uniformity	++	++	+	++

++ fully satisfied, + partially satisfied, - not satisfied

Table 6.8. Evaluation - Process-Related Requirements - Combining UI Object Detection Analyzers

Results-related Requirements

This category analyzes the implemented solution against results-related requirements.

RB7. Enhancement of Detection Quality

The Thulium Platform facilitates testing whether the UI object detection quality results improve with the combination of techniques when compared to the quality of detection results of isolated techniques. This evaluation uses a randomly-selected testing dataset to assess Faster-RCNN and RetinaNet isolatedly and the combination of both.

Combining the object detectors managed, iteratively, to enhance the AP value of almost all detected object types -a few more significantly than others-, and to increase the mAP score to 45.84% when compared to isolated Faster-RCNN (42.19%) and RetinaNet (31.12%). At the end of the three iterations using combined tools, the final detection metrics are better than the evaluation metrics of both isolated tools. Therefore this requirement is *fully satisfied.*

Hypothesis B

One can think that since the UI object detection quality results obtained with the combination of techniques is better than the quality results of isolated techniques, Hypothesis B is positive. Nevertheless, one must take into consideration the evaluated dataset size and its domain, including newspapers, wikis, educational, non-profit organizations, and form-based data collection sites. The random example selection lacks blogging, online business, e-commerce, personal, social media, and other domains; a limited domain selection restricts the generalization of results.

Moreover, the object types distribution in the testing dataset can be still improved; in this evaluation, some object types have considerably lower frequency than others, e.g., *tabs*, *datepicker*, *pagination*, *passwordfield.* Consequently, the AP scores for these object types have a low degree of flexibility; their scores can change abruptly even when the number of TP incidences is increased or decreased by 1.

Concerning evaluation scenarios similar to the one in this work, it is possible to say that the hypothesis is valid. However, a much more extensive evaluation is required in order to generalize this statement.

6.3. Summary

This chapter drives an assessment of the solutions that this work proposes in order to test the two hypotheses. The overall assessment is composed of two sub-assessments that help to evaluate each solution across its related requirements stipulated in section 2.1 and section 2.2, respectively. Each assessment describes an evaluation environment and a methodology with its corresponding results; these results are analyzed in order to test whether the solution parts meet their concerning requirements.

The next chapter draws conclusions that come out of this joint thesis and proposes further research work in this domain.

7. Conclusion

This research produced the concept and implementation of a solution capable of integrating UI object detectors, with a strong emphasis on vision-based SotA techniques, aiming at the improvement of detection metrics provided by combinations of detectors when compared to the results of isolated detectors; this work is composed of two solution parts, where each one addressed a specific objective and tested one particular hypothesis.

On the one hand, the first solution part of this work -UI Object Detection Using DNNs- consisted in applying state-of-the-art object detection techniques on the domain of UIs. After the evaluation of such solutions, one could argue whether their performance is consistent across the two domains, or if it varies in any direction. Towards achieving the goal of this part, two central pillars were identified: data collection and object detection. While the former refers to tasks for acquiring a consistent and useful training dataset, the latter relates to picking up, training, fine-tuning, and evaluating a suitable object detection strategy (or a set of strategies).

The data collection strategy proposed considers the acquisition of a training dataset in an automated way. For such purpose, a set of tools were designed and prototyped accordingly. Following this approach has allowed collecting a considerably large dataset in a short time period. However, it was observed that the resulting dataset also contains noisy data, i.e., mislabeled or imprecise training examples. It happens in most of the cases for websites that do not rely on any of the considered CSS frameworks, and end up being mislabeled by the generic parser. Another aspect noticed is the padding variability on bounding boxes for certain classes on the generated dataset; for instance, in the case of links, the <*a*> tag not always surrounds the text; it can also contain other elements having as a consequence inconsistent bounding boxes.

For object detection, a pair of models have been considered comprising a two-stage and a one-stage detector: Faster-RCNN and RetinaNet, respectively. Regardless of the models, the solution includes the design and the prototype of an object detection wrapper that encapsulates technical-oriented tasks required to train an object detection model. Such encapsulation also helps in lowering the complexity of training an object detection model progressively. Besides that, the wrapper exposes a REST API for inference, enabling easy integration of the object detection model with third-party systems. After evaluating the two models, it has been noticed that one of the most crucial aspects affecting their performance in terms of precision and recall is the lower inter-class variability introduced by some sets of classes. Having very visually similar classes confuses the objected detection model.

On the other hand, the second solution part of this work -Combining UI Object Detection Analyzers- focused on the architecture of the Thulium Platform, a workflow-oriented tool for integration of various UI object detection implementations; it offers a common infrastructure for virtually joining and reusing distributed tools that can be dynamically plugged and become accessible through defined interfaces. Although this work is scoped to parser- and vision-based techniques, the platform is not limited to them. Object detection techniques and their strategies are configurable thanks to the Thulium UI object detection reference process, a workflow that defines the arrangement of tools execution generically. Thulium *supports* a wide variety of object detection tools as it enables arranging components during definition time and coordinating tools execution.

By evaluating, first, the Faster-RCNN and RetinaNet implementations, described in the first solution part, isolatedly and second, a combination of both detectors, the platform was able to prove that the combination improved the AP and mAP scores compared to the scores of isolated tools; this was achieved using an iterative evaluation procedure, where an adjustment of parameters followed each iteration. The Thulium Soft-NMS conflict-resolution algorithm reduced the number of detected objects significantly with inconsistent predictions, resulting in a cleaner and more accurate output.

One can think that since there was an improvement in the detection quality, Hypothesis B -the combination of approaches improves the detection results when compared to the results of individual detectors- is positive. Nevertheless, one must take into consideration a few limitations. First, the evaluated dataset size and its domain, including newspapers, wikis, educational, non-profit organizations, and form-based data collection sites. The random example selection lacks blogging, online business, e-commerce, personal, social media, and other domains; a limited domain selection restricts the generalization of results.

Second, the object types distribution in the testing dataset can be improved; some object types have considerably lower frequency than others, e.g., *tabs*, *datepicker*, *pagination*, *passwordfield*. Consequently, the AP scores for these object types have a low degree of flexibility; their scores can change abruptly even when the number of TP incidences is increased or decreased by 1.

Concerning evaluation scenarios similar to the one in this work, it is possible to say that Hypothesis B is valid. However, a much more extensive evaluation is required in order to generalize this statement.

7.1. Future Work

The above findings lead to the identification of aspects in this work that require improvements and where alternative solutions and a more extensive evaluation can enhance the results even more. Thus, further research in the directions described next is encouraged.

Concerning the dataset collection in the first solution part of this work, the existing noise in the labeled data generated by the existing prototype can be decreased by adjusting the labeling criteria applied by the generic parser in an iterative way; it means making small adjustments to the labeling criteria, testing it on small sets of websites, checking the resulting labels, and re-adjusting if necessary on a new iteration. The labeling performance can also be improved by integrating more CSS framework parsers, and having more dedicated CSS framework parsers increments the chances to have a more accurate labeling process while decreasing the dependency on the generic parser. Additionally, something that can improve the labeling is adjusting the way bounding boxes are computed for specific classes; an initial hint can be identifying those classes consisting mainly of text, such as *label*, *link*, or *list*, and trying to rely on a text-based API to build the bounding boxes (see the DOM Range Specification[1]). An interesting research area in the context of dataset collection can be experimenting with GANs [17]. Finding a way to apply them in this scenario can be valuable since it can be a potential new source for generating labeled data automatically. Besides that, it could also be valuable to design such a platform that unifies and integrates different dataset collection techniques, including crowdsourcing and automated approaches; in this regard, it is crucial to define an unambiguous set of labeling rules that ensures consistency across the different techniques.

In the object detection context, it was observed that merely visual perception is only able to partially satisfy the precision and recall requirements defined in this thesis; this comes as a consequence of having different classes that are visually similar. One approach that can improve this situation is introducing other components in the object detection pipeline that consider *contextual* information of the detected objects. Another logical approach can be to merge certain sets of classes that can cause conflicts in the detection, such as *textinput*, *passwordfield*, and *selectbox*; however, evaluating the trade-off between merging certain classes vs. losing crucial semantic

[1]https://www.w3.org/TR/DOM-Level-2-Traversal-Range/ranges.html

information about the detected objects becomes vital. Another attractive research area is object segmentation, which, instead of predicting bounding boxes it predicts a pixel-wise outline of the objects [38]; following an object detection approach is still very useful in this scenario, considering that most of the UI objects comply with a rectangular shape; nevertheless, with an object segmentation technique, the calculation of more precise area-dependent metrics is possible.

Regarding the second solution part of this work, while experimenting with the Evaluations Controller to iteratively improve the detection metrics by adjusting definition and conflict-resolution parameters, it was identified that penalizing the entire set of detected object types with a shared process definition weight leads to suboptimal results. More precisely, this affects situations in which an object detector performs outstandingly with particular object types and poorly with others. Reducing the detector's weight implies penalizing its best and worst detections equally. One alternative to enhance this is using object-type-specific weights, maximizing the flexibility in evaluations and conflict-resolution mechanisms. With this solution, one can strongly penalize only the object types for which the detector is not optimal, and skipping (or at least, reducing) the penalization to a detector's best predictions.

The Thulium Soft-NMS conflict-resolution algorithm proved to be crucial for improving the detection metrics, reducing the total number of detected objects, resulting in a cleaner and more accurate detection output. Although this component managed to satisfy the requirement that it pursued entirely, by defining conditions to identify conflicts in either of the bounding-box and classification dimensions and proposing resolutions to each, an extension to this algorithm can still enhance its results. Particularly in the domain of UIs, a potential extension can additionally consider the knowledge that arises from previous detection experiments. To illustrate this, one can think of an example conflict in the classification dimension comprising a *search symbol* within a *textinput*, that combined depict a search box. If the IoU ratio of the two bounding boxes labeled with different object types is greater than the threshold IoU value defined for the resolution, the resolution penalizes the box with lower confidence. One can design an additional custom rule-based resolution step, emphasizing on cases where the conflict-resolution parameters limit the potentially correct predictions. Knowing that the algorithm commonly struggles to resolve cases like the one above, adding a rule per possible case can increase the detection quality even more.

As a final note, this work encourages a more extensive evaluation, particularly considering two aspects. First, increasing the testing dataset to a higher number of examples, and second, ensuring that the examples selection includes a higher number of instances per relevant domain of UIs and a higher frequency of examples per object type. A limited domain selection restricted the generalization of results.

Moreover, the AP scores for object types with low frequency had a low degree of flexibility; their scores changed abruptly, even when the number of TP incidences slightly changed. These improvements can lead to a better generalization of results.

References

[1] ISO/IEC/IEEE International Standard - Systems And Software Engineering - Vocabulary. 2017.

[2] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, and C. Mohan. Functionality and Limitations of Current Workflow Management Systems. 1997.

[3] Felix Altenberger and Claus Lenz. A Non-Technical Survey On Deep Convolutional Neural Network Architectures. 2018.

[4] Ali Arsanjani and Combining Structured. Service-Oriented Modeling And Architecture. (December), 2014.

[5] Maxim Bakaev, Sebastian Heil, Vladimir Khvorostov, and Martin Gaedke. Auto-Extraction And Integration Of Metrics For Web User Interfaces. pages 561 – 590, 2018.

[6] Anant Bhardwaj, Amol Deshpande, Aaron J. Elmore, David Karger, Sam Madden, Aditya Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. Collaborative Data Analytics With DataHub. *Proceedings of the VLDB Endowment*, 2015.

[7] Valentin Tertius Bickel, Charis Lanaras, Andrea Manconi, Simon Loew, and Urs Mall. Automated Detection Of Lunar Rockfalls Using A Convolutional Neural Network, 2018.

[8] Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S. Davis. Soft-NMS - Improving Object Detection With One Line Of Code. *Proceedings of the IEEE International Conference on Computer Vision*, 2017-October:5562–5570, 2017.

[9] Léon Bottou and Yann Le Cun. Online Learning For Very Large Datasets. *Applied Stochastic Models in Business and Industry*, 2005.

[10] Daren C. Brabham. Crowdsourcing As A Model For Problem Solving: An Introduction And Cases. *Convergence*, 2008.

[11] Dan Brickley, Matthew Burgess, and Natasha Noy. Google Dataset Search: Building A Search Engine For Datasets In An Open Web Ecosystem. In *The World Wide Web Conference on - WWW '19*, 2019.

[12] Michael Buckland and Fredric Gey. The Relationship Between Recall And Precision. *Journal of the American Society for Information Science*, 1994.

[13] Deng Cai, Shipeng Yu, Ji-rong Wen, and Wei-ying Ma. VIPS: A Vision-Based Page Segmentation Algorithm. pages 1–32, 2013.

[14] Victoria M. Catterson, Euan M. Davidson, and Stephen D.J. McArthur. Practical Applications Of Multi-Agent Systems In Electric Power Systems. *European Transactions on Electrical Power*, 22(2):235–252, 2012.

[15] Deepayan Chakrabarti, Ravi Kumar, and Kunal Punera. A Graph-Theoretic Approach To Webpage Segmentation. 2008.

[16] Cisco and San Jose. Cisco Visual Networking Index (VNI) Global Mobile Data Traffic Forecast Update. 2019.

[17] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative Adversarial Networks: An Overview, 2018.

[18] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: Object Detection Via Region-Based Fully Convolutional Networks. may 2016.

[19] Navneet Dalal and Bill Triggs. Histograms Of Oriented Gradients For Human Detection. In *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, 2005.

[20] Brajesh De. *API Management*, pages 15–28. Apress, Berkeley, CA, 2017.

[21] Sebastian Deterding, Miguel Sicart, Lennart Nacke, Kenton O'Hara, and Dan Dixon. Gamification. Using Game-Design Elements In Non-Gaming Contexts. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems - CHI EA '11*, 2011.

[22] James J DiCarlo, Davide Zoccolan, and Nicole C Rust. How Does The Brain Solve Visual Object Recognition? *Neuron*, 2012.

[23] Alan Dix. Human–Computer Interaction: A Stable Discipline, A Nascent Science, And The Growth Of The Long Tail. *Interacting with Computers*, 22(1):13–27, 12 2009.

[24] Arthur De M Del Esposte, Fabio Kon, Fabio M Costa, and Nelson Lago. InterSCity: A Scalable Microservice-Based Open Source Platform For Smart Cities. (Smartgreens):978–989, 2017.

[25] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces Of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[26] David S Evans, Andrei Hagiu, and Richard Schmalensee. A Survey Of The Economic Role Of Software Platforms In Computer-Based Industries. 51(July 2004):189–224, 2005.

[27] Mark Everingham, S. M.Ali Eslami, Luc Van Gool, Christopher K.I. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision*, 2014.

[28] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[29] Kent Gauen, Ryan Dailey, John Laiman, Yuxiang Zi, Nirmal Asokan, Yung Hsiang Lu, George K. Thiruvathukal, Mei Ling Shyu, and Shu Ching Chen. Comparison Of Visual Datasets For Machine Learning. In *Proceedings - 2017 IEEE International Conference on Information Reuse and Integration, IRI 2017*, 2017.

[30] Yaser Ghanam, Frank Maurer, and Pekka Abrahamsson. Making The Leap To A Software Platform Strategy: Issues And Challenges. *Information and Software Technology*, 54(9):968–984, 2012.

[31] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

[32] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich Feature Hierarchies For Accurate Object Detection And Semantic Segmentation. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014.

[33] Hector Gonzalez, Alon Y.Halevy, Christian S.Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, Warren Shen, and Jonathan Goldberg-Kidon. Google Fusion Tables: Data Management, Integration And Collaboration In The Cloud. *Proceedings of the 1st ACM symposium on Cloud computing.*, 2010.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[35] Tom Gross, Tareg Egla, and Nicolai Marquardt. A Service-Oriented Platform For Developing Sensor-Based Infrastructures. (April), 2006.

[36] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Managing Google's Data Lake: An Overview Of The Goods System. *IEEE Data Engineering Bulletin*, 2016.

[37] He Hao. What Is Service-Oriented Architecture. pages 1–5, 2003.

[38] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, 2017.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial Pyramid Pooling In Deep Convolutional Networks For Visual Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2015.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning For Image Recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016.

[41] Aaron Helsinger and Todd Wright. Cougaar: A Robust Configurable Multi-Agent Platform. *2005 IEEE Aerospace Conference*, pages 1–10, 2005.

[42] Kerrie Holley and Edward M Tuggle. Migrating To A Service-Oriented Architecture. (April), 2004.

[43] David Hollingsworth. Workflow Management Coalition - The Workflow Reference Model. (1):1–55, 1995.

[44] Jeff Howe. The Rise Of Crowdsourcing. *Wired Magazine*, 2006.

[45] George Hripcsak and Adam S. Rothschild. Agreement, The F-measure, And Reliability In Information Retrieval. *Journal of the American Medical Informatics Association*, 2005.

[46] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/Accuracy Trade-Offs For Modern Convolutional Object Detectors. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017.

[47] Nancy Ide, Collin Baker, Christiane Fellbaum, Charles Fillmore, and Rebecca Passonneau. MASC: The Manually Annotated Sub-Corpus Of American English. In *Proceedings of the 6th International Conference on Language Resources and Evaluation, LREC 2008*, 2008.

[48] Systems And Software Engineering - Vocabulary. Standard, International Organization for Standardization, Geneva, CH, 2017.

[49] N. Jennings, K. Sycara, and M. Wooldridge. A Roadmap Of Agent Research And Development In Agents And Multiagents Systems. 306:275–306, 1998.

[50] Nicholas R. Jennings. Agent-Oriented Software Engineering. In Francisco J. Garijo and Magnus Boman, editors, *Multi-Agent System Engineering*, pages 1–7, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[51] Mei Kobayashi and Koichi Takeda. Information Retrieval On The Web. *ACM Computing Surveys*, 2000.

[52] Jun Kong, Omer Barkol, Ruth Bergman, Ayelet Pnueli, Sagi Schein, Kang Zhang, and Chunying Zhao. Web Interface Interpretation Using Graph Grammars. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, 42(4):590–602, 2012.

[53] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. Bricolage: Example-Based Retargeting For Web Design. *Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11*, page 2197, 2011.

[54] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, and Vittorio Ferrari. The Open Images Dataset V4: Unified Image Classification, Object Detection, And Visual Relationship Detection At Scale. pages 1–20, 2018.

[55] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature Methods*, 13(1):35, 2015.

[56] Tsung Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature Pyramid Networks For Object Detection. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017.

[57] Tsung Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal Loss For Dense Object Detection. In *Proceedings of the IEEE International*

Conference on Computer Vision, volume 2017-Octob, pages 2999–3007. Institute of Electrical and Electronics Engineers Inc., dec 2017.

[58] Tsung Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common Objects In Context. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8693 LNCS, pages 740–755. Springer Verlag, 2014.

[59] Jiamin Liu, David Wang, Le Lu, Zhuoshi Wei, Lauren Kim, Evrim B. Turkbey, Berkman Sahiner, Nicholas A. Petrick, and Ronald M. Summers. Detection And Diagnosis Of Colitis On Computed Tomography Using Deep Convolutional Neural Networks. *Medical Physics*, 2017.

[60] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Xinwang Liu, and Matti Pietikäinen. Deep Learning For Generic Object Detection: A Survey. 2018.

[61] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. SSD: Single-Shot Multibox Detector. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[62] D.G. Lowe. Object Recognition From Local Scale-Invariant Features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 1999.

[63] Carol M Lushbough, Douglas M Jennewein, and Volker P Brendel. The BioExtract Server: A Web-Based Bioinformatic Workflow Platform. 39(May):528–532, 2011.

[64] Jason Mahdjoub, Zahia Guessoum, Fabien Michel, and Michel Herbin. A Multi-Agent Approach For The Edge Detection In Image Processings. *CEUR Workshop Proceedings*, 223, 2006.

[65] Jonathan Mallinson, Rico Sennrich, and Mirella Lapata. Paraphrasing Revisited With Neural Machine Translation. 2017.

[66] Pierre Marcenac and Sylvain Giroux. GEAMAS: A Generic Architecture For Agent-Oriented Simulations Of Complex Processes. *Applied Intelligence*, 8(3):247–267, May 1998.

[67] Agnieszka Mikołajczyk and Michał Grochowski. Data augmentation For Improving Deep Learning In Image Classification Problem. *2018 International Interdisciplinary PhD Workshop, IIPhDW 2018*, pages 117–122, 2018.

[68] Aliaksei Miniukovich and Antonella De Angeli. Quantification Of Interface Visual Complexity. pages 153–160, 2014.

[69] Aliaksei Miniukovich, Simone Sulpizio, and Antonella De Angeli. Visual Complexity Of Graphical User Interfaces. pages 1–9, 2018.

[70] R. K. Swain N. K. Kamila. Role-Based Architecture For Complex Agents. 2013.

[71] Samuel Newman. *Building Microservices At Squarespace*. O'Reilly Media, 2015.

[72] Jakob Nielsen and Hoa Loranger. *Prioritizing Web Usability*. New Riders Publishing, Thousand Oaks, CA, USA, 2006.

[73] A Omicini, T Payne, P Mcburney, M P Locatelli, G Vizzari, S Paurobally, V Tamma, M Wooldrdidge, S Poslad, and J Mylopoulos. ACM Transactions On Autonomous And Adaptive Systems Specifying Protocols For Multi-Agent Systems Interaction. 2(4), 2007.

[74] Antti Oulasvirta, Samuli De Pascale, Janin Koch, Thomas Langerak, Jussi Jokinen, Kashyap Todi, Markku Laine, Manoj Kristhombuge, Yuxi Zhu, Aliaksei Miniukovich, Gregorio Palmas, and Tino Weinkauf. Aalto Interface Metrics (AIM): A Service And Codebase For Computational GUI Evaluation. In *The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings*, UIST '18 Adjunct, pages 16–19, New York, NY, USA, 2018. ACM.

[75] M Pajusalu and Torres Rui. The Evaluation Of User Interface Aesthetics. Masters. page 74, 2012.

[76] Randall Perrey and Mark Lycett. Service-Oriented Architecture. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, SAINT-W '03, pages 116–, Washington, DC, USA, 2003. IEEE Computer Society.

[77] Helen Petrie and Nigel Bevan. The Evaluation Of Accessibility, Usability, And User Experience. 2010.

[78] Helmut Petritsch. Service-Oriented Architecture vs. Component-Based Architecture. 2006.

[79] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016.

[80] Joseph Redmon and Ali Farhadi. YOLO9000: Better, Faster, Stronger. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017.

[81] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. apr 2018.

[82] Katharina Reinecke, Tom Yeh, Luke Miratrix, Rahmatri Mardiko, Yuechen Zhao, Jenny Liu, and Krzysztof Z. Gajos. Predicting Users' First Impressions Of Website Aesthetics With A Quantification Of Perceived Visual Complexity And Colorfulness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, 2013.

[83] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection With Region Proposal Networks. *IEEE transactions on pattern analysis and machine intelligence*, 2017.

[84] Ruth Rosenholtz, Yuanzhen Li, and Lisa Nakano. Measuring Visual Clutter. *Journal of Vision*, 2007.

[85] Andrés Sanoja and Stéphane Gançarski. Web Page Segmentation Evaluation. pages 753–760, 2015.

[86] Andrés Sanoja and Stéphane Gançarski. Block-Based Migration From HTML4 Standard To HTML5 Standard In The Context Of Web Archives. 3(1):23–37, 2016.

[87] Andrés Sanoja and Stéphane Gançarski. Migrating Web Archives From HTML4 To HTML5: A Block-Based Approach And Its Evaluation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.

[88] Rainer Schmidt, Axel Kieninger, R Fischer, and Christian Zirpins. Meta-Services – Towards Symmetric Service-Oriented Business Ecosystems. volume 530, 01 2009.

[89] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. OverFeat: Integrated Recognition, Localization And Detection Using Convolutional Networks. 2013.

[90] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A Survey On Deep Transfer Learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11141 LNCS:270–279, 2018.

[91] Vlad Trifa, Dominique Guinard, Vlatko Davidovski, Andreas Kamilaris, and Ivan Delchev. Web Messaging For Open And Scalable Distributed Sensing Applications. In Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, editors, *Web Engineering*, pages 129–143, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[92] Rodrigo Verschae and Javier Ruiz-del Solar. Object Detection: Current And Future Directions. *Frontiers in Robotics and AI*, 2015.

[93] P. Viola and M. Jones. Rapid Object Detection Using A Boosted Cascade Of Simple Features. pages I–511–I–518, 2005.

[94] Xiaoliang Wang, Peng Cheng, Xinchuan Liu, and Benedict Uzochukwu. Focal Loss Dense Detector For Vehicle Surveillance. In *2018 International Conference on Intelligent Systems and Computer Vision, ISCV 2018*, 2018.

[95] Jing Xie and Chen-Ching Liu. Multi-Agent Systems And Their Applications. *Journal of International Council on Electrical Engineering*, 7(1):188–197, 2017.

[96] D F Yates and N Malevris. JJ-Paths And Structured Path Testing: A Redress. *Online*, (November 2008):199–213, 2009.

[97] Mathieu Zen and Jean Vanderdonckt. Towards An Evaluation Of Graphical User Interfaces Aesthetics Based On Metrics. *Proceedings - International Conference on Research Challenges in Information Science*, pages 1–12, 2014.

[98] Yanhong Zhai and Bing Liu. Web Data Extraction Based On Partial Tree Alignment. 2005.

[99] Yumeng Zhang, Yanchao Ma, and Fuquan Zhang. Application Of R-FCN Algorithm In Machine Visual Solutions On Tensorflow Based. In Pavel Krömer, Hong Zhang, Yongquan Liang, and Jeng-Shyang Pan, editors, *Proceedings of the Fifth Euro-China Conference on Intelligent Data Analysis and Applications*, pages 359–366, Cham, 2019. Springer International Publishing.

[100] Xiaotong Zhao, Wei Li, Yifan Zhang, T. Aaron Gulliver, Shuo Chang, and Zhiyong Feng. A Faster RCNN-Based Pedestrian Detection System. In *IEEE Vehicular Technology Conference*, 2017.

[101] Delong Zhu, Tingguang Li, Danny Ho, Tong Zhou, and Max Q.H. Meng. A Novel OCR-RCNN For Elevator Button Recognition. In *IEEE International Conference on Intelligent Robots and Systems*, 2018.

Acronyms

ACL Agent Communication Language. 57–59, 67

AI Artificial Intelligence. 110

AIM Aalto Interface Metrics. 2

AMQP Advanced Message Queuing Protocol. 55

ANN Artificial Neural Network. 18

AP Average Precision. 120, 125, 147, 148, 154–157, 161, 162, 165, 166, 168, 173–175, 178, 181, 195

API Application Programming Interface. 26, 47, 54, 61, 64–66, 68, 75, 79, 81, 84, 85, 93, 128–130, 132, 139, 153, 159, 171, 177, 179

BoM Block-o-Matic. 4, 106

BPEL Business Process Execution Language. 60

BPMN Business Process and Model Notation. 60, 61, 67

BPMS Business Process Management System. 61

CLI Command Line Interface. 136

CMMN Case Management Model and Notation. 61

CNN Convolutional Neural Network. 5, 6, 18, 36, 37, 39, 46, 49, 74, 110, 127, 160

COCO Common Objects in Context. 20, 42, 129, 153

CSS Cascading Style Sheet. 9, 177, 179

CSV Comma Separated Value. 137

CTA Click-To-Action. 191

CVS Control Version System. 42

DAI Distributed Artificial Intelligence. 56

DCNN Deep Convolutional Neural Network. 18, 19

DF Directory Facilitator. 58

DMN Decision Model and Notation. 61

DNA Deoxyribonucleic Acid. 62

DNN Deep Neural Network. 3, 4, 8, 10–12, 15, 19, 32, 35, 36, 51, 71–73, 81, 127, 128, 131, 151

DOM Document Object Model. 3–6, 9, 18, 47, 78, 79, 83, 89, 105–108, 114, 118, 137, 141, 144, 160, 179

DPS Distributed Problem Solving. 56

EEM Evaluation Engine Metaservice. 96

FCN Fully Convolutional Networks. 75

FIPA Foundation for Intelligent Physical Agents. 57–59, 67, 69, 94

FP False Positive. 125

FPN Feature Pyramid Netwrok. 41, 74, 75, 81, 86, 87

GAN Generative Adversarial Network. 44, 45, 179

GOODS Google Data Search. 42

GPS Global Positioning System. 56

GPU Graphical Processing Unit. 84

GUI Graphical User Interface. 2, 18

HCI Human-Computer-Interaction. 1

HOG Histogram of Oriented Gradients. 5, 109

HTML Hypertext Markup Language. 4, 7, 9, 18, 80, 89, 91, 105, 106, 141, 155

HTTP Hypertext Transfer Protocol. 58, 79, 85, 89, 90, 119, 130, 139, 143, 146

IoT Internet of Things. 55, 56

IoU Intersection over Union. 121, 122, 125, 148, 161, 165, 167, 180

JPL Jet Propulsion Laboratory. 59

M2M Machine-to-Machine. 1

mAP Mean Average Precision. 120, 125, 147, 148, 154, 155, 158, 161–163, 165, 166, 168–170, 173, 174, 178, 195

MAS Multi-agent System. 56, 57

MASC Manually Annotated Sub-Corpus. 136

MTS Message Transport Service. 58

NASA National Aeronautics and Space Administration. 59

NMS Non-Maximum Suppression. 121, 122, 133, 167, 170, 172, 178, 180

OCR Optical Character Recogniton. 16, 158

ORM Object-Relational Mapping. 140

PNG Portable Network Graphics. 140, 141, 147

R-FCN Region-based Fully Convolutional Networks. 37, 38

RCNN Regions with CNN features. 37, 38, 82, 87, 110, 157, 169, 177, 178, 197

REM Runtime Engine Metaservice. 95, 96, 100

ResNet Residual Network. 86, 87

REST Representational State Transfer. 61, 84, 94, 96, 103, 129, 130, 132, 159, 177

RoI Region of Interest. 5, 11, 12, 37–39, 46, 50, 82, 86, 109, 120, 172

RPN Region Proposal Network. 38, 82, 86, 87

SD Service Discovery. 59

SGD Stochastic Gradient Descent. 48, 81

SGG Spatial Graph Grammar. 109

SIFT Scale Invariant Feature Transformation. 5, 109

SL Semantic Language. 58, 67

SOA Service-oriented Architecture. 53–55

SotA State of the Art. 10, 15, 18, 32, 35, 36, 42, 68, 71–73, 81, 94, 95, 99, 152, 171, 177

SPP Spatial Pyramid Pooling. 6, 110

SSD Single Shot Multibox Detector. 6, 39

SVM Support Vector Machine. 5, 109

TP True Positive. 125, 175, 178, 181

TPU Tensor Processing Unit. 84

UI User Interface. 1–13, 15–18, 20, 23–33, 35, 42, 45–47, 49–51, 61, 71–75, 78, 80–83, 85, 89, 90, 92, 95, 97–103, 105, 106, 108, 109, 111–114, 117, 120, 122, 124, 125, 127–129, 132, 137, 138, 141, 147–149, 151, 152, 158–162, 164, 166, 171–175, 177, 178, 180, 192

UIM User Interface Management System. 1

URI Uniform Resource Identifier. 116, 143

URL Uniform Resource Locator. 79, 80, 89, 90, 114, 115, 119, 137, 141, 142, 152

UUID Universally Unique Identifier. 55, 56, 134

VIPS Vision-based Page Segmentation. 5, 106

VOC Visual Object Classes. 20, 21, 42, 89, 132, 137, 152, 153, 159–161, 163

W3C World Wide Web Consortium. 4, 9, 106

WFM Workflow Management. 60, 102, 118

WFMS Workflow Management System. 59, 93

WSDL Web Services Description Language. 94

WUI Web User Interface. 3, 18, 77–80, 82, 83, 88–92, 133–137, 152, 155, 159, 160

XML eXtensible Markup Language. 89, 140, 141, 147, 152

YOLO You Only Look Once. 6, 39, 40, 74

Appendix A.

Digital Content

The printed version of this document includes an SD Card with the following contents:

- The PDF version of this document.
- Complete Implementation Source Code.
- Evaluation Dataset.
- Thulium Platform - User Walkthrough.
- Thulium Platform - SQL Script - Database Schema.
- Thulium Platform - SQL Script - Database Data.
- Training Datasets.
- Trained Object Detection Models.

Appendix B.

UI Object Types

The following table contains a list of the different classes considered for this research.

Label	**Type**	**OCR-Related**	**Details**
button	Atomic	No	Click-To-Action (CTA) object. Allows the execution of actions. Visually seen as an object that can be pushed/clicked.
checkbox	Atomic	No	Input object visualized as a small box that can be ticked
textinput	Atomic	No	Input object where text can be entered as input. Supports only one single line of text
textarea	Atomic	No	Input object where text can be entered as input. Supports multiple lines of text
passwordfield	Atomic	No	Input object where text can be entered as input. Supports only one single line of text. Compared to a textinput, the entered characters are covered with different ones in order to prevent being visualized
radiobutton	Atomic	No	Input object that allows the selection of mutually-exclusive options. Normally a circular input that gets filled with a dot when clicked/selected

link	Atomic	No	Allows the linking between different intra/inter UIs views. Normally visually recognized by an underlined text that is also different in color compared to the text surrounding it
selectbox	Atomic	No	A list box that expands when clicked, displaying several options which can be selected or multi-selected
panel	Composite	No	An area that is visually separated from its surroundings by borders, shadows, and/or background color and contains at least one other UI object
list	Composite	No	Any kind of list (numbered or unnumbered) that uses bullet points, numberings, borders, background color, etc. to display a set of similar items
table	Composite	No	Any visually recognizable table (using alignment, lines or background color to represent rows and columns)
pagination	Composite	No	An area that contains pagination controls e.g. next and previous buttons and page links
tabs	Composite	No	Navigation elements that allow the reuse of a certain area in a UI for displaying different context-related content
datepicker	Composite	No	Represents a calendar with clickable days for selecting a date
symbol	Atomic	No	Any graphical symbol. Can appear on buttons, tabs, links, in texts, etc. or separately
image	Atomic	No	Foreground images that the web page displays

label	Atomic	No	small text portion, typically one word or only a few words, that are used together with another UI control like a radiobutton
paragraph	Atomic	Yes	A portion of text consisting of one or more lines of text that are not visually separated by white space and/or indentation from other text
textblock	Composite	Yes	Two or more subsequent paragraphs of text
text	Atomic	Yes	Any other portion of text that is neither a label nor a paragraph or textblock

Table B.1. UI Object Types

Appendix C.

Training - PascalVOC Metrics

The following tables show a detailed view of the different metrics computed through the three training iterations for Faster-RCNN and RetinaNet. The computed metrics contain AP values for each object type, as well as an overall mAP score.

	1st Iteration	**2nd Iteration**	**3rd Iteration**
button	0.8865289	0.9228126	0.9717989
checkbox	0.56497353	0.71450984	0.8949435
datepicker	1	1	0.96666664
image	0.6695559	0.6743582	0.7851627
label	0.7473322	0.81294215	0.9446322
link	0.6849003	0.89101994	0.9643344
list	0.12338624	0.80972177	0.9219001
pagination	0.7315283	0.9446186	0.9906781
panel	0.6550111	0.80456424	0.9339868
passwordfield	0.8911976	0.89260834	0.9604912
radiobutton	0.2702175	0.8682316	0.9693105
selectbox	0.7153046	0.9024155	0.9612554
symbol	0.6726449	0.74892694	0.8787822
table	0.94298613	0.9667047	0.9915399
tabs	0.010204081	0.59640735	0.8266215
textarea	0.9400353	0.9746835	0.9989301
textinput	0.7892208	0.8954782	0.9524223
mAP	0.6644134	0.8482355	0.93608564

Table C.1. PascalVOC Metrics - Faster-RCNN

	1st Iteration	**2nd Iteration**	**3rd Iteration**
button	0.84016156	0.9341655	0.9643116
checkbox	0.104286015	0.31924713	0.8582962

datepicker	0.989011	1	1
image	0.65374494	0.7063412	0.7563125
label	0.56366676	0.8227006	0.918581
link	0.50932753	0.8612196	0.9442941
list	0.029072741	0.74550736	0.8412665
pagination	0.57924557	0.86595035	0.9225803
panel	0.41425845	0.73971254	0.8191973
passwordfield	0.14244878	0.6289659	0.7767992
radiobutton	0	0.68348	0.92702645
selectbox	0.503054	0.70274246	0.80376744
symbol	0.44751567	0.6419823	0.8150262
table	0.7910589	0.9269892	0.96415305
tabs	0	0.35772136	0.5139712
textarea	0.6614887	0.73051953	0.89411926
textinput	0.39607906	0.7436811	0.8460912
mAP	0.44849527	0.7300545	0.8568114

Table C.2. PascalVOC Metrics - RetinaNet

Appendix D.

Precision/Recall Curves

The following figures refer to the precision/recall curves for each of the 17 classes considered for object detection in this research. Each of the curves contain the resulting values of both Faster-RCNN and RetinaNet.

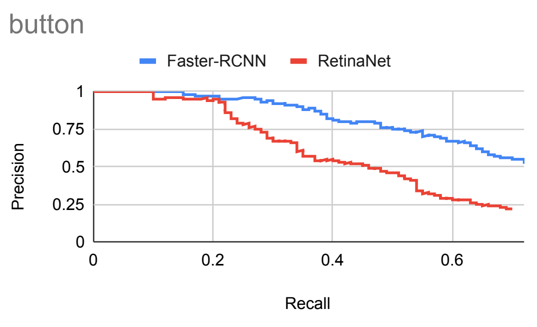


Figure D.1. Button - Precision/Recall Curve

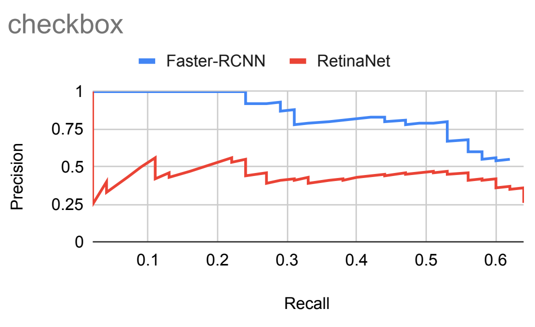


Figure D.2. Checkbox - Precision/Recall Curve

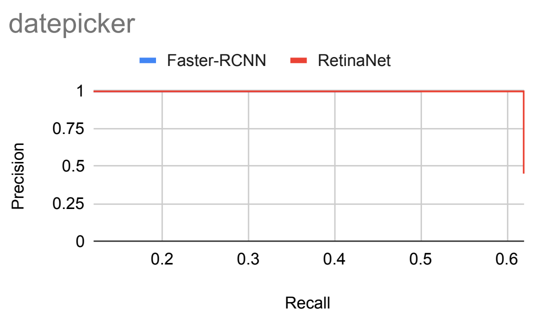


Figure D.3. Datepicker - Precision/Recall Curve

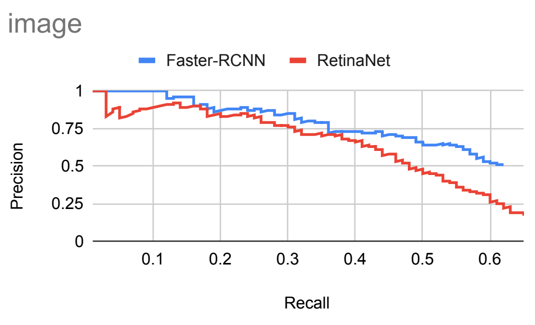


Figure D.4. Image - Precision/Recall Curve

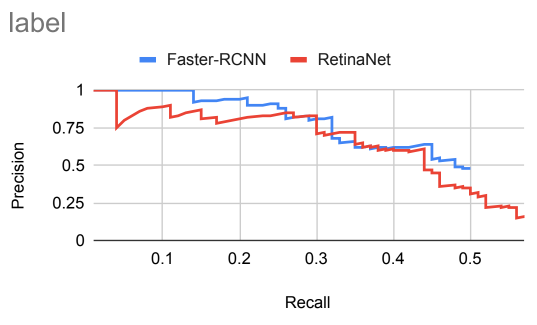


Figure D.5. Label - Precision/Recall Curve

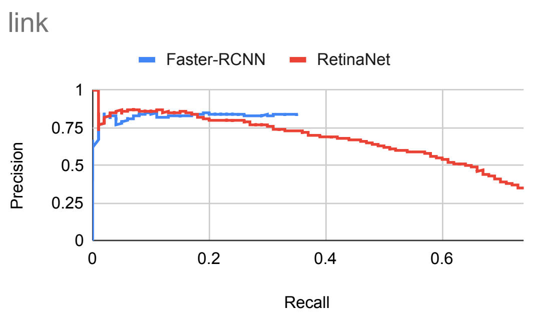


Figure D.6. Link - Precision/Recall Curve

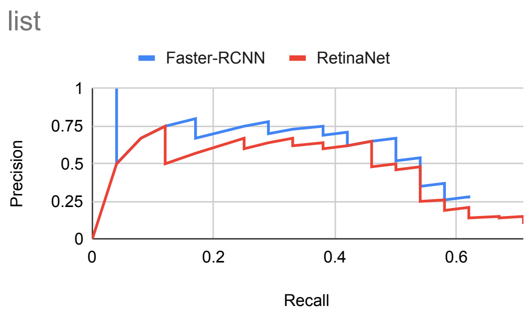


Figure D.7. List - Precision/Recall Curve

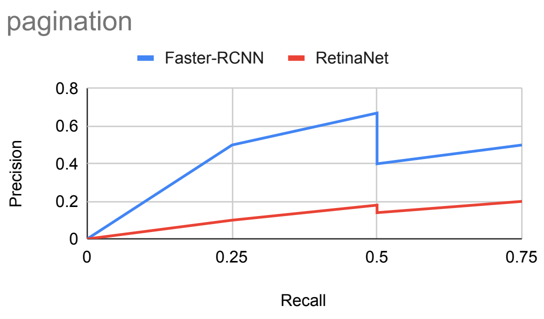


Figure D.8. Pagination - Precision/Recall Curve

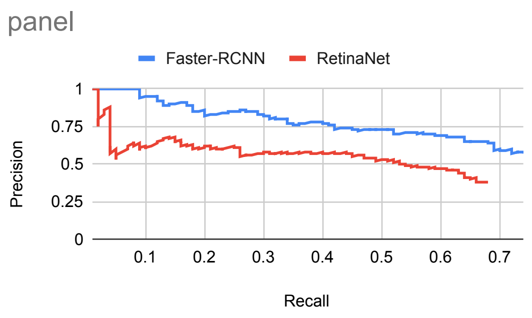


Figure D.9. Panel - Precision/Recall Curve

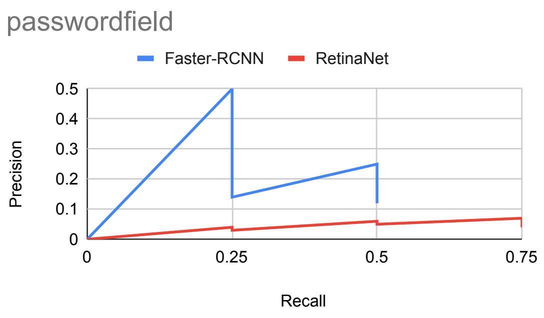


Figure D.10. Passwordfield - Precision/Recall Curve

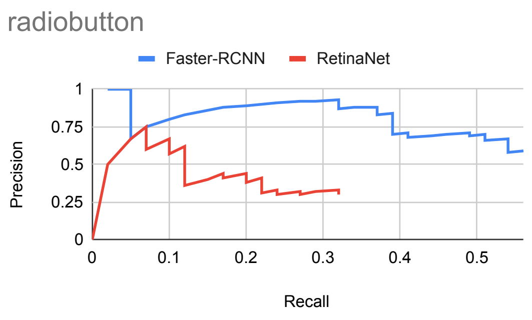


Figure D.11. Radiobutton - Precision/Recall Curve

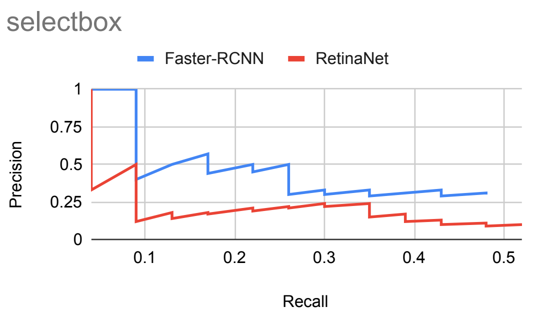


Figure D.12. Selectbox - Precision/Recall Curve

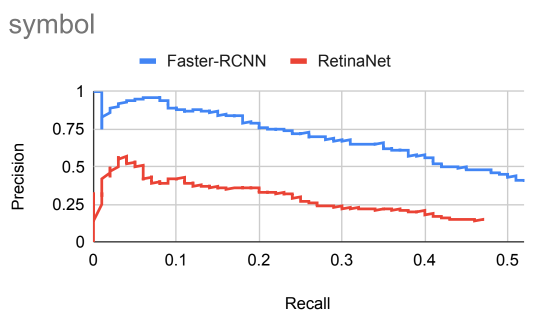


Figure D.13. Symbol - Precision/Recall Curve

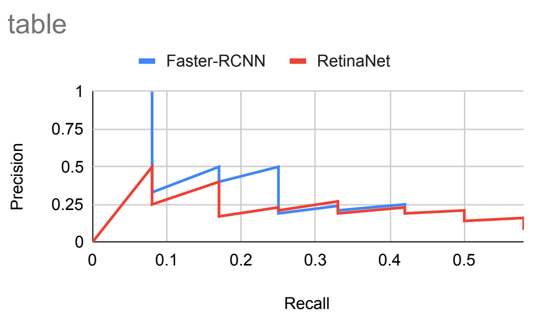


Figure D.14. Table - Precision/Recall Curve

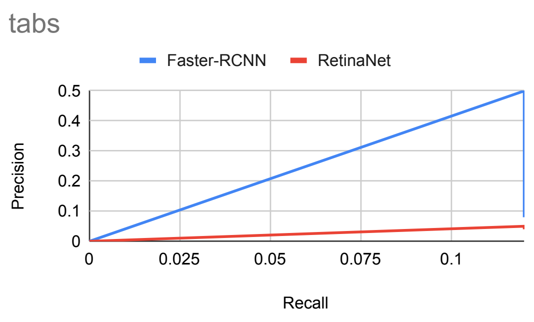


Figure D.15. Tabs - Precision/Recall Curve

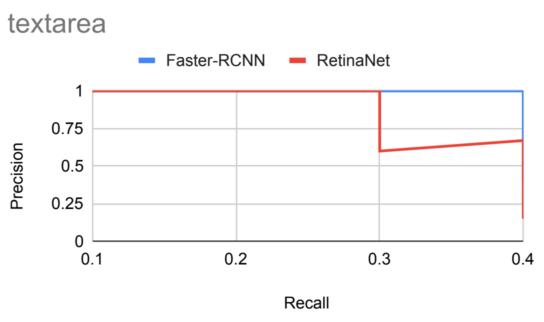


Figure D.16. Textarea - Precision/Recall Curve

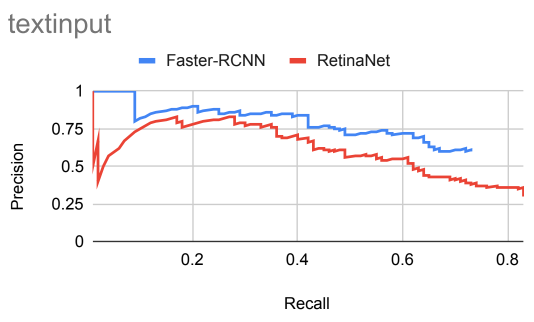


Figure D.17. Textinput - Precision/Recall Curve