UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Machine learning computer vision for robotic disassembly of e-waste

by

Mihail Georgiev

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: November 2023 Student ID: *removed*

Supervisor: Prof. Maurice Pagnucco
Assessor: Dr. Gwendolyn Foo

Abstract

This document describes a system for detecting screw fasteners as part of the robotic disassembly of e-waste products. The proposed architecture combines already explored approaches of using the deep learning, computer vision, and logical inference by separating the functionality into distinct modules with clearly defined interfaces and interactions. The work also defines a two-pass detection algorithm where computer vision and the logical inference stages are clearly separated. Further to that, it explores an implementation of a logic inference module that learns from already seen screw fastener layout patterns.

Acknowledgements

The work of this thesis has been inspired by Prof. Maurice Pagnucco and Dr. Gwendolyn Foo who encourage me to explore the scope of this project leading to an enhancement in my research skills and understanding the issues and opportunities in the field of robotic disassembly, computer vision and machine learning. I would also like to acknowledge all my teachers and tutors at UNSW, who shaped my vision and equipped me with the necessary skills to work on this project.

Abbreviations

AI - Artificial Intelligence

ANN - Artificial Neural Network

CNN - Convolutional Neural Network

CV - Computer Vision

GHT - Generalized Hugh Transform

MST - Minimal Spanning Tree

MV - Machine Vision

ROI - Region of Interest

SSD - Single Shot Detection

UML - Unified Modeling Language

WEEE - Waste Electrical or Electronic Equipment

YOLO - You only Look Once

Notation

For the rest of the text, the `courier` font is used to denote class name, method name, code extracts or operating system commands. The words in *italics* are used to denote terms.

Contents

List of Figures

List of Tables

List of terms

Term	*Definition*
Accuracy	The offset between the actual and detected screw centres
Base model	First stage of the *two-pass algorithm*
Detector	A module that classifies an *region-of-interest* image as a screw, a hole or neither
F1-score	Harmonic mean of *precision* and *recall*
Final layout	The *two-pass algorithm* output
Hough transform	A method for detecting circles in an image
Initial layout, Initial ROI	The output of the *base model*
Label	An area in the image, manually classified as a screw
Layout extension	The output of the second stage of the *two-pass algorithm*
Layout, Screw layout	Collection of coordinates on the *disassembly scene* image that denote the screw locations.
Pattern, Screw pattern	A screw layout graph, containing the matrix with the distances between the screws centres.
Precision	Performance measure for the rate of false positives
Proposer	Module that proposes additional screw locations based on detected screw locations and some logic and/or prior knowledge
Recall	Performance measure for the rate of false negatives
Region-of-interest	Area of the image that may be a screw
Scene, Disassembly scene	Image of the device that is being disassembled
Two-pass model, Two-pass algorithm	The subject of this thesis. It is an algorithm for detecting screw locations in a *disassembly scene*.

Chapter 1

Introduction

E-waste is defined as anything with a plug, electric cord or battery that has reached the end of its life, as well as the components that make up these end of-life products.

The future of e-waste

According to the World Economic Forum, the e-waste is the fastest growing waste-steam in the world, with 44 million tonnes generated at 2017 and estimating that to reach 120 million by 2050 [1].

For comparison, the amount of e-waste generated in 2017 was equivalent to 125,000 jumbo jets (more than all commercial aircrafts ever created) and equivalent to almost 4,500 Eifel towers [2].

By 2040, carbon emissions from the production and use of electronics, including devices like PCs, laptops, monitors, smart-phones and tablets (and their production) will reach 14% of total emissions (fig.1). This is one-half that of the total global transport sector today.

Figure 1 : The future of e-waste [2]

Economic value of e-waste

There is a lot of economic value in e-waste, particularly from such materials as gold, silver, copper, platinum and palladium, among others. There are also concerns about the availability and supply of new materials for electronics and electrical devices in the future. Rising commodity prices have highlighted those risks. Yet e-waste contains many high-value and scarce materials. For example, there is 100 times more gold in a tone of smart phones than in a tone of gold ore [2].

Benefits of disassembling

From environmental sustainability point of view, bringing material from end-of-life products back into the product cycle facilitates the conservation of natural resources and prevents pollution by preventing potentially hazardous materials from entering the landfill stream [3].

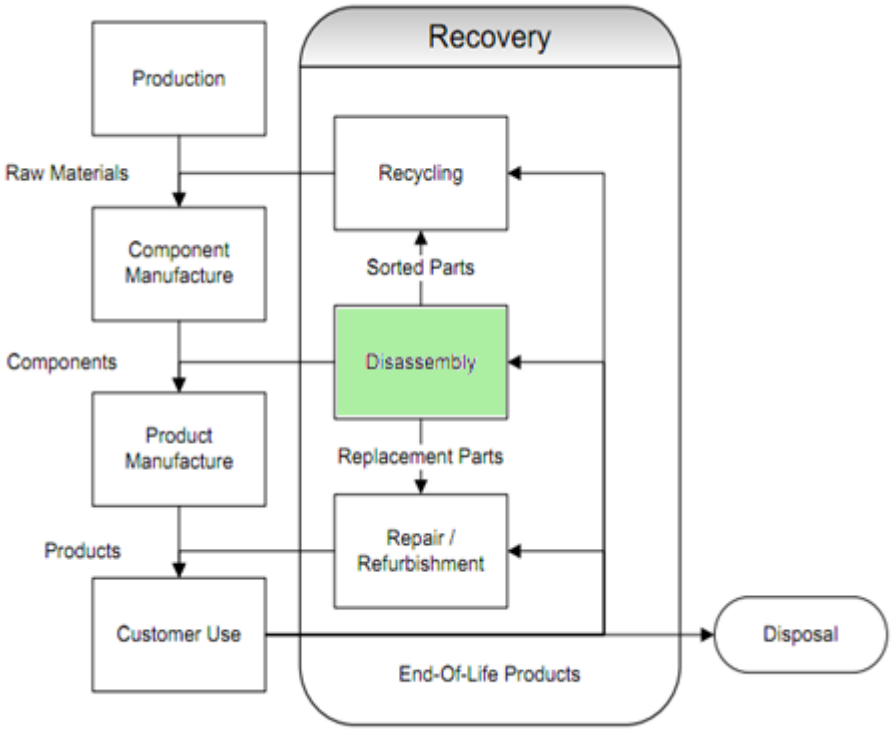


Figure 2 : Recovery options for end-of-life products [3].

End-of-life product disassembly allows the recovery of whole parts for reuse and remanufacture, thus preserving energy and value invested in the product manufacturing (fig.2) [3].

Benefits of robotic disassembly

Electronics disassembly is still extensively carried out using manual labour (fig.3). This is expensive and has the potential to expose workers to hazardous substances [3]. Robotic disassembly could have the potential to offer an efficient alternative for disassembling e-waste [4].

Figure 3 : Manual disassembly of e-waste [3]

The role of the machine vision

Robotic disassembly systems (fig.4) need to be able to recognize components and fasteners and their locations in order to remove them [5].

Therefore, a Machine vision (MV) system capable of automatically recognizing and locating these screws is a critical component.

Figure 4 : Robotic disassembly cell [6]

Chapter 2

Background

Robotic disassembly challenges

Robotic automation is very effective in context of production assembly lines, where the consistent execution of repetitive actions is wide spread. However, in the context of automated disassembly, there are challenges not present in the typical factory environment.

Those include the uncertainty of the product type, condition and location [5] [6] [7]. To deal with the uncertainties, the disassembly robots would require cognitive ability, and the machine vision is one of the most popular options.

Fastener detection, which is the topic of this work, is one of the first tasks in the disassembly process. The straight forward approach would involve direct application of computer vision methods; however, due to the small size of the fasteners, the standard computer vision (CV) algorithms do not provide sufficient reliability to be effectively used for industrial automation. That challenges led researchers to seek alternative approaches to fastener detection.

Fastener detection approaches

Computer Vision (CV)

The pure computer vision approaches rely solely on the photographic image of the *disassembly scene* obtained via static camera. They are easier to deploy and do not require extra investment in specialized equipment; however they are less efficient in handling the uncertainties [8] [9]. Such variations could be rust, dirt or scratches, as well as variation of the light conditions due to shadows and reflections from rest of the components, as well as the variations in the type and quality of the fasteners used across the variety of WEEE items.

Aided CV

In this approach the image of the static camera is supplemented by introducing of depth perception. That can involve the use of depth cameras or 3D vision system to provide

additional information to the cognitive unit. This approach can achieve higher resilience to variability of the disassembly process but require extra investment in specialized equipment and additional logic to combine and process the depth perspective [10].

Localized CV

This approach introduces a moving camera or "camera in the hand", where the robot inspects the *regions of interest* (ROI) in the *disassembly scene* by moving the camera. It requires some other mechanism to detect the ROI. Because of the inherent movement in this approach the CV object detection algorithms are limited to the single step detection algorithms such as YOLO [11]. Those type of algorithms are less precise, however that is compensated by the richer visual information obtained by the camera from various points of view.

AI supplemented CV

In this approach, the initial processing of the *scene* is similar to the CV - it involves analysing a static image of the *scene* obtained from a fixed location camera, however the results are further improved by applying heuristics or inference rules to predict any missed objects / fasteners [5] [6] [12]. This approach will be in the focus of the current work.

Deep Learning and CV

The following section provides a brief overview of the deep learning and computer vision and may be skipped by readers familiar with the subject.

The deep learning in computer vision refers to the use of multilayer (a.k.a. deep) artificial neural networks (ANN), and in particular, convolution neural networks (CNN) to analyse images [13].

Unlike regular multilayer ANN, the CNN has several input (convolution) layers that are not fully connected. Each neuron in the convolution layers connected to a localized region of the input. That allows CNN to be able to handle much larger input (e.g. high resolution images). Each convolution layer is also known as a feature map and is usually aggregating the features from the previous layer. It may also involve various type of image filtering. The figure 5 shows the structure of the VGG-16 CNN architecture, which won the ImageNet 2014 competition.

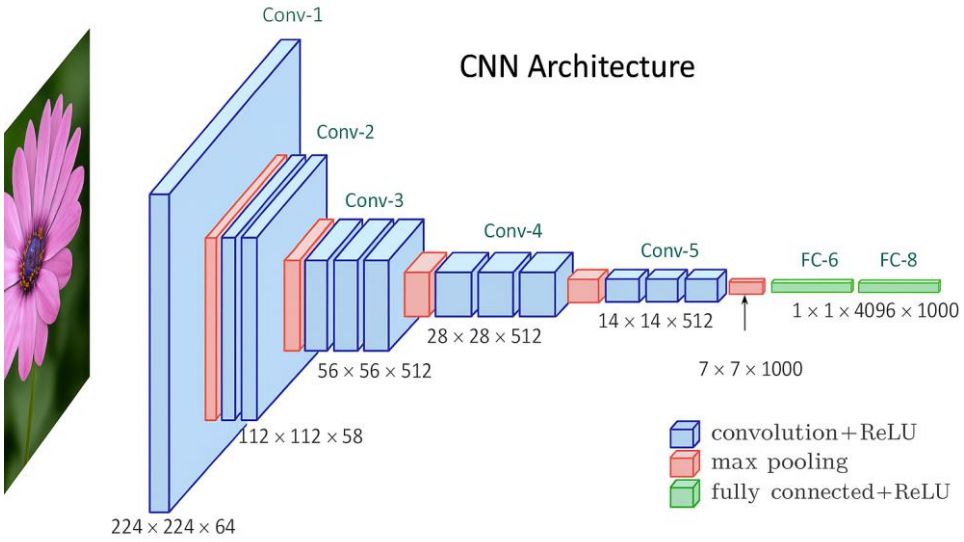


Figure 5 : CNN architecture

CNN and Object Detection

From computer vision perspective, detecting objects in a *scene* involves two tasks [13]:

- Localization - detecting where the object are
- Classification - determining the category the objects belong to.

So the pipeline of object detection algorithms could be divided in three stages [13]:

- Selection *regions of interest* (ROI),
- Feature extraction
- Object classification

Considering the above tasks, the CV algorithms for object detection can be classified in two types [13]:

- Two stage algorithms, such as R-CNN
- Single stage algorithms (SSD, YOLO)

The main representative of the two stage algorithms is the regional proposal CNN or R-CNN algorithm. It features higher *precision* but is relatively slow compared to its single stage competitors. That makes it inappropriate for object detection in video streams. The algorithm

had undergone several stages of improvement (e.g. Fast & Faster RCNN) which greatly improved the speed and the *precision*.

The Single Shot Detection (SSD) and You Only Look Once (YOLO) are the main representatives of the single stage algorithms. Their main feature is fast operation, which makes them applicable for object detection in video streams. However, the disadvantage is the relatively low *accuracy* compared to R-CNN. Similar to the R-CNN those algorithms have undergone several stages of improvements in the last decade.

Separation of ROI and Fastener detection

Using classic CV object detection for fastener detection does not produce sufficiently reliable results due to the relatively small size of the screws.

That led the researchers to explore designs where ROI and fastener detection were separated in order to apply alternative approaches for solving them.

Most recent work in that area is [8], where *Hough transform* was used for selecting the ROI from the scene, then a combination of classical CV algorithms are used to detect fastener in the selected ROI.

According to the paper, the screw classifier achieved very high *accuracy* (98%), however the overall performance was limited by the relatively low *accuracy* of the ROI detection stage (75%) [11].

Using AI to enhance CV results

Alternative approach to improve the reliability of the CV results for fastener detection was explored in [5], where an AI (Visual Reasoning) module was used to predict location of undetected screws based on predefined ontology and a system of inference rules.

Learning Patterns

Another area of enhancing the screw detection system performance was explored in [4], where the system was equipped with a module to remember pieces of information and screw locations for already seen laptop models.

Chapter 3

Proposed Architecture

Combine the approaches

In this thesis I propose a design that combines some of the previous approaches as follow.

- Separate the main components into three distinct modules with clearly defined interfaces.
- Implement *two-pass* detection algorithm, where first pass is using CV to detect some of the screw and the second pass is using the result of the first pass and some extrapolation logic to propose addition *regions of interest* (ROI)
- The proposed additional ROI are further classified by a CV module and the detected screws are added to the final result
- In addition to above, I also provide a *pattern* matching *propos*er implementation that remembers the previously seen *screw layouts* and use them for predictions.

To my knowledge all of the three propositions (above) are novel.

Modular Architecture

The proposed modular architecture provides flexibility for swapping module implementation without affecting the rest of the system. That enables easy exploration of different combinations of the module implementations.

The following diagram outlines the main modules in the proposed screw detection system architecture, their interfaces and relations.

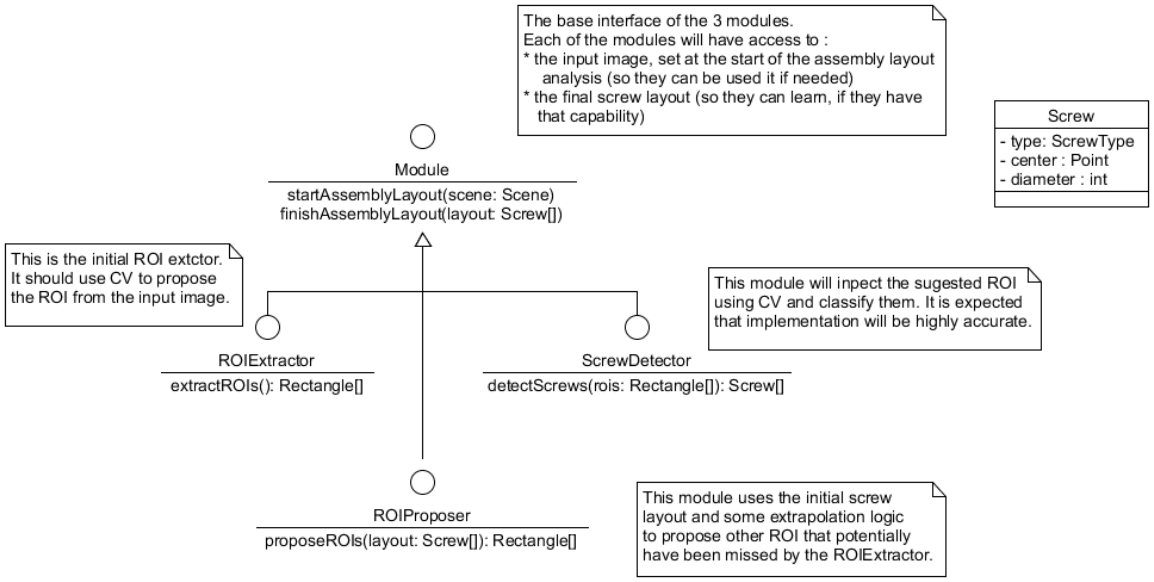


Figure 6 : UML class diagram of the module interfaces

The `Module` interface

The `Module` interface is the base interface and the root of the hierarchy. It has couple of methods that are common to all modules.

The `startAssemblyLayout` method takes as argument the input *scene* and is called at the beginning of *scene* analysis.

The `finishAssemblyLayout` method takes as argument the final *screw layout* as detected by the system. That method is called at the end of analysing the *scene* and its main purpose is to allow providing feedback to the module implementation. Some implementations, such as the screw *matching proposer* implementation, can use that to learn and adapt their behaviour.

ROIExtractor

The role of the ROI extraction module is to analyse the *scene* and propose the initial *regions of interest* (ROI). The `ROIExtractor` interface extends the base `Module` interface and adds one method `extractROIs`, which returns list of the rectangles representing the *regions of interest* (ROI) in the input scene. In this thesis I use implementation based on the *Hough transform*, detecting circles in the input scene, which is the same approach as in [8].

ScrewDetector

The screw detection module is classifying the *regions of interest* (ROI) extracted from the *disassembly scene*. The `ScrewDetector` interface inherits the methods of the `Module` interface and adds a single method `detectScrews` that takes a list of ROIs as input and returns list of detected screws. In this thesis, I have implemented the ScrewDetector using the Xception CNN network, trained over selected images, manually classified as screw, whole or other. The same type of *detector* was used in [8] [9] however it was a combination of two CNN networks (Xception and Inception). I decided to use only Xception as it performed slightly better and the using two networks did not improve much the performance but it significantly slowed down the processing. Considering that I was running the tests on a single general-purpose laptop, the combination of two CNNs as in [8] [9] could provide benefits in case a high-performing machine is being used.

ROIProposer

The `ROIProposer` module is taking the *initial screw layout* detected so far and is using it along with some extrapolation logic to propose additional *regions of interest*. Similar to other modules, the `ROIProposer` interface extends the `Module` base interface and adds a single method `proposeROIs` that takes a list of screw locations (*screw layout*) as input and return a list of additional ROIs.

Two-Pass Screw Detection Algorithm

The three modules described earlier collaborate in the *two-pass screw detection algorithm*.

In the first pass, the algorithm uses pure computer vision to perform the initial screw detection. We will refer to the first pass of the algorithm as the *base model*.

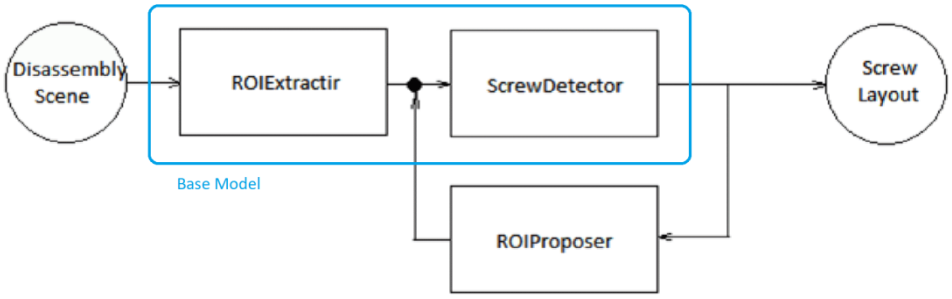


Figure 7 : Two-pass detection - system diagram

In the second pass, the result of the first pass is used to logically infer additional, potentially missed *regions of interest* that may contain screws. Those ROI are tested and added to the *screw layout* in case they are classified as screws.

In the final stage the algorithm provides feedback to the participating modules allowing them to learn and enhance their performance.

The following is a pseudo code of *two-pass screw detection algorithm*.

```
/*
 * Pseudocode of the Two Phase Screw Detection Algorithm
 */
algorithm TwoPassScrewDetection
    input:
        scene // input photo of the scene
    output:
        layout // list of screw locations
    state
        Extractor // module that does the initial ROI extraction
        Detector // module that inspects a list of ROI and returns list of detected screw locations
        Proposer // module that extrapolates a screw layout based on some rules
begin
    // Initialise modules
    Load the scene into Extractor, Detector and Proposer

    // Pass 1: Layout extraction
    initialROIs = Extractor extracts the ROIs from the scene
    initialLayout = Detector detect screws by inspecting the initialROIs

    // Pass 2: Layout extrapolation
    additionalROIs = Proposer proposes ROIs by inspecting the initialLayout and using some extrapolation rules
    additionalLayout = detector detect screws by inspecting additionalROIs

    layout = join initialLayout and additionalLayout
end
```

The following sequence diagram represents the *two-pass screw detection algorithm*, outlining the interactions between the main modules (`ROIExtractor`, `ScrewDetector` and `ROIProposer`).

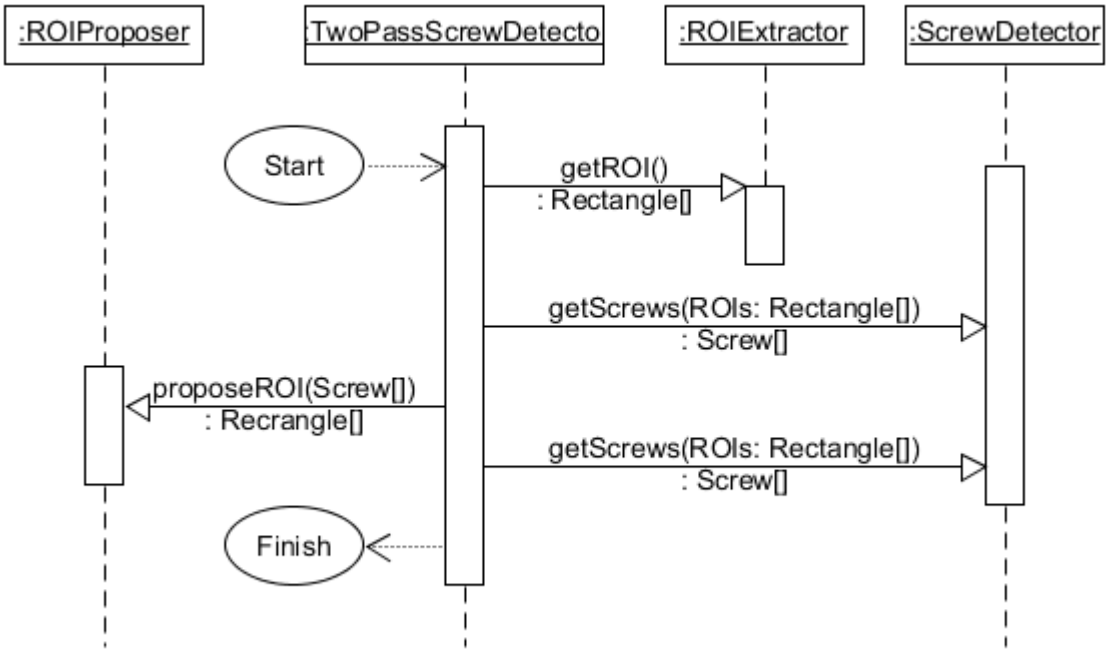


Figure 8 : Two-pass detection - sequence diagram

The algorithm starts by the `TwoPassDetector` calling the `extractROIs` method on the `Extractor`, which returns the list of rectangles, representing the extracted *regions of interest*.

Next, the `TwoPassScrewDetector` calls the `detectScrews` method of the `ScrewDetector`, passing the ROI from previous step. The `detectScrews` method returns the location of the detected screws. This concludes the first pass and is the output of the *base model*.

On the second pass, the `TwoPassScrewDetector` calls the `proposeROIs` method of the `ROIProposer`. It returns a list of additional *regions-of-interest*.

Next the `TwoPassScrewDetector` again calls the `detectScrews` method of the `ScrewDetector`, this time passing the additional ROI returned by the proposer. The returned list of screws is the *layout extension* and is the output of the *second pass*. The *final layout* is the combination of the *initial layout* and the *layout extension.*

Pattern Matching Proposer

The `PatternMatchingProposer` class is a `ROIProposer` implementation that matches the already detected *screw layout*[1] against a collection of known *screw layouts* and proposes new *regions of interest*.

It organizes the stored *screw layouts* by creating unique descriptors composed of the set of the unique distances between the screws in the fully connected graph of the *layout*.

It uses the *layout* descriptors to select the candidates when trying to find a *layout extension* of a partially matched[2] *screw layout*.

Then for each of the selected layouts, it tries to find the best possible match by applying various combinations of translation and rotation. The best possible match is the one that matches the most screws from the *initial layout*.

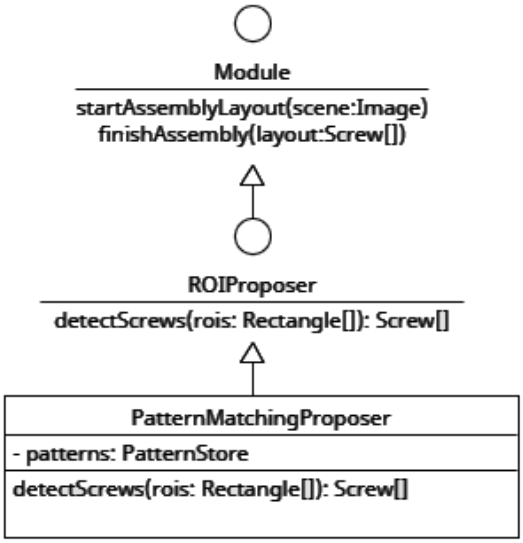


Figure 9 : Pattern Matching Proposer

[1] The terms *screw layout*, *screw pattern* and *pattern* are interchangeable for the rest of the text.

[2] The partially matched *layout* coming from the *base model* will be also referred as the *initial layout*

The following diagram shows the *pattern* matching algorithm in action.

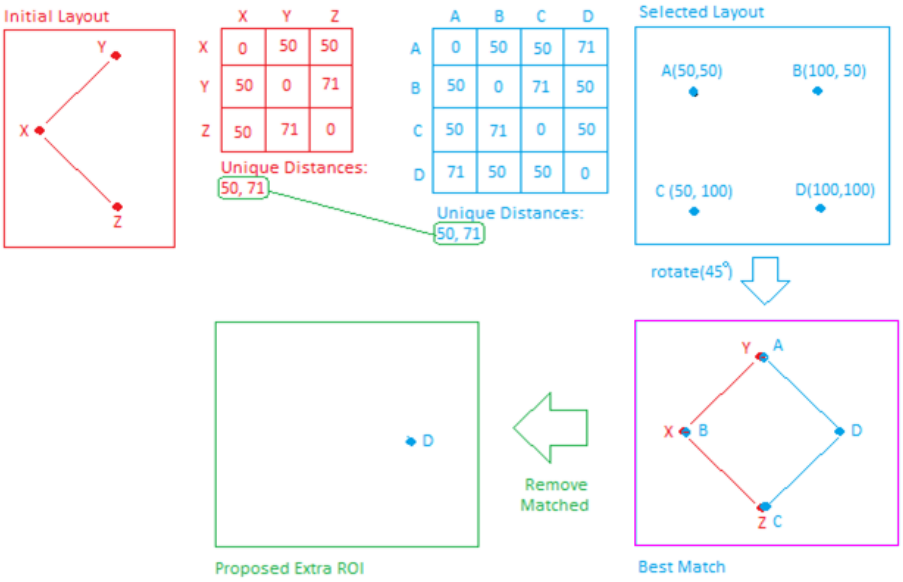


Figure 10 : Pattern matching example

First we build the graph of the *initial layout* and extract the unique distances (red). Then we search for a graph that contains those distances (blue). We try various rotations and translations on the selected graph to find best match for the *initial layout* (magenta). Finally we subtract (remove) the matched points from the transformed graph (green).

Following is a pseudocode of the `PatternMatchingProposer` algorithm.

```
algorithm PatternMatchingProposer
    input:
        initailLayout // list of screw locations detected by the base model
    output:
        proposedROIs // list of proposed extra ROIs
    state:
        layoutDatabase // repository of known layouts
begin
    uniqueDistances := Get the unique distances between the screws in the initailLayour
    selectedLayouts := select from the stored layouts the patterns that contain those unique distances
    for each of the selectedLayouts
        bestMatch := get best match with the initail layout
        matchScore := number of matched screws / number of screws in the initalLayout
        if matchScore > best matchScore so far, then
            proposedROIs := bestMatch - initaillayout
        end if
    end for
end
```

Pattern Learning

In order to add learning functionality to the proposer, I first created a new component – `PatternStore` and defined its interface.

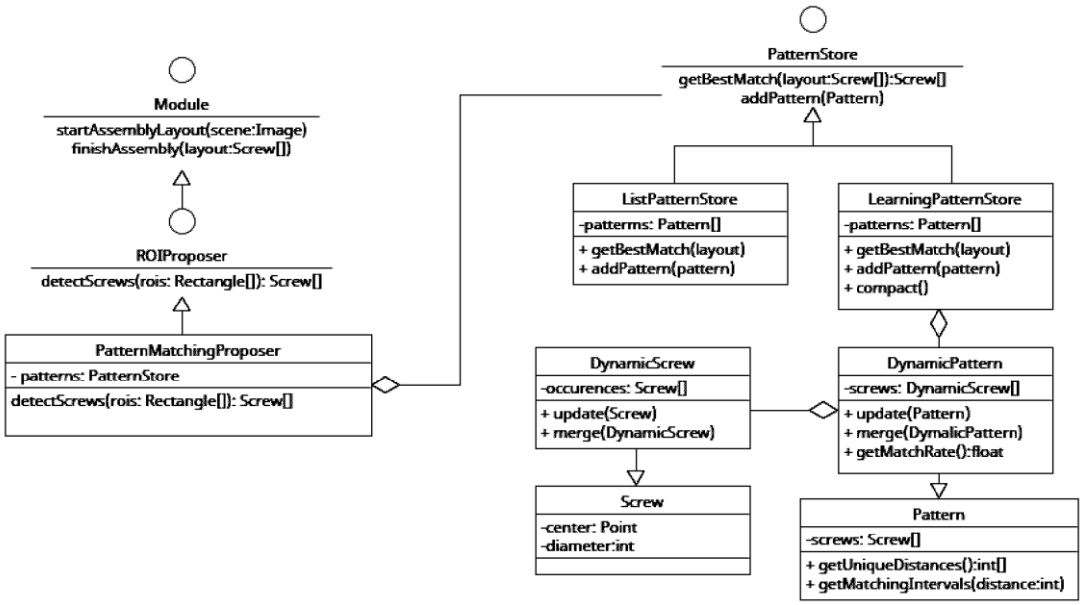


Figure 11 : Pattern Store - class diagram

The `PatternStore` is a repository for already known patterns and its interface provides methods for adding more patterns as well as for finding the best match among the stored patterns for a given input *screw layout*. It also provides self-cleaning functionality in the form of merging patterns that are very similar as well as removing or forgetting patterns that are rarely used. That functionality can also be triggered via the `compact` method.

The `PatternStore` interface has two implementations – the `ListPatternStore`, which is the non-learning implementation (it is instantiated with a set of pre-loaded patterns). The second implementation is the `LearningPatternStore`, which implements the *learning pattern* behaviour.

The `LerningPatternStore` in turn relies on `DynamicScrew` and `DynamicPattern` classes to keep track of the multiple occurrences of the same pattern.

The `DynamicPattern` represents a set of screw locations that occur together in a scene. It is built by accumulating many instances of the same *pattern* over many scenes. Each

instance may have some of the screws in the pattern, miss some and also contain some false positives.

Following diagram is a conceptual representation of a `DynamicPattern` instances.

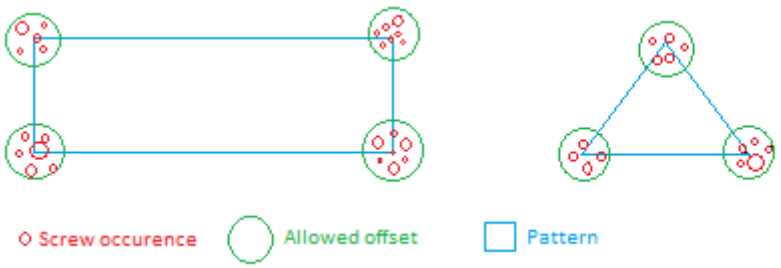


Figure 12 : Dynamic pattern example

The screw locations in the *pattern* instance (the red dots) must be within some range from the mean centre of the other screws (the green circle). Over time, the screw location that are not detected often (or false positives) will be removed from the pattern, while those that are often used will become the core of the pattern.

Learning algorithm

The following is the pseudocode of the *pattern learning algorithm*.

```
algorithm MemoriseLayout:
    input:
        patternStore // List of memorised patterns
        newLayout // Layout (list of screws) to be memorised
        mergeThreshold // minimum matchscore needed for two patterns to be considered equal
    output:
        patternStore // the updated list of patterns
begin
    bestMatchPattern := Find the best matching pattern from memorised patterns
    bestMatchScore := number of matched scrws / total number of screws in the newLayout

    if bestMatchScore > mergeThreshold,
    then update the bestMatchPattern with the newLayout
    else create a new dynamic patterns from the newLayout and add it to the store

    Periodically (every N additions) compact the store
end
```

The algorithm takes a collection of memorised patterns and a new *screw layout* to be memorised. It also needs a threshold to determine if two patterns are equal.

The output is a new collection of patterns that contains the same or fewer patterns than the input.

On the first step, we find the best matching memorised *pattern* for the new *layout*. The best match is the one that matches the most screws from the new *layout*.

Next we calculate the match score, which is the ratio of the number of matched screws divided by the number of screws in the new *layout*. If that ratio exceeds the specified merge-threshold, we merge both patterns. Otherwise, we add the new *layout* as a separate pattern.

Finally, on every *N layout* additions, we run the self-cleaning / store-compact algorithm.

Compacting the Pattern Store

Following is the pseudocode of the *pattern store compacting algorithm*.

```
algorithm CompactPatternStore:
    input:
        patternStore // List of memorised patterns
        mergeThreshold // the minimum matchscore needed for two patterns to be considered equal
        forgetThreshold // the minimum usage score for pattern to be retained in store
    output:
        patternStore // the updated list of patterns
begin
    for (each pattern in patternStore:
        if usage ratio < forgetThreashold, then remove the pattern from store
        else
            bestMatch := Find the best match among other patterns in the patternStore
            if bestMatchScore > mergeThreashold, then merge the two patterns
end
```

The store compact algorithm's purpose is to minimise the store size without affecting the predictor performance significantly. It does two things:

- Merge patterns that are too similar
- Remove patterns that are rarely used

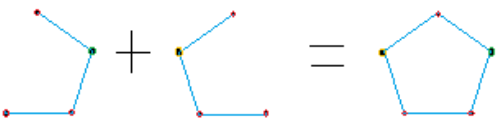


Figure 13 : Pattern merging example

To do so it need two thresholds:

- The forget threshold – which is the minimal number of *pattern* uses for the last N additions.
- The merge threshold – which is the minimal match score between two patterns to be considered equal.

Considering the above, the algorithm simply iterates through the memorised patterns and for each *pattern* and:

- Check if the usage is below the forget threshold, if so the *pattern* is removed

- Find the best match among other patterns and if it is above the merge threshold, it merges the two patterns

Testing and Simulation Framework

In order to test the performance of the *pattern matching proposer*, I have implemented proxy implementations of the `ROIExtractor` and the `ScrewDetector`.

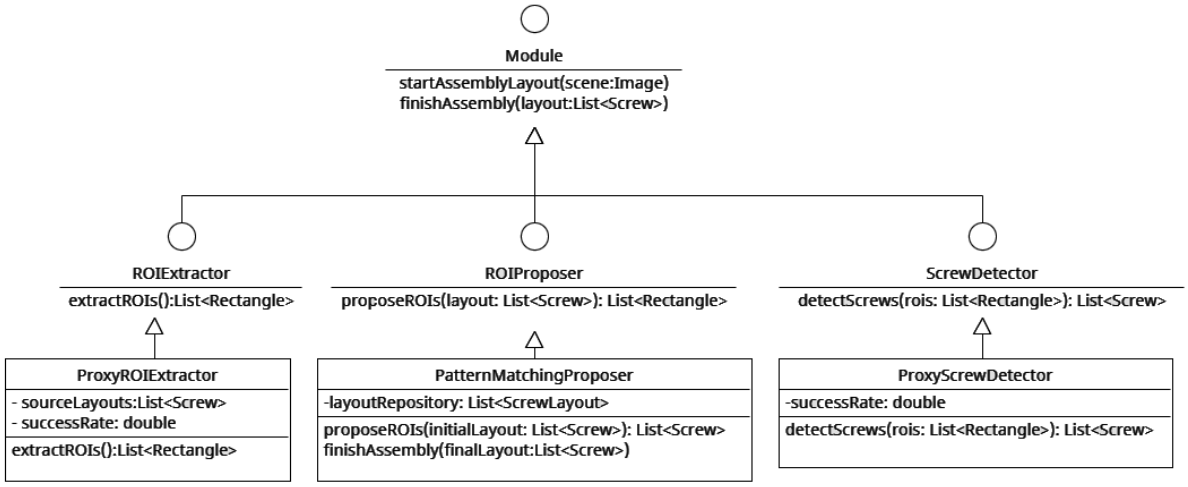


Figure 14 : Proxy Implementation - Class Diagram

The `ProxyROIExtractor` randomly selects between three predefined *layouts* (triangle, square and a hexagon), then applies random rotation and translation.

It also does small random offset of the screws, randomly drops screws based on a predefined probability as well as adds random locations – all that to simulate noise in the initial detection.

The `ProxyScrewDetector` also randomly drops some of the ROIs that is receives from the `ROIDetector` and that way simulating errors in the classification.

Chapter 4

Evaluation

Generating test images

In order to evaluate the performance of the *base* and the *two-pass model* I needed sufficiently large input of *disassembly scene*s with labelled actual screw locations. To create such dataset of *disassembly scene*s, I selected several images of different devices subject to disassembly. Then I applied the following transformations:

- Remove perspective
- Remove background
- Label the scene

And finally I implemented a *scene* generator tool that takes a set of source scenes and generates an infinite number of new scenes by randomly rotating and translating the source scenes.

The removal of perspective is necessary for the *pattern matching proposer* operation. Memorised patterns are geometrical, so any change in the distances between corresponding set of points in two images of the same device would result in different *layouts*. I removed the perspective from the images by applying the Open-CV [14] `warp-perspective` function on the input scenes.

In order to prevent detecting screws that do not belong to the device being disassembled, I manually cropped the device from the background. Such transformation can easily be applied in real life by using the calibration image of the empty workbench and subtracting it from the *disassembly scene* using binary segmentation.

As I mentioned previously, in order to use the images for performance evaluation; we need to provide the locations of the actual screws. To do so, I manually annotated the scenes, marking and classifying the *regions of interest* containing screws and holes. I used the online annotation tool www.makesense.ai

Finally, I needed a module to read the *disassembly scene* images and the labelling information and feed that into the *two-pass algorithm*. For that purpose I defined the `SceneSource` interface and provided couple of implementations – `SceneReader` and `SceneGenerator`. The following class diagram represents the `SceneSource` interface methods and relationship.

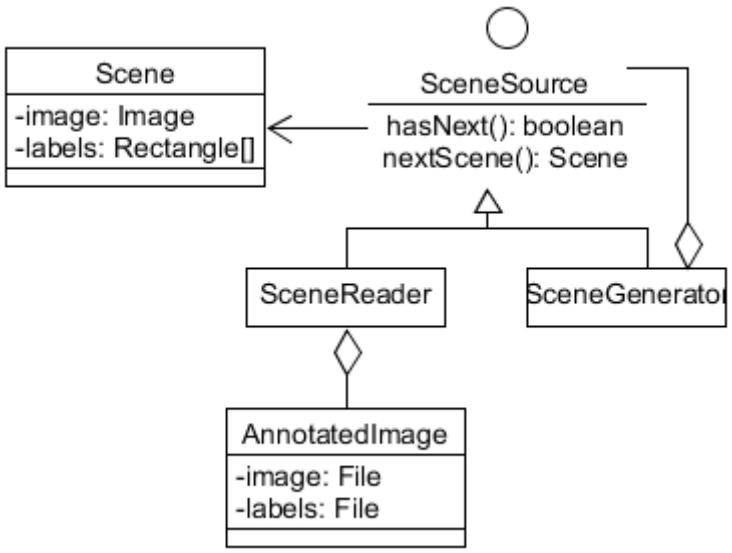


Figure 15 : SceneSource interface and implementations

The `SceneSource` interface offers two methods – `hasNextScene` and `nextScene`. The `nextScene` returns the next available *scene* in the `SceneSource` and may return `None` if all scenes have been provided. The `hasNextScene` method returns `True` if there are more scenes in the `SceneSource` and is guaranteed that the next call to `nextScene` will return a valid *scene* instance.

The `SceneReader` implementation of the `SceneSource` is responsible for loading stored *scene* images and annotations from the file system and parsing them into `Scene` instances. It takes list of directories and tries to find the images and the corresponding annotation files in each of them ordering them alphabetically. The combination of image file and annotation file is represented internally as `AnnotatedImage` class instance.

The `SceneGenerator` implementation takes another `SceneSource` (usually a `SceneReader`) and uses its images as a base for generating any number of *disassembly scene*s. It does that by applying random rotation, translation, light and contrast change to the

base scenes. In addition to transforming the base *scene* image, it also applies the same transformations to the image *labels*, so the generated *scene* is also annotated.

The following are samples of the base and some generated *disassembly scene*s using the above approach.

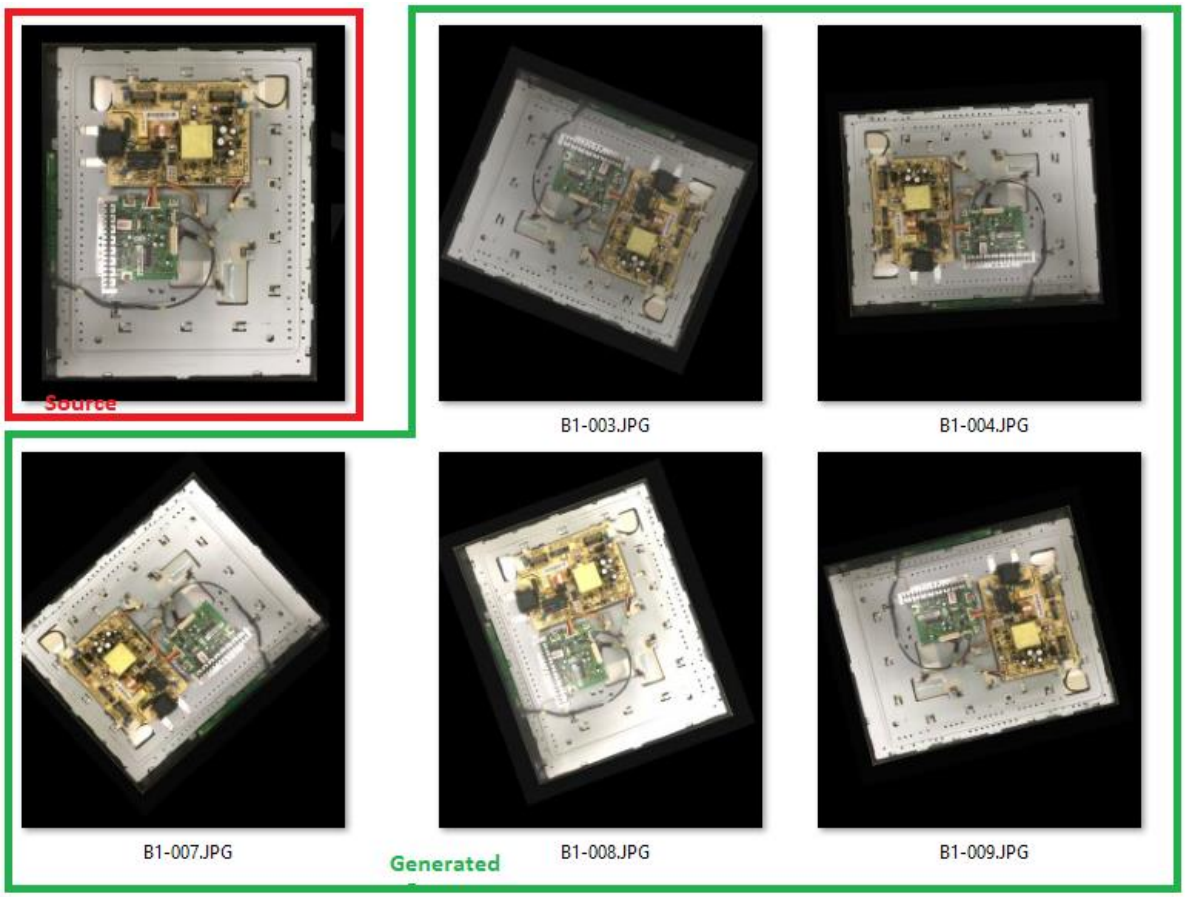


Figure 16 : Sample generated disassembly scenes

Performance Metrics

I used the following metrics to measure the performance of the system as well as the performance of the *base model*:

- Precision
- Recall
- F1-score
- Mean offset

Precision is a measure of the rate of false positives. It is defines as a ratio between the true positives over all positives: $Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$

Recall is a measure of the rate of false negatives. It is defined as a ratio between the true positives over the sum of true positives and false negatives:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

F1-score is a harmonic mean between the *Precision* and *Recall* and I use it as a primary measure of the performance. It has the benefits that combines both *precision* and *recall* into a single metric and is also less sensitive to extremes in either of them compared to the simple mean: $F1score = \frac{2 \times Precision \times Recall}{Precision + Recall}$

The *mean-offset* is the average distance between the centres of the detected and actual screws.

In order to evaluate and report model performance, I defined a new module type - `Observer` that can be plugged into *two-pass algorithm* and notified of the results of each stage of the image processing.

The following diagram defines the interface methods and shows at which stage of processing they are called.

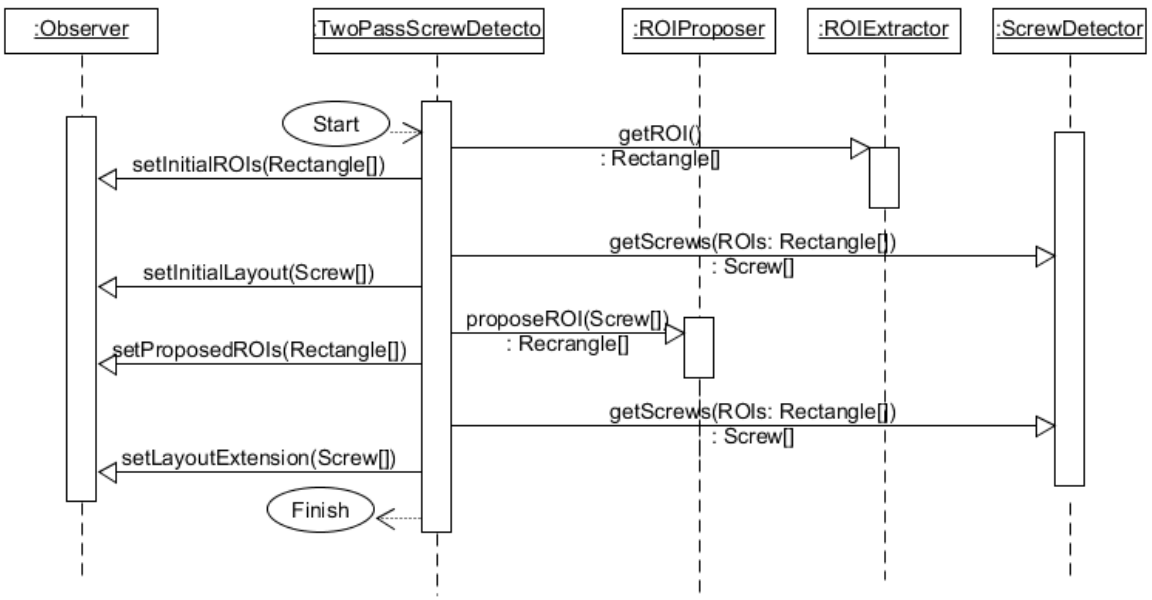


Figure 17 : Two Pass algorithm using Observer interface sequence diagram

I have provided three implementations of the `Observer` interface: `CompositeObserver`, `PerformanceObserver` and `ImageDecoratorObserver` classes.

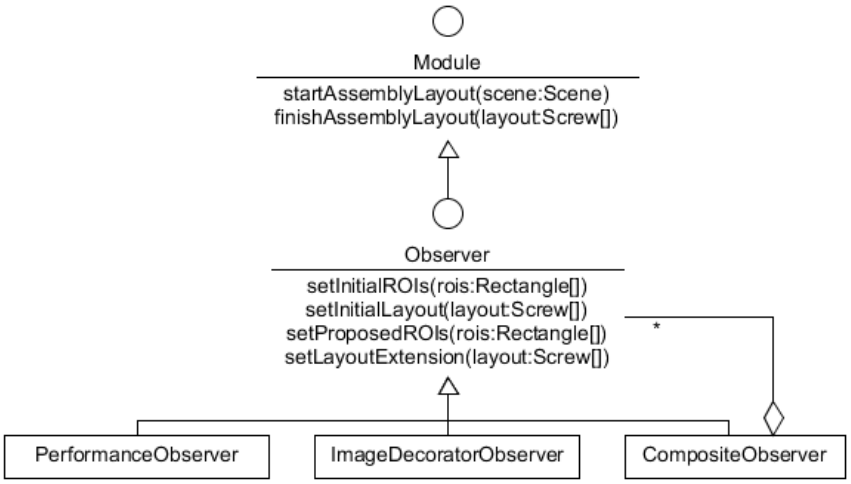


Figure 18 : Observer interface and implementing classes

The `PerformanceObserver` is responsible for calculating the four metrics during each *scene* processing for the *base* and for the *two-pass model*. It also keeps a record of the metric values and calculated the metrics means. On the last step of the *scene* processing the `PerformanceObserver` reports the calculated metrics and the means on the console.

The function of the `ImageDecoratorObserver` is to create an output image that visually represents the output of the different stages of the two pass algorithm.

The `CompositeObserver` allows combining several observer instances. It takes a collection of `Observer` implementation and each time one of its methods is called, it calls the corresponding method of each one of its `Observer` instances that it contains.

Following is a sample decorated output image.

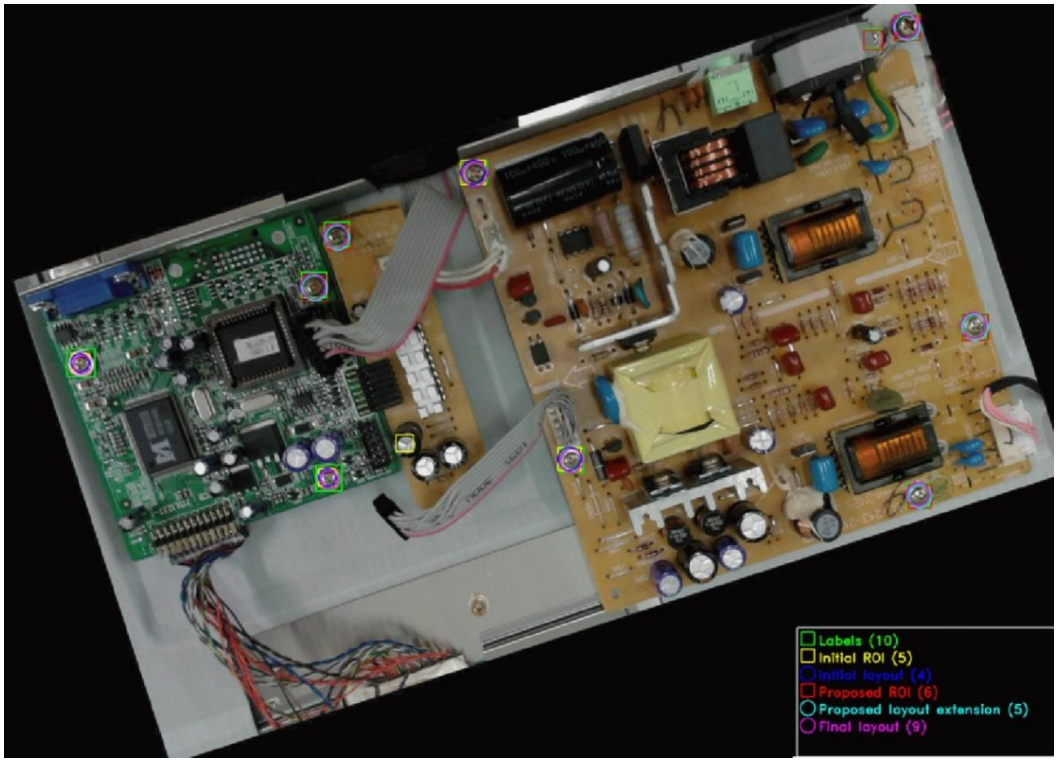


Figure 19 : Sample decorated disassembly scene

The *labels* are marked with green squares and represent the actual screw locations as seen during the *scene* manual labelling.

The initial ROIs (yellow) are the *regions of interest* returned by the `ROIExtractor`. The *initial layout* screws (dark blue) represent the output of the *base model*. The proposed ROIs (red) are the output of the `PatternMatchingProposer`, where the proposed *layout extension* (cyan) is the subset of those that were classified as screws. Finally, the output of the *two-pass algorithm* is the *final layout* (magenta).

Best Case Scenario

The purpose of this test is to explore the best-case scenario for the *two-pass algorithm*, where all the patterns of the input scenes are already in the *pattern* memory. The only source of variability in this case is the imperfections of the *base model* `ROIExtractor` and `ScrewDetector`.

For this test I have generated 100 scenes out of three base images. The test scenes are generated through the use of the `SceneGenerator`, which transformed the base images by a random rotation, translation, light & contrast change.

The *labelled layout* of the three base scenes was already pre-loaded in the *proposer pattern store* memory.

The test result showed that the *two-pass algorithm* improved the *base model* performance on the primary metric *F1-sore* by 63%.

Table 1 : Pre-loaded patterns test results

	Precision	Recall	F1-score	Offset
Base Model	0.98	0.28	0.43	3.08
Two-Pass Model	0.93	0.61	0.70	4.07
Performance Gain	-5%	117%	**63%**	-32%

Appendix C shows few sample decorated scenes from the test, generated by the use of the `ImageDecoratorObserver` class.

Testing the Learning Proposer

The purpose of this test is evaluating the effectiveness of the *two-pass algorithm* with a *learning pattern store*. For the test I configured the *two-pass model* with an empty instance of the `LearningPatternStore` and similarly to the best case scenario, I pass as input 100 scenes generated through the use of the `SceneGenerator`, which applies the same set of random transformations.

In this case the expectation is that the patterns will eventually be learned from the input and the *two-pass model* will gradually improve the *base model* performance. Since it takes time the patterns to be learned, it is expect that the average gain will be less that in the previous test.

Following table summarizes the test results.

Table 2 : Pattern learning test results

	Precision	Recall	F1-score	Offset
Base Model	0.99	0.28	0.43	3.24
Two-Pass Model	0.97	0.34	0.50	3.40
Performance Gain	-2%	21%	**16%**	-5%

The test results confirmed the expectations; however the performance gain on the main metric *F1-score* was significantly lower than in previous case, which was unexpected.

Base Model performance

The purpose of this test is to determine the effect of the quality of the *base model* on the performance of the *two-pass algorithm*.

In this test I configured the *two-pass algorithm* to use the used the `ProxyROIExtractor` and the `ProxyScrewDetector`. I run a series of tests and gradually improving the quality of the `ProxyROIExtractor` and `ProxyScrewDetector`. In each series I generated 100 scenes over three base patterns (triangle, rectangle and hexagon) by applying random translation rotation and screw centre offset.

As a primary measure of the performance I used the *F1-score* and compared the values of the *base-model* against the *two-pass model*.

Following table summarizes the test results.

Table 3 : Base model quality effect on performance - test results

ProxyROIExtractor success rate[3]	ProxyScrewDetector success rate[3]	Base Model F1-score	Two Pass F1-score	Gain [%]
0.5	0.6	0.34	0.35	3
0.6	0.8	0.56	0.64	14
0.7	0.9	0.71	0.79	11

In all cases the *two-pass model* outperformed the *base model*, however lower quality of the *base model* decreased the *two-pass model* performance.

[3] I use the term *success rate* as a common name for both *recall* and *precision*

Chapter 5

Analysis

Questions

Going back to the test results from the previous section, we could ask the following questions.

- Why do we have negative *accuracy* and *precision* gain in best-case and *learning proposer* tests?
- Why the *learning proposer* gain was inferior to the one with pre-loaded *patterns*?
- Can the *two-pass model* yield a negative gain (perform worse than *base model*)?
- If negative gain is possible, how can we prevent it?

The rest of the chapter attempts to provide answers as well as further analysis on these questions.

Precision decrease

First, let look at the decrease in the *precision* metric. Why would *two-pass model* worsen the *precision* of the *base model*?

According to the definition, the *precision* is a ratio of true positives to all positives. The only way that ratio can decrease is if the *two-pass model* have introduced some new false positives.

$$Precision = \frac{True\ Positives}{True\ Positives\ +\ False\ Positives}$$

That could have happened if the *two-pass model* proposed some new *regions of interest* that have been incorrectly classified as screws.

A close inspection of the test scenes confirmed that hypothesis. The following image (fig.20) is an example of false positives introduced by the 2nd pass, where incorrect *pattern* has been matched and that lead to passing true negative ROI to the *detector* (the red squares).

Then some of those true negatives are actually incorrectly classified as screws (false positives) by the detector.

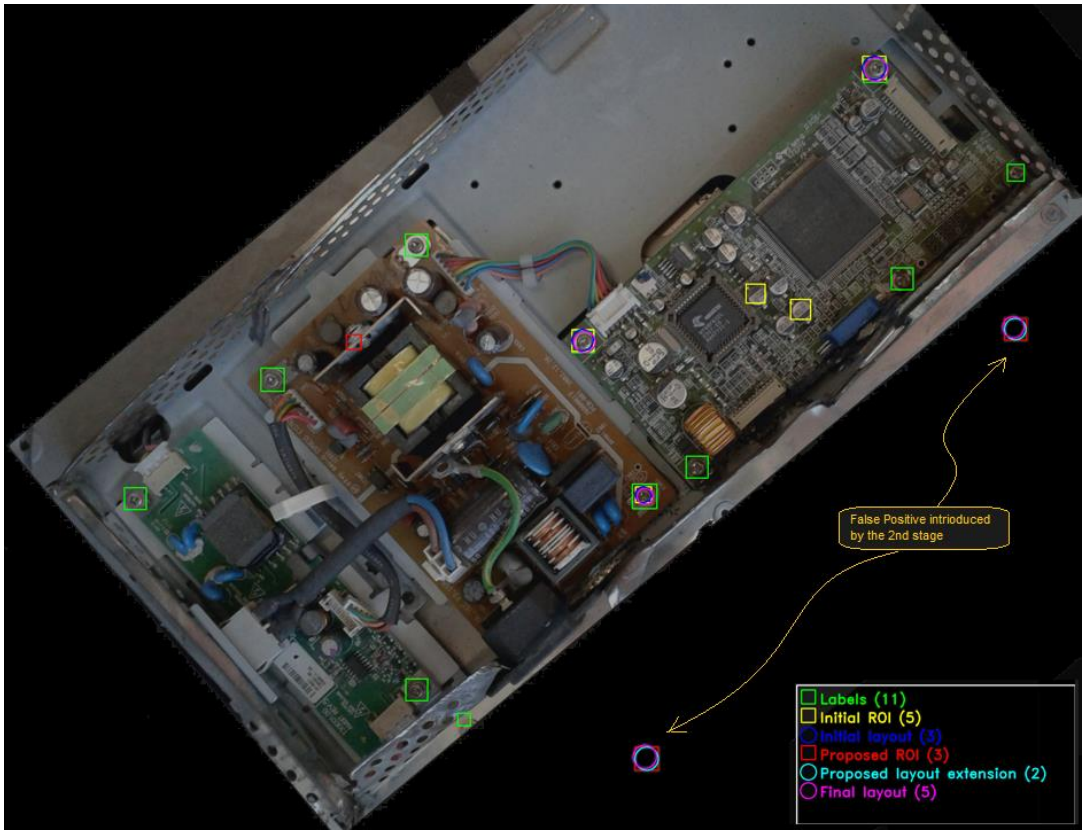


Figure 20 : Sample test scene showing false positives added by 2nd pass

Accuracy decrease

Now let look at the *accuracy* decrease. In our case, the *accuracy* is defined as the average offset of the detected screws centres from the actual (*labelled*) screw centre.

Why is the *accuracy* of the *two-pass model* worse than the *base model*?

Accuracy decrease can be explained by 2nd pass adding screws that are on average further away from the actual location than the *base model* ones

Let consider a linear *screw pattern* such as the one shown below.

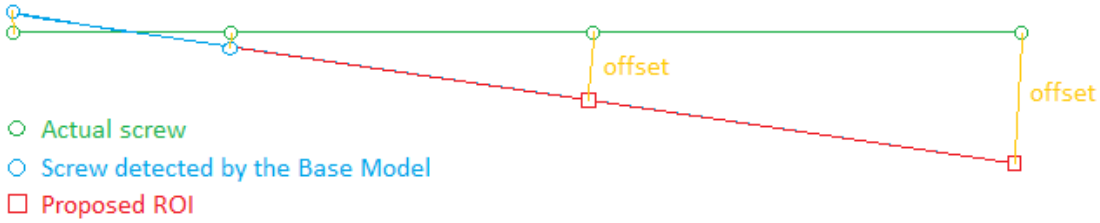


Figure 21 : Example of two-pass model increasing average offset

The *base model* has detected the left most pair of screws, and it did that with some error – an offset of the detected screw centres from the actual screw centres. When the *proposer* tried to match the detected screws with the memorised linear pattern, it used the detected screw locations, which contain the *base model* error. As we can see, the error in the proposed ROI for detecting the rest of the screws in the *pattern* is bigger the further away the ROI is from the *base model* detected screws. That could explain why the *base model* average screw offset is lower than the *two-pass model*.

Learning Proposer gain

Next, let look at why learning-proposer performance gain is significantly lower than the one with pre-loaded patterns.

Detailed analysis of the test scenes showed that for each of the patterns derived from same base image we have some screws that are never detected by the *base model*. Since we start with empty *pattern* store that means those screws never get a chance to become part of a memorised pattern. Since we evaluate the performance on a *labelled* test scenes that means that the *two-pass model* will never have chance to achieve maximum performance.

That phenomenon is most likely caused by a deficiency of the generated scenes approach. The base image could have some features that prevent the *Hough transform* or the CNN from detecting it as a screw. That however may not be the case in real world scenario where different instances of the same device would provide sufficient diversity so every screw location get chance to be detected by the *base model*.

Following is a sample *scene* showing the screws that were never proposed by the *base model*.

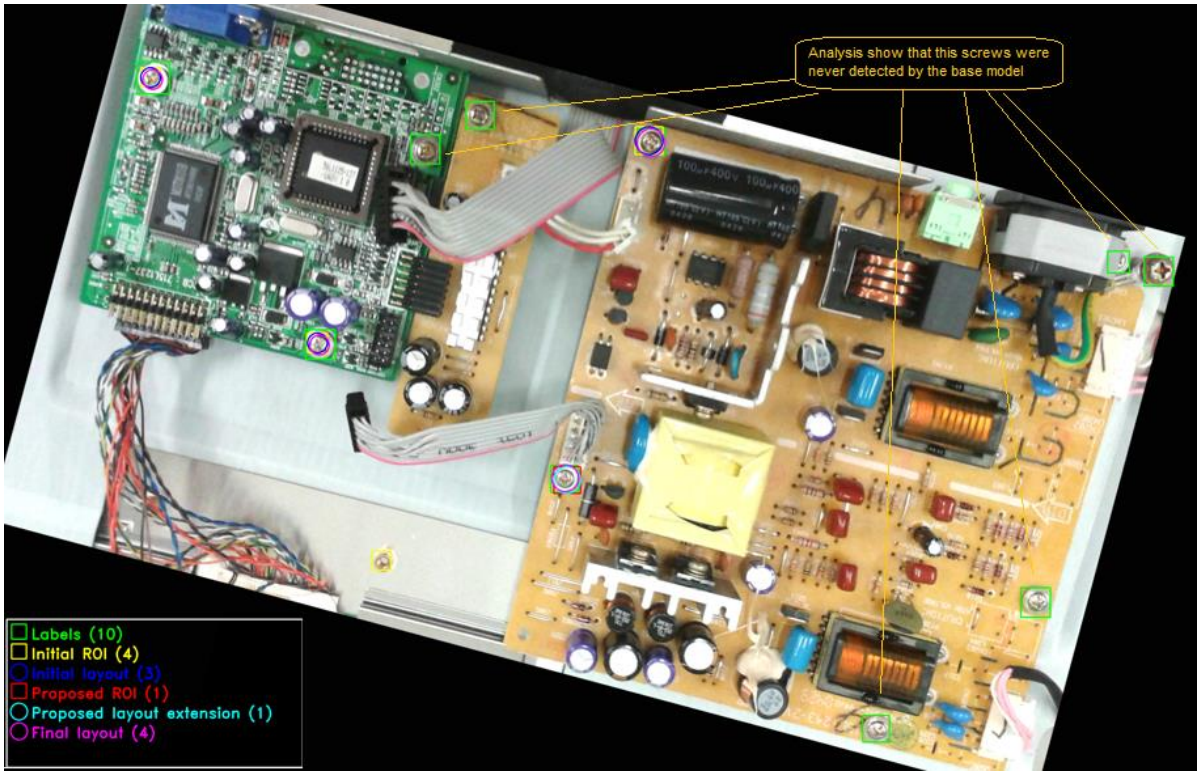


Figure 22 : Sample test scene showing *base-model* persistent false negattives

Negative Gain

Now let's look at the worst case scenario, when the *two-pass model* worsens the *base model* performance. Is this even possible?

Let consider the following hypothetical but plausible scenario. The *base model* detects three actual screws (true positives), incorrectly detects one capacitor as a screw (false positive), and dismisses six actual screws (false negatives). In this case the calculated F1- score would be 0.458:

True Positive (TP) = 3, False Positive (FP) = 1, False Negative (FN) = 6

$$Precision = \frac{TP}{TP+FP} = 0.75\,, \quad Recall = \frac{TP}{TP+FN} = 0.33$$

$$Base\ Model\ F1score = \frac{2\ x\ Precision\ .x\ Recall}{Precision + Recall} = \mathbf{0.458}$$

Now, let consider that the *proposer* matches the wrong *pattern* and proposes a 100 new *regions of interest*. The *base model detector* incorrectly classifies 10 of those as screws (false positives). Calculating the *F1-score* again we get 0.26, which is a lot less than the *base model*:

True Positive (TP) = 3, False Positive (FP) = 1+10 = 11, False Negative (FN) = 6

$$Precision = \frac{TP}{TP+FP} = 0.21\,, \quad Recall = \frac{TP}{TP+FN} = 0.33$$

$$TwoPass\ Model\ F1score = \frac{2 \times Precision \times Recall}{Precision + Recall} = \mathbf{0.26}$$

$$Gain = \frac{TwoPass\ Model\ F1score}{Base\ Model\ F1score} - 1 = \mathbf{-0.43}$$

The above result proves that in the worst case scenario it is possible that our *two-pass model* performs worse than the *base model*.

Preventing negative gain

How can we prevent the worse-case scenario?

If we get closer look, the issue happened when *proposer* proposed large number of additional locations. However, the *two-pass model* strength is detecting the false negatives of the *base model*. So, it would perform best when it proposes all of the *base model* false negatives.

If we know the *precision* and the *recall* of the *base model*, we can estimate the number of false negatives that it is expected to produce compared to the all positives that it yields.

$$Expected\ FN\ = TP\left(\frac{1}{Recall} - 1\right) = Precision\ \times\ Positives\ \times (\frac{1}{Recall} - 1)$$

Applying the above formula to the previous hypothetical example, we can estimate that the number of false positive would be six, which is significantly smaller than the 100 ROIs proposed by the incorrect *pattern* match.

$$Positives = TP + FP = 3 + 1 = 4,\ \ Precision = 0.75, Recall = 0.33$$

$$Expected\ FN = Precision\ \times Recall \times \left(\frac{1}{Recall} - 1\right) = \mathbf{6} \ll \mathbf{100}\ (proposed\ ROI)$$

In concussion, if we dismiss the patterns that result in extra ROI that are significantly different than the expected false negatives we could avoid the worst case scenario.

Chapter 6

Conclusion and Future Work

In conclusion, the project confirmed the expectation that the *two-pass model* can improve the *base model* performance. All of the test results confirmed that expectation. In the best-case scenario test we demonstrated that the *two-pass model* can significantly improve the *f1-score* and the *recall* of the *base model*. In the learning performance test, even though the *two-pass* model performance gain was impacted by the imperfections of the test data, it still demonstrated persistent improvement on the primary performance metric. Finally, the simulation tests using configurable quality of the *base model* components showed that the *two-pass* model can yield benefits over wide range of conditions.

Using theoretical analysis we have identified the worst case scenario, where the *two-pass model* worsens the *base model* performance. Using statistical analysis on the *base model recall* and *f1-score* we proposed a method to estimate the expected number of false negatives produced by the first pass. Based on that estimation, we have proposed a mitigation approach of the worst case scenario by ignoring the pattern matches that deviates significantly from the expectations.

Future development could explore different implementations of the base components. For example it could explore the use of *Generative AI* for implementing the *proposer*. The *generative AI* is applying already learned patterns to extrapolate the given input, which is basically the function of the *proposer* module.

Another avenue to explore would be using multiple cameras and this way combining different angles of the *disassembly scene*. That could help with improving the performance of the *base model*.

In the test so far the weakest link proved to be the *Hough transform* extractor module. Exploring options to improve the *Hough transform* performance or replacing it with alternative approach would be another way to improve the *base model* performance.

Bibliography

[1] C. Baldé et al., "The Global E-waste Monitor 2017", UNU, ITU, ISWA, 2017.

[2] "A New Circular Vision for Electronics Time for a Global Reboot", World Economic Forum, January 2019.

[3] W. Chen, "Towards a generic and robust system for the robotic disassembly of end-of-life electronics", Ph.D. Thesis, University of New South Wales, Kensington, NSW, Australia, 2017.

[4] N. DiFilippo and M. Jouaneh, "Using the Soar Cognitive Architecture to Remove Screws From Different Laptop Models", IEEE Transactions on Automation Science and Engineering, vol. 16, no. 2, pp. 767–780, 2019.

[5] G. Foo, S. Karaa, M. Pagnucco, "Screw detection for disassembly of electronic waste using reasoning and retraining of a deep learning model", 28th CIRP Conference on Life Cycle Engineering, 2021.

[6] G. Foo, "Robotic disassembly of waste electrical and electronic", PhD Thesis, UNSW 2022.

[7] G. Foo, S. Kara, and M. Pagnucco, "Challenges of robotic disassembly in practice", Procedia CIRP, vol. 105, 2022.

[8] E. Yildiz, F. Wörgötter, "DCNN-Based Screw Detection for Automated Disassembly Processes", in 2019 15th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS), 2019.

[9] E. Yildiz et al., "A Visual Intelligence Scheme for Hard Drive Disassembly in Automated Recycling Routines", in Proceedings of the International Conference on Robotics, Computer Vision and Intelligent Systems (ROBOVIS), Nov. 2020.

[10] S. Vongbunyong, “Applications of cognitive robotics in disassembly of products,” Ph.D. Thesis, University of New South Wales, Sydney, 2013.

[11] S. Mangold et al., "Vision-Based Screw Head Detection for Automated Disassembly for Remanufacturing", 29th CIRP Life Cycle Engineering Conference, 2022.

[12] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features”, in Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Kauai, HI, USA, 2001.

[13] Z. Zhao at al. "Object Detection with Deep Learning - A Review", IEEE Transactions on Neural Networks and Learning Systems, 2019.

[14] Open CV & AI, [Online]. Available: www.opencv.org.

[15] M. Ulrich, C. Steger, "Real-Time Object Recognition Using A Modified Generalized Hough Transform", Pattern Recognition, November 2003.

Appendix

Appendix A - Python Code Overview

The following is a high level overview of the program modules and their content.

Module	Description
data.py	Contains the definitions of the common data structures used across the program, such as `Point`, `Rectangle`, `Screw`, `Scene`, `Pattern`
interface.py	Defines main interfaces, such as: `Module`, `ROIExtractor`, `ScrewDetector`, `ROIProposer`, `Observer`, `SceneSource`.
extractor.py	Contains the *Hough transform* based implementation of the `ROIExtractor`.
detector.py	Provides a CNN (`Xception`) based implementation of the `ScrewDetector` interface.
proposer.py	Contains the `PatternMatchingProposer` implementation, as well as the `PatternStore` interface and its implementations.
observer.py	Contains the `Observer` interface implementations – `PerformanceObserver` and `ImageDecoratorObserver`
scenes.py	Contains the implementations of the `SceneSource` interface – `SceneReader` and `SceneGenerator`
main.py	Contains the *two-pass algorithm* implementation – the class `TwoPassScrewDetector`
proxy.py	Contains the proxy implementations of the `ROIExtractor` and `ROIProposer` as well as `ProxySceneSource`.
utility.py	Contains various utility functions used by the rest of the modules.

The following section describes the modules that contain main function and can be run via the command line.

main.py - Detects screws using the *two-pass algorithm* in the specified directories of input scenes.

Argument	Description
-scenes <scenesDir> ...	List of directories containing *disassembly scene* images
[-output <outputDir>]	Optional output directory to store the annotated scenes
[-generate <count>]	Optional. Number of scenes to generate
[-learn]	Optional. If present learning *pattern* store will be used. Otherwise the input scenes must contain labelling info.
-weights <weight.h5>	The Xception CNN weights for the ROI detector

Example:

```
python main.py
  -scenes /var/repo/LayoutB1
  -output /tmp/LayoutB1
  -generate 100
  -learn
  -weights /var/weights/xception-final.h5
```

proxy.py – Runs a simulation with specified extractor and *detector precision* and *recall*.

Argument	Description
-count <scenes>	Number of proxy scenes to generate
-ep <extractor-precision>	Extractor *precision* as decimal between 0 and 1
[-er <extractor-recall>]	Optional extractor *recall*. If missing the *recall* will be same as the *precision*.
-dp <detector-*precision*>	Detector *precision* as decimal between 0 and 1
[-dr <detector-recall>]	Optional *detector* recall. If missed, the *precision* will be used as *recall*.
[-learn]	Optional. If specified the *learning proposer* will be used

Example: `python proxy.py -count 100 -ep 0.7 -dp 0.9`

detector.py – Evaluates the CNN *detector* performance over specified set of ROI images or trains the CNN.

Argument	Description
-screw <imageDir>	Directory with images of screws
-hole <imageDir>	Directory with images of holes
-other <imageDir>	Directory with images of neither screw or hole
-weights <weights.h5>	CNN weights file location. If missing will trigger learning mode.

Example:
```
python detector.py -screw ~/screws -hole ~/holes \
    -other ~/other -weights ~/weights.h5
```

extractor-py – Evaluates the Hough transform extractor over specified set of *labelled* scenes.

Argument	Description
-scenes <scenesDir> ...	List of directories containing *labelled disassembly scenes*.
[-output <outputDir>]	Optional output directory to store the annotated scenes
[-optimise]	Optional. If specified will try to optimize the *Hough transform* parameters to maximize the *f1-score*

Example: `python extractor.py -scenes ~/LayoutA -output /tmp`

Appendix B - Practical Considerations

As mentioned in the chapter four, the patterns used by the `PatternMatchingProposer` are geometrical representation of the actual devices *screw layouts*. This means that the following calibrations need to be performed when setting up a real system.

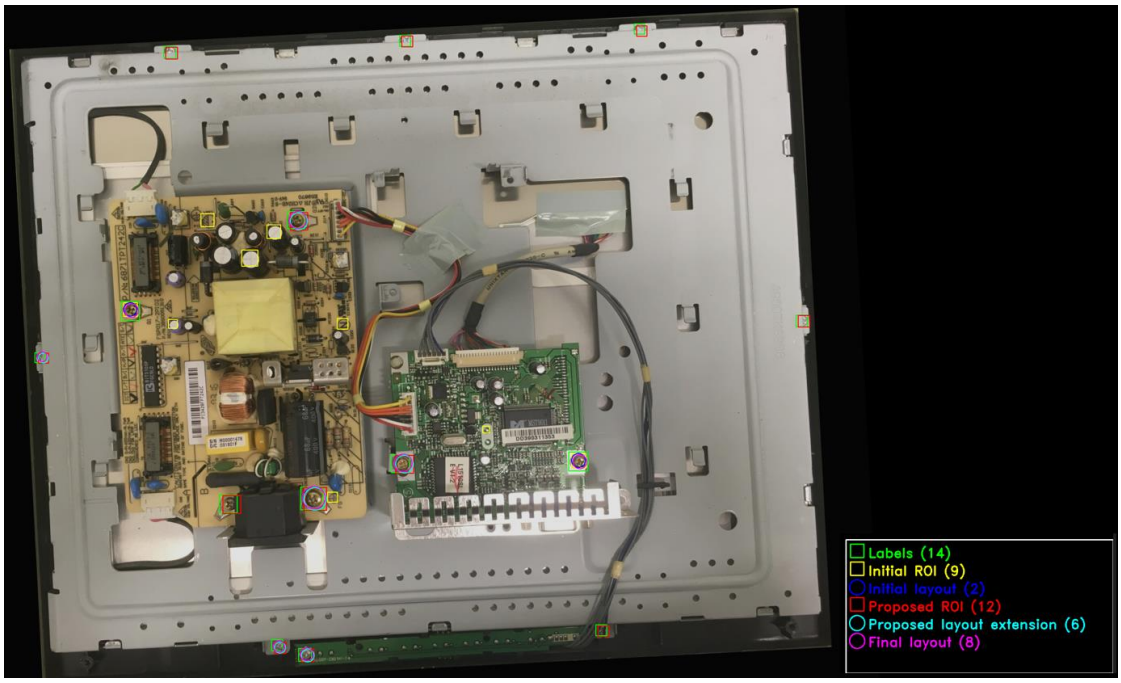
Removing the camera perspective is necessary in order to avoid changes of the location and orientation of the same device on the workbench to result in different *screw layouts*. The *Open-CV* library provides a standard transformation function (`warpTransform`) for removing the image perspective; however it requires a transformation matrix to be calculated in advance by calling the `getPerspectiveTransform` and providing a four pair of points showing mapping the input image to the desired target image.

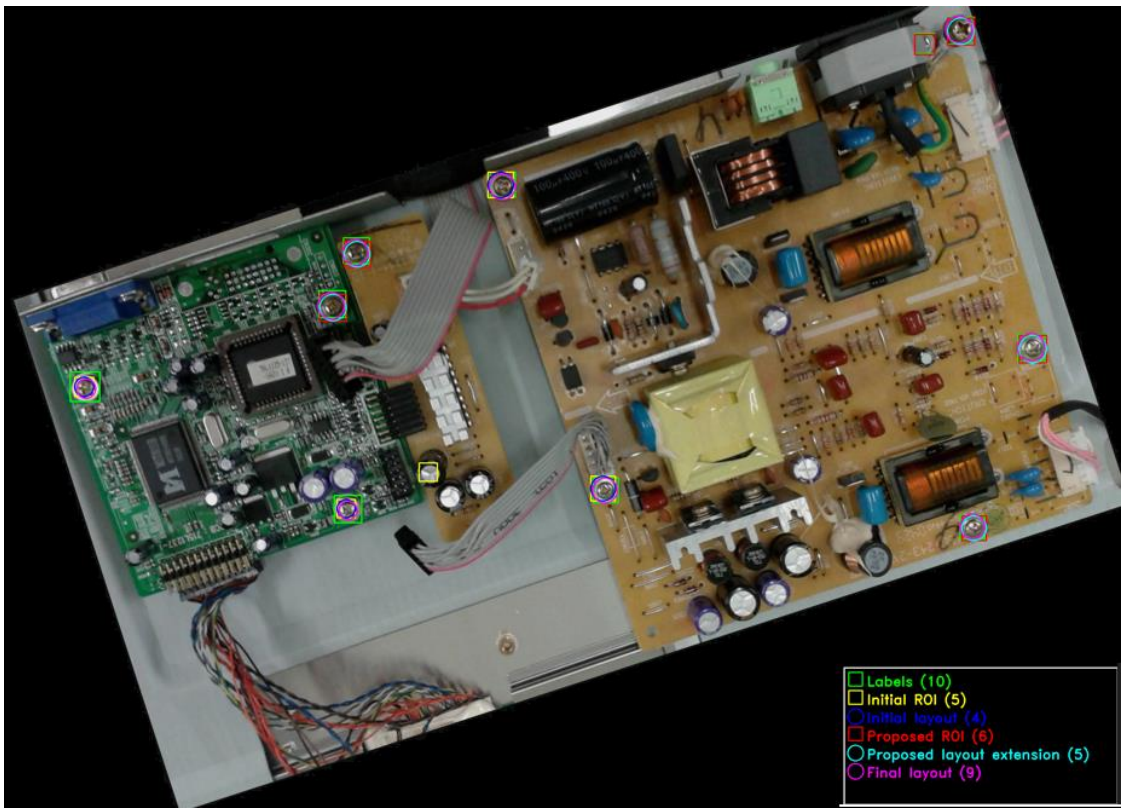
Calculating the pixel / mm ratio is necessary to accommodate changes in the camera distance from the work bench or changing the camera resolution.

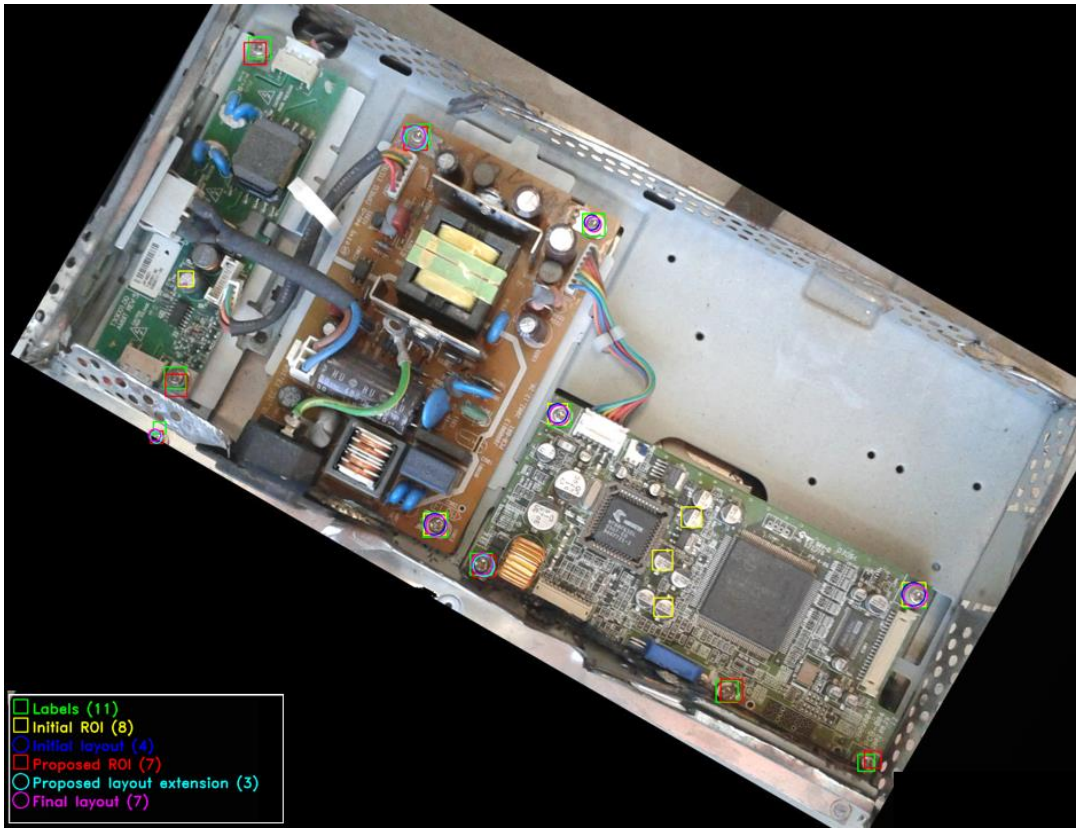
Taking image of the empty workbench is necessary to allow extracting the device being disassembled form the image by applying binary segmentation. Such extraction would prevent the system from detecting screws (false positives) which are outside of the boundaries of the disassembled device.

The above transformations will have to be coded and executed as part of the image pre-processing, which can happen in the `SceneReader` class.

Appendix C – Sample preloaded patterns test scenes






Labels (11)
Initial ROI (8)
Initial layout (4)
Proposed ROI (7)
Proposed layout extension (3)
Final layout (7)