Gerd Berger, BSc

Design and Implementation of a Test-Pattern Generator for Printhead Adjustment of Digital Inkjet Printers

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology
Institute for Technical Informatics

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. Georg Klima
(Durst Phototechnik Digital Technology GmbH)

Graz, November 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Kurzfassung

Die Durst Phototechnik Digital Technology GmbH ist eine österreichische Firma aus Lienz, Osttirol, die sich auf digitale Tintenstrahldrucksysteme spezialisiert hat. Ihr Portfolio reicht von ihrem Kern-Segment, dem Großformatdruck über Wellpappe- und Textil- bis hin zu Label- und sogar Keramikdruck.

Ähnlich wie bei einfachen Bürodruckern müssen auch bei den Industriedrucksystemen die Druckköpfe adjustiert werden. Zu diesem Zweck werden spezielle Testmuster gedruckt welche Ungenauigkeiten bei der Ausrichtung zeigen welche dann von Servicemitarbeitern behoben werden.

In dieser Arbeit wird ein System entworfen und implementiert, welches das Erzeugen dieser Testmuster automatisiert und wiederkehrende Aufgaben bei der Druckkopfjustage beschleunigt. Der erste naive Algorithmus zum Exportieren der Testmuster war nicht performant und wurde deshalb durch einen Plane-Sweep Algorithmus ersetzt, welcher später durch Multithreading verbessert wurde.

Abstract

The Durst Phototechnik Digital Technology GmbH is an Austrian company located in Lienz, East Tyrol, specialized in digital inkjet printing systems. Their portfolio range from their main segment, the large format printing over corrugated and textile to label and even ceramics.

Similar to simple office printers, the printheads of those industry printing systems must also be adjusted. For this purpose, special test patterns are printed to show inaccuracies in the adjustment which are eliminated by the service staff.

In this work, a system was designed and implemented to automate the generation of those test patterns and speed up the recurring tasks of printhead adjustment. The first naive algorithm for exporting the test patterns was not performant and therefore replaced by a plane-sweep algorithm which was later improved with multithreading.

Acknowledgments

This thesis was created in 2018 at the Institute for Technical Informatics at Graz University of Technology and in cooperation with Durst Phototechnik Digital Technology GmbH in Lienz.

First, I would like to thank Durst Phototechnik Digital Technology GmbH, especially DI Wolfgang Knotz for the great opportunity to do this work as well as DI Georg Klima for his professional technical assistance.

I also would like to thank Prof. Christian Steger for his supervision not only for this thesis but also previous work.

Furthermore, I would like to thank my parents and family not only for their financial but also for their mental support during my life. Without their support, I would not have been able to pursue an academic degree.

Last but not least, my thanks go out to my friends for their support and motivation during the whole studies.

Gerd Berger Graz, November 2018

Contents

Abstract **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 State of the Art 3
1.3 Objective 3
1.4 Structure of the Thesis 4

2 Literature Research **6**
2.1 Keep It Simple, Stupid 6
2.2 Dynamic Class Loading 6
2.3 File Formats 7
2.4 TIFF 8
2.5 Scanline Principle and Plane-Sweep Algorithm 11
2.6 Design Patterns Revisited 12
2.6.1 Serializer 13
2.6.2 Factory Pattern 14
2.6.3 Prototype 15
2.6.4 Singleton 16
2.6.5 Composite 16
2.6.6 Strategy 17
2.6.7 Visitor 18
2.6.8 Whole-Part 20
2.6.9 Publisher-Subscriber aka Observer 20
2.6.10 Concluding Remarks on Design Patterns 20
2.7 PImpl Idiom 23
2.8 Smart Pointers 23
2.9 The Property System 24
2.10 Rule of Three/Five/Zero 24

3 Design **25**
3.1 Specification . . . 25
3.2 User Stories . . . 26
3.2.1 User Stories for Users . . . 26
3.2.2 User Stories for Developers . . . 30
3.3 GUI . . . 33
3.4 System Architecture . . . 34
3.5 Printer Configuration File . . . 36
3.6 Pattern Definition File . . . 38

4 Implementation **39**
4.1 Development Workflow . . . 39
4.2 UI Classes . . . 40
4.2.1 MainWindow . . . 40
4.2.2 PatternSection . . . 41
4.2.3 PatternDefinitionSection . . . 41
4.2.4 PropertiesSection . . . 42
4.3 Core Classes . . . 43
4.3.1 FileIO . . . 45
4.3.2 ISerializable . . . 45
4.3.3 Pattern Definition . . . 46
4.3.4 Printer . . . 48
4.3.5 Print Primitives . . . 49
4.3.6 Test Patterns . . . 53
4.3.7 Exporter . . . 62
4.4 Main Function and Commandline Execution . . . 66
4.5 Case Study . . . 66

5 Use Cases **71**
5.1 Adding a new Printer . . . 71
5.2 Adding a Property to a Test Pattern . . . 73
5.3 Creating a new Print Primitive . . . 75
5.4 Creating a new Test Pattern . . . 76

6 Conclusion and Future Work **78**

Bibliography **80**

List of Figures

1.1 P5 250 HS . . . 2
1.2 Printhead of the Rho 1xxx Series HS . . . 3

2.1 Class Diagram of Serializable Interface . . . 13
2.2 Class Diagram of Factory Pattern . . . 14
2.3 Class Diagram of Prototype Pattern . . . 15
2.4 Class Diagram of Singleton Pattern . . . 16
2.5 Class Diagram of Composite Pattern . . . 17
2.6 Class Diagram of Strategy Pattern . . . 18
2.7 Class Diagram of Visitor Pattern . . . 19
2.8 Class Diagram of Whole-Part Pattern . . . 21
2.9 Class Diagram of Observer Pattern . . . 21

3.1 User Stories . . . 27
3.2 GUI of TestPatternGenerator . . . 33
3.3 Components of TestPatternGenerator . . . 34
3.4 Overview of UI Classes . . . 35
3.5 Overview of Core Classes . . . 35
3.6 Printer Configuration for P5 250 HS . . . 37
3.7 Pattern Definition of SlotOffset for P5 250 HS . . . 38

4.1 Class diagram of MainWindow . . . 40
4.2 Class diagram of PatternSection . . . 41
4.3 Class diagram of PatternDefinitionSection . . . 42
4.4 Class diagram of PropertiesSection . . . 42
4.5 Core Classes . . . 44
4.6 Class diagram of FileIO . . . 45
4.7 Class diagram of ISerializable . . . 46
4.8 Class diagram of PatternDefinition . . . 47

4.9 Class diagram of Printer and associated classes 48
4.10 Class diagram of print primitives 50
4.11 Class diagram of patterns 54
4.12 NozzleWarmup pattern 56
4.13 NozzleWarmup pattern printed with P5 250 HS 56
4.14 Adjusting Printhead Rotation 57
4.15 Rotation pattern 58
4.16 RotationPattern printed with P5 250 HS 59
4.17 RotationPattern printed with Rho 13xx 60
4.18 SlotOffset 61
4.19 SlotOffset printed 62
4.20 SlotOffsetPattern printed zoomed 63
4.21 Exporter 63
4.22 Sequence Diagram of Export Workflow 65
4.23 Folder Structure of Exported Files 67

5.1 Slot Arrangement Rho 163 TS 72
5.2 Printhead Configuration Rho 163 TS 72
5.3 Printer Configuration of Rho 163 TS 74
5.4 SlotOffsetPattern original and with 1px lines 75
5.5 NozzleTest Pattern 77

List of Tables

2.1 Used TIFF Tags 10

3.1 Versions of Used Technologies 26

4.1 Comparison of Algorithms on Laptop 69
4.2 Comparison of Algorithms on Workstation 70

1 Introduction

Like normal home and office printers, industrial printers also need adjustments after delivery e.g. after changing the printer cartridge. While in the home and office area one predefined test site is printed and some properties are set in the printer software, the process for industrial printers is more complex.

The digital inkjet printers developed by the Austrian company Durst Phototechnik Digital Technology GmbH which is part of the Durst Phototechnik AG from Brixen, Italy[1] are worldwide leading systems. They are separated into five segments:

- Large Format Print
- Label
- Ceramics
- Textile
- Corrugated

For each segment there exist various different printer systems. The newest development from Durst, a large format printer named P5 250 HS is shown in Figure 1.1.

1.1 Motivation

The printers differ not only in terms of the material on which it is printed but also if they are one pass or multi pass printers, the color and type of the inks and even how the drying of the ink works. They have a different

[1] www.durst-group.com

Figure 1.1: P5 250 HS. Image taken from [Dur18a].

amount and layout of more than one hundred printheads with up to four different colors on each.

A printhead from the Rho 10xx and Rho 13xx series is illustrated in Figure 1.2 where 1 and 2 are adjustment screws and 3 are fixing screws. On the right, the layout of the slots is shown. As can be seen, for this printhead two colors (cyan and black) are used.

What they have in common for all printers is that after replacing old printheads, the new ones have to be adjusted mechanically and through software. For this purpose there exist test patterns for the different adjustment tasks which are, e.g., rotation or slot offset. These test patterns are basically equal, but differ in some details like the number of printheads and their layout.

The creation of such adjustment patterns is very time consuming and error-prone. Since it has to be done for each new printer of each category, it would be a major benefit to have the possibility to generate those patterns.

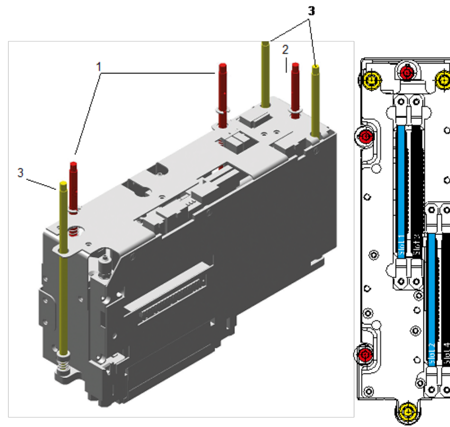


Figure 1.2: Printhead of the Rho 1xxx Series HS. Image taken from [Dur18b].

1.2 State of the Art

Creating test patters is a very time consuming and error-prone task which is done using Adobe Photoshop[2]. The technician who wants to create test patterns for a new printer must have a precise knowledge of the layout of the printheads. Usually, he or she takes the pattern for a similar printer and modifies it. Most patterns consist of a lot of individual lines with a width of one pixel. The space between these lines also differs but always have to be exact. Once the basic pattern for one printhead is created, the work consists of a lot of copy and paste work. One Printer could easily have 128 individual printheads. Finally, labels must be added to make it possible to identify the printheads and other parameters. Additional to the images, the printer also needs a configuration file, where the media advance is specified.

1.3 Objective

In this work, a software should be designed and implemented which allows the generation of test patterns. It should be possible to create a test pattern,

[2] `https://www.adobe.com/at/products/photoshop.html`

modify some properties and export it to be printed on various systems. The TestPatternGenerator should help different departments like customer service and quality management to save time by automating the test pattern generation and also by reducing possible errors in the test pattern.

The software should not only provide a Graphical User Interface (GUI) to create these test patterns but also be able to be called from another tool, e.g., the printer software on the workstation.

Target platforms are Microsoft Windows[3] (desktop PCs of users) as well as Red Hat Enterprise Linux[4] (workstations of printers).

The main challenges to address are:

- Usability: The software should be easy to use not only for technicians but also for customers.
- Flexibility: One tool should be enough to create test patterns for all types of printers.
- Expandability: Adding a new printer should not require a new software version. Adding new patterns should not afford big changes in the software.

In this thesis, we address these challenges and explain the design and implementation of the TestPatternGenerator.

1.4 Structure of the Thesis

In **Chapter 2** used technologies, algorithms and design patterns are explained.

In **Chapter 3** the design of the solution is shown. The technological requirements, the user stories and the overall system architecture are discussed. At the end of the chapter, the GUI is described.

The implementation is described in **Chapter 4**. As the main part, the different implemented classes, especially the print primitives and the test patterns

[3]https://www.microsoft.com/en-us/windows

[4]https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

are explained. Since an iterative development process was chosen, the chapter concludes with a comparison between the different implementation approaches.

The following **Chapter 5** deals with some use cases which could occur more frequent. The most important ones, adding a new printer and a new test pattern are explained as well as adding a property to an existing test pattern and creating a new print primitive.

The thesis closes with **Chapter 6** in which the whole work is recapitulated and an outlook to future work is given.

2 Literature Research

In this chapter, we describe concepts and technologies used for the design and the implementation of the TestPatternGenerator. It starts with the Keep it Simple, Stupid (KISS) principle, over dynamic class loading and file formats for serialization. After that, TIFF itself and how it is used is explained, followed by a section about the scanline principle and plane-sweep algorithm, followed by a repetition of the most important design patterns for this work. The chapter closes with an overview over the PImpl Idiom, smart pointers, Qt's property system and the rule of three/five/zero.

2.1 Keep It Simple, Stupid

The fundamental principle, on which the whole design and implementation is based on, is KISS. An overview of this principle, variations and alternatives are given in [Fou18]. The basic thought of KISS applied to the TestPatternGenerator is that the tasks, the user needs to fulfill should be done as simple as possible. The main functionality of the tool is to generate test patterns for printers. Therefore the GUI should provide the possibility to select a test pattern and a printer to export it. Additional settings to adjust the test pattern should be as little as possible.

2.2 Dynamic Class Loading

Since the software should be as flexible as possible, loading classes dynamically should also be considered. The part where it could be used is for the different test patterns. Norton describes in his article ([Nor00]) how to use

the concept of polymorphism and dynamic class loading for adding and working with classes loaded at runtime, respectively.
This approach could provide an immense improvement in flexibility and expandability for a software because adding a class, e.g., a new test pattern, would not require a complete rebuild. Since the test pattern also depends on the printer configuration, it could occur that for a new test pattern, also the configuration files of the printers have to be changed. For example by adding a new reference property. Therefore it could not be guaranteed that a certain version of the software could handle a newly added test pattern. Also, it is not very likely that new patterns are needed very often after all basic patterns are implemented. Due to these reasons, the concept of dynamic class loading was not implemented in this software. It is also considered to implement a so-called generic test pattern, where the user could define his own test pattern consisting of print primitives. This could be serialized and deserialized and would make the dynamic class loading obsolete.

2.3 File Formats

There exist a wide range of different file formats suitable for serialization. The most common ones are XML, JSON and YAML. A documentation of these file formats can be found at [Bra+08], [Int17] and [BEN09].

Keshavarzi and Bayer [KB11] give a short overview of these formats and compare them. They state that JSON and YAML are simpler and also shorter and therefore repress XML more and more.

Nurseitov et al. [Nur+09] also compare XML and JSON. They made a case study where they created a simple client-server application where XML-encoded and JSON-encoded objects are sent from the client to the server and measure the transmission time and resource utilization. The results of the two tested scenarios were that the transmission of objects using JSON is up about 50 times faster than using XML.

Although JSON would be the faster technology with more compact files, the decision was made to use XML. The main reason for this is that we aim to

follow the KISS principle and to keep the number of different technologies as small as possible. Two other tool for test-patterns already works with XML and a future version of the TestPatternGenerator should be able to work with it.

2.4 TIFF

The Tagged Image File Format (TIFF) was created by Aldus Corporation in 1986 which was later purchased by Adobe Systems Incorporated. A complete specification of TIFF can be found at [Ass92] where also the structure and other details about the TIFF are explained. Worth mentioning here is just that a TIFF has a header containing the byte order, "An arbitrary but carefully chosen number (42) that further identifies the file as a TIFF file." ([Ass92, p. 13]) and the offset to the first image file directory. Additionally, a complete tag-reference of the TIFF could be found at [SYS18].

For writing the files, LibTIFF in the latest version 4.0.9 is used which could be downloaded from `https://download.osgeo.org/libtiff/`. Examples, how the library could be used are provided under [Sys17]. Since the dimensions of the resulting test-images are very big, the files could not be written at once and therefore they must be written line by line. This is called scanline-based image I/O. The creation of a file follows the following steps:

1. **Open File:**
   For opening a tiff file, TIFFOpen is used, which takes two arguments, a filename and a mode parameter for read, write or append. It returns a handle which is used in the subsequent steps or NULL if opening the file failed.
2. **Write Tags:**
   TIFFSetField is used to write the different tags into the header directory of the file. This method takes the handle, the name of the tag and one or two parameters for the values. It returns 1 if the operation was successful or 0 if not.
3. **Write Lines:**
   To write the lines into the file, TIFFWriteScanline is used. This method takes the handle, the buffer and the row number as parameter. It returns 1 if the write was successful or -1 if an error was detected.
4. **Close File:**
   TIFFClose closes the file specified by the handle.

The tags which are needed for the printer to interpret the images correctly are described in Table 2.1. The descriptions and types are taken from [SYS18]. Note that all tags except TIFFTAG_BITSPERSAMPLE and TIFFTAG_EXTRASAMPLES allow one value. TIFFTAG_BITSPERSAMPLE defines the number of bits per component, e.g., an RGB image could have a different number of bits for each color component, but since LibTIFF does not support it, only one value for all components is set. TIFFTAG_EXTRASAMPLES allows N values where the first specifies the number of extra samples.

Tag	Description	Type	Values
TIFFTAG_IMAGEWIDTH	The number of columns in the image, i.e., the number of pixels per row.	SHORT or LONG	width of the test pattern
TIFFTAG_IMAGELENGTH	The number of rows of pixels in the image.	SHORT or LONG	height of the test pattern
TIFFTAG_SAMPLESPERPIXEL	The number of components per pixel.	SHORT	8
TIFFTAG_BITSPERSAMPLE	Number of bits per component.	SHORT	8
TIFFTAG_PLANARCONFIG	How the components of each pixel are stored.	SHORT	PLANARCONFIG_CONTIG = 1
TIFFTAG_COMPRESSION	Compression scheme used on the image data.	SHORT	COMPRESSION_LZW = 5
TIFFTAG_ORIENTATION	The orientation of the image with respect to the rows and columns.	SHORT	ORIENTATION_TOPLEFT = 1
TIFFTAG_RESOLUTIONUNIT	The unit of measurement for XResolution and YResolution.	SHORT	RESUNIT_INCH = 2
TIFFTAG_XRESOLUTION	The number of pixels per ResolutionUnit in the ImageWidth direction.	RATIONAL	x-resolution of printer
TIFFTAG_YRESOLUTION	The number of pixels per ResolutionUnit in the ImageLength direction.	RATIONAL	y-resolution of printer
TIFFTAG_PHOTOMETRIC	The color space of the image data.	SHORT	PHOTOMETRIC_SEPARATED = 5
TIFFTAG_ROWSPERSTRIP	The number of rows per strip.	SHORT or LONG	1
TIFFTAG_EXTRASAMPLES	Description of extra components.	SHORT	four extra samples with value EXTRASAMPLE_UNSPECIFIED = 0

Table 2.1: Used TIFF Tags

2.5 Scanline Principle and Plane-Sweep Algorithm

Using the scanline principle on a given set of data means that the algorithm goes through the input only once and it only takes the currently relevant elements into account. The plane-sweep algorithm uses this principle to find intersections of lines or planes.

As Bentley shows in his book Programming Pearls [Ben86, pp. 69 sqq.] in Chapter 7, the choice of the algorithm is essential for the runtime of the program. Bentley explains this with different algorithms to solve the subset sum problem. The fourth algorithm, using the scanline principle, iterates over the array with n elements only once and has a time complexity of O(n). Bentley compares this algorithm with other, naive algorithms and also shows that constant factors, e.g., by using better hardware for worse algorithms, did not pay off for a bigger problem size.

Nievergelt and Preparata already explained in 1982 in their article "Plane-Sweep Algorithms for Intersecting Geometric Figures" [NP82] the increasing importance of algorithms for computer-aided design. They explain the plane-sweep algorithm on two examples, the region-finding algorithm and an algorithm for the intersection of convex maps.

As described in Section 4.3.5, the test patterns consist of a big number of print primitives. To write the data of the different print primitives to the output file, they have to be iterated. In the previous section, it is described that the TIFF files are written line after line. Iterating over all primitives in each line is a big overhead since not all print primitives are relevant for all lines. To reduce the effort for retrieving the individual data, a modification of the plane-sweep algorithm was implemented. The basic idea for using this algorithm came from Aichholzer and his lecture "Entwurf & Analyse von Algorithmen" ([Aic15]). In Chapter 5.3 he explains how it is used to find intersecting line segments.

The algorithm maintains two data structures. The begin-, end- and intersection-points of the lines are called events. The X-structure contains the X-coordinates of all known events which are not yet reached by the scanline. Additionally, the information if it is a start-, end- or intersection-point must be stored. This structure must be sorted in ascending order all the time. For example,

in the beginning it contains the X-coordinates of the start- and end-points of all line segments. The Y-structure contains all segments which are hit by the scanline. The segments are sorted by their Y-coordinates.

Algorithm:

1. Get minimum m from X and remove it
2. If m is a start point, add the segment to Y
   if m is an end point, remove it from Y
   if it is an intersection point, exchange the two segments in Y
3. For all new neighboring segments, check if they intersect and if so, add the intersection point to X and report the intersection

The plane-sweep algorithm is event-driven and has a complexity of $O((n + k) \log(n))$ in time and $O(n + k)$ in space where n is the number of line-segments and k is the number of intersections. For a detailed analysis check [Aic15].

The Encyclopedia of GIS is a reference of topics which concern geographic information systems. In Volume 2, Wood and Kim [**Wood:2015**]) give an overview of the plane-sweep algorithm. There are a lot of application areas like in robotics and motion sensing, computer graphics and in spatial databases. The example in the book deals with the intersection of geometric shapes represented by their minimum bounding rectangles. These minimum bounding rectangles are also separated in two sets and only intersections from different sets are considered.

The advantage of using the plane-sweep approach in this work is shown in a case study in Subsection 4.5, where the export duration of the test patterns for the three supported printers using different algorithms are compared.

2.6 Design Patterns Revisited

This section gives a brief overview of all implemented design patterns and those which were considered. It is not the aim of this work to explain design patterns in detail, but the essence of some selected should be brought to mind. Additionally, the idea where and why the design pattern could be

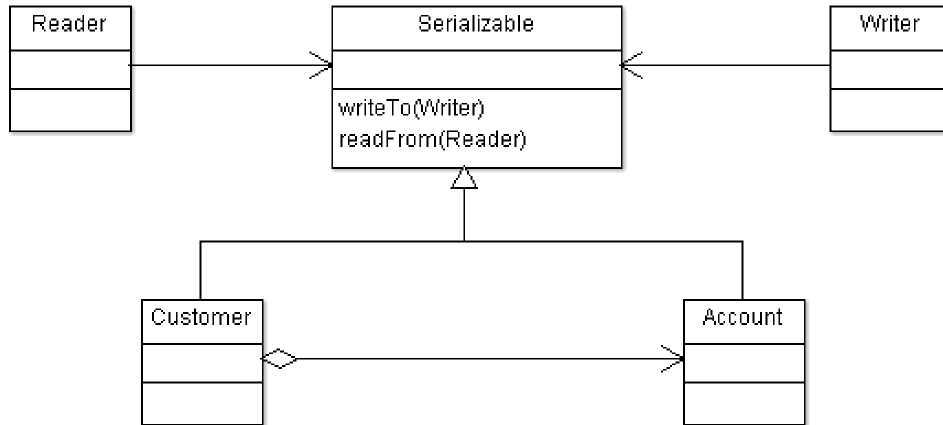


Figure 2.1: Class Diagram of Serializable Interface. Image adapted from [Rie+97].

used is explained. More details about the actual implementation is given in the corresponding classes in Chapter 4.

2.6.1 Serializer

In the serializer design pattern ([Rie+97]) an interface is used to define methods for serializing and deserializing objects. To perform these tasks, a reader and a writer are used and each class implementing this interface is able to store and restore their data. The class diagram of the serializable interface is shown in Figure 2.1.

This design pattern is used throughout the whole software. It is used to load a `Printer` and save and load a `PatternDefinition`.

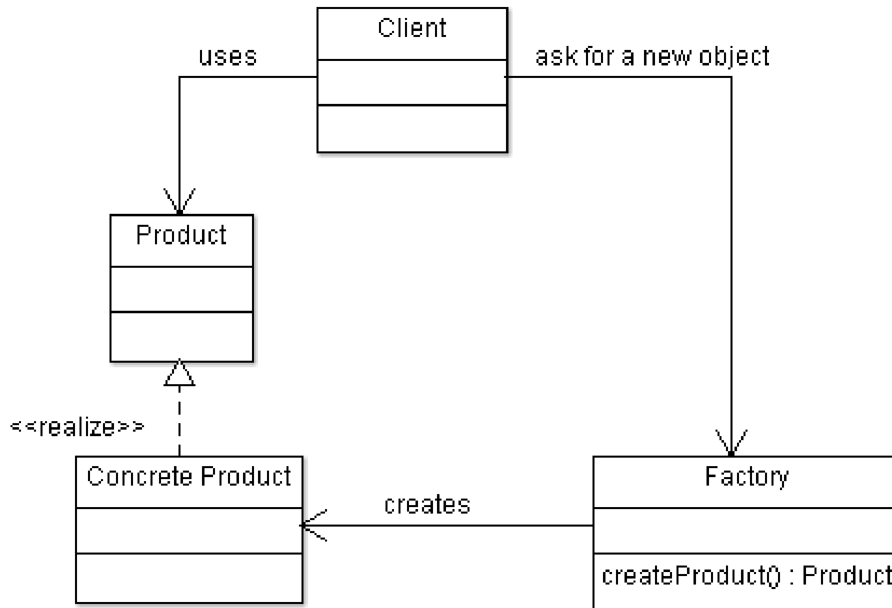


Figure 2.2: Class Diagram of Factory Pattern. Image adapted from [Des18].

2.6.2 Factory Pattern

The factory pattern is used for object creation. All products are in an inheritance hierarchy and derived from a virtual root-class. A factory class provides a method which is responsible for the object creation. Usually, it takes the type as a parameter and returns a concrete product as root-object. In Figure 2.2 the class diagram of this pattern is shown and additional information can be found at Object Oriented Design [Des18]. The advantage of this design pattern is that the effort of object creation is encapsulated within a factory class.

Similar to the factory pattern is the factory method defined by Gamma et al. [Gam+94, pp. 107 sqq.]. It is more general because the design explicitly allows sub-classing the factory and to change the behavior of the method

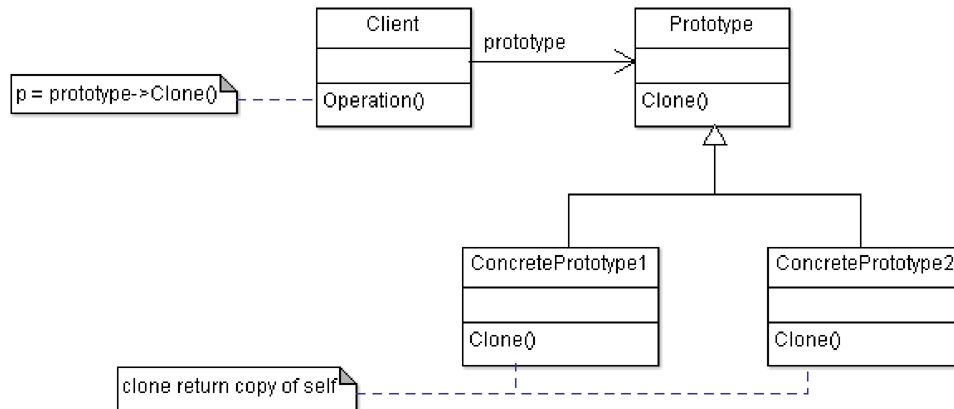


Figure 2.3: Class Diagram of Prototype Pattern. Image adapted from [Gam+94, p. 119].

for object creation. For the sake of completeness, also the abstract factory ([Gam+94, pp. 107 sqq.]) should be mentioned. Here also the factory classes are in an inheritance hierarchy and responsible to create a group of objects.

The creation of the test patterns is done using the factory pattern. It was also considered to implement a factory for creating the different `PrintPrimitives` but due to little benefit this idea was rejected.

2.6.3 Prototype

The next creational pattern is the prototype ([Gam+94, pp. 117 sqq.]) where all objects implement a clone method. If the client needs an object, it simply calls this method from a prototype. The big advantage of this design pattern becomes visible if the objects have a lot of properties and a lot of similar objects have to be created. The prototype has set the common properties and is just copied. One drawback is that implementing the clone method is error prone especially if during the implementation, the properties of the objects change a lot. The class diagram can be found in Figure 2.3.

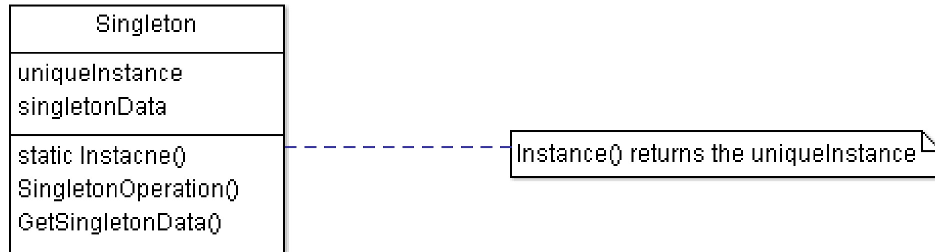


Figure 2.4: Class Diagram of Singleton Pattern. Image adapted from [Gam+94, p. 127].

The prototype pattern is implemented within the `PatternFactory` which has the advantage that the setup of all print primitives have only be done once.

2.6.4 Singleton

The last design pattern concerning object creation which is mentioned here is the singleton ([Gam+94, pp. 127 sqq.]). As the name says, there exists only one instance of a singleton. This is done by making the constructor of the class private and also hold a private instance of the class as a member. Additionally, it provides a method for retrieving this instance. This method checks if the private member is already set. If not, an object is created and set to it and afterwards, the instance is returned. Figure 2.4 shows the class diagram of the singleton.

The `PatternFactory` is implemented as a singleton which is necessary because of the usage of the prototype pattern.

2.6.5 Composite

The composite [Gam+94, pp. 163 sqq.] is a structural design pattern. The intention is to let the client use objects and compositions of objects uniformly.

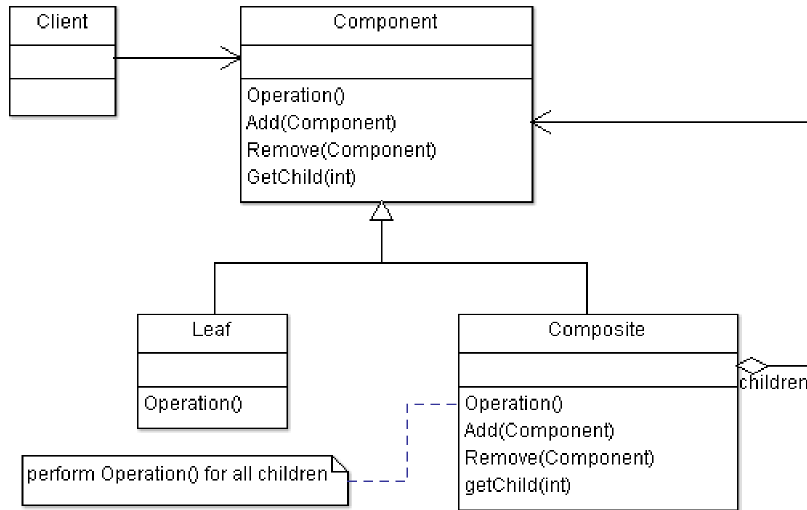


Figure 2.5: Class Diagram of Composite Pattern. Image adapted from [Gam+94, p. 164].

The part-whole hierarchy forms a tree structure. The class diagram of this pattern is shown in Figure 2.5.

The design pattern was considered to be implemented in the print primitives. The basic class is `PrintPrimitive`. Subclasses are among others `Line`, `Rectangle`, `Label` and `LineBlock` which consists of a set of separate lines. This approach was not implemented because it would not eliminate the problem of too many objects for a test pattern as explained in Subsection 4.3.5.

2.6.6 Strategy

The strategy is a behavioral design pattern [Gam+94, pp. 315 sqq.]. It is used to make algorithms interchangeable by defining a base strategy and derive concrete strategies. The class diagram can be found in Figure 2.6.

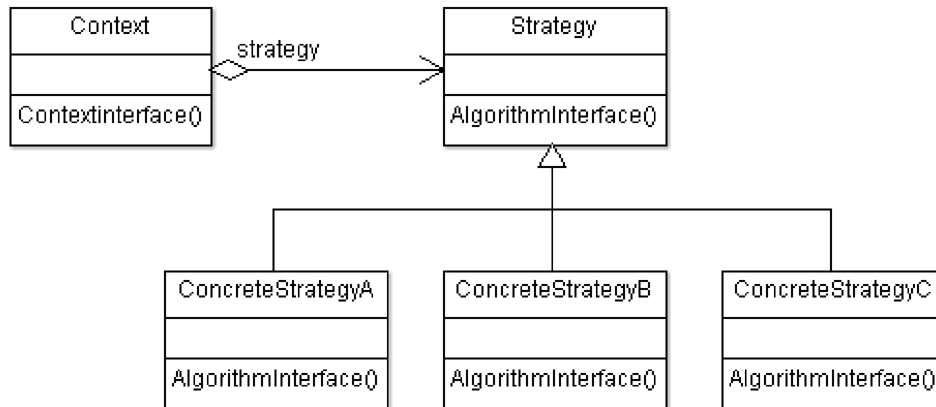


Figure 2.6: Class Diagram of Strategy Pattern. Image adapted from [Gam+94, p. 316].

This design pattern is used for the exporting functionality. We have a base `Exporter` class and several inherited exporters. In the first version only one, namely the `TiffExporter` was implemented.

2.6.7 Visitor

Another behavioral design-patter is the visitor ([Gam+94, pp. 331 sqq.]). Concrete visitors are derived from a base visitor class and operate on an object structure. The benefit of this design pattern is that the operations on the objects can be changed without changing the objects itself. Figure 2.7 shows the class diagram of the visitor pattern.

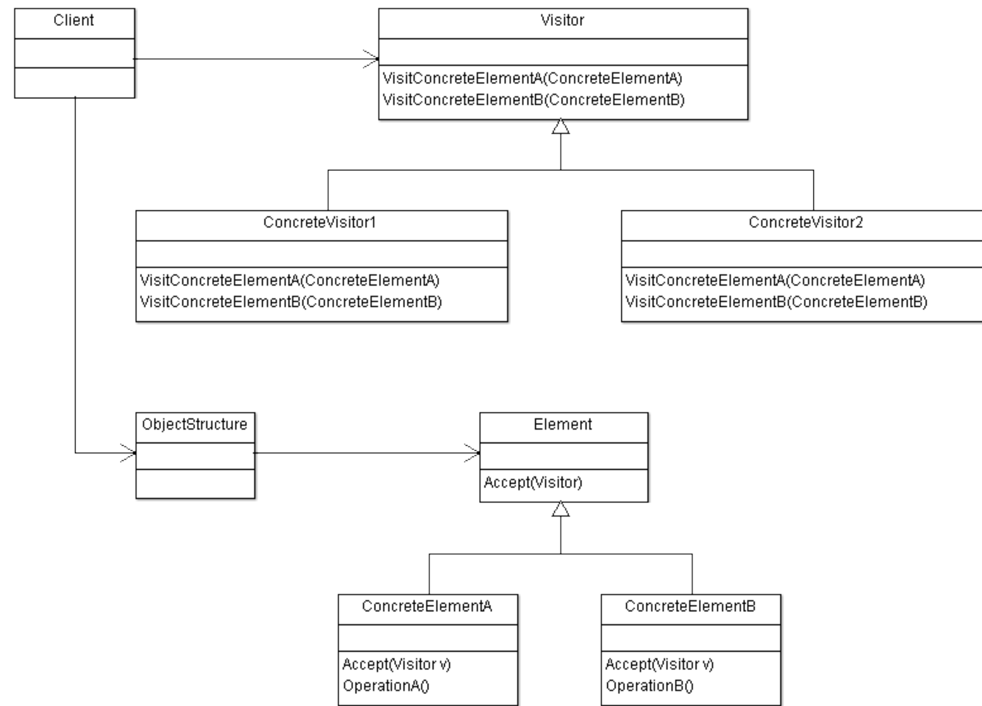


Figure 2.7: Class Diagram of Visitor Pattern. Image adapted from [Gam+94, p. 334].

Palsberg and Barry [PJ98] explain the visitor pattern and compare it with two other approaches (instanceof and type cast, and dedicated methods) using a Java example. They also demonstrate how to implement the Visitor "without relying on accept methods and without knowing all classes of the objects in advance." ([PJ98, p. 1]).

The idea was to implement the export functionality for the different print primitives of a test pattern using the visitor but since there will not be many new exporters, the cost-benefit ratio would not pay off.

2.6.8 Whole-Part

Buschmann et al. [Bus+96, pp. 225 sqq.] explain that Whole-Part uses structural decomposition to help structuring and to encapsulate parts of a unit. It also provides an interface to the rest of the system. For example, a geometric figure consist of individual lines, rectangle etc. To rotate it, only the rotate method of whole figure must be called. The rotation of the individual parts is done by the figure itself. The class diagram of this pattern can be found in Figure 2.8.

2.6.9 Publisher-Subscriber aka Observer

An important design pattern used by Qt in its signal-slot implementation is known as Publisher-Subscriber [Bus+96, pp. 339 sqq.] or Observer [Gam+94, pp. 293 sqq.]. Subscribers, as their name says, subscribe to changes of a certain property of a publisher who holds a list of the subscribers. When this certain property is changed, all subscribers are informed about the change. In Figure 2.9 the class diagram of this pattern is shown.

2.6.10 Concluding Remarks on Design Patterns

Using design patterns could be a great benefit for a company. Riehle describes this in his paper "Lessons Learned from Using Design Patterns in Industry Projects" ([Rie11]). He gained experience as a developer, in-house

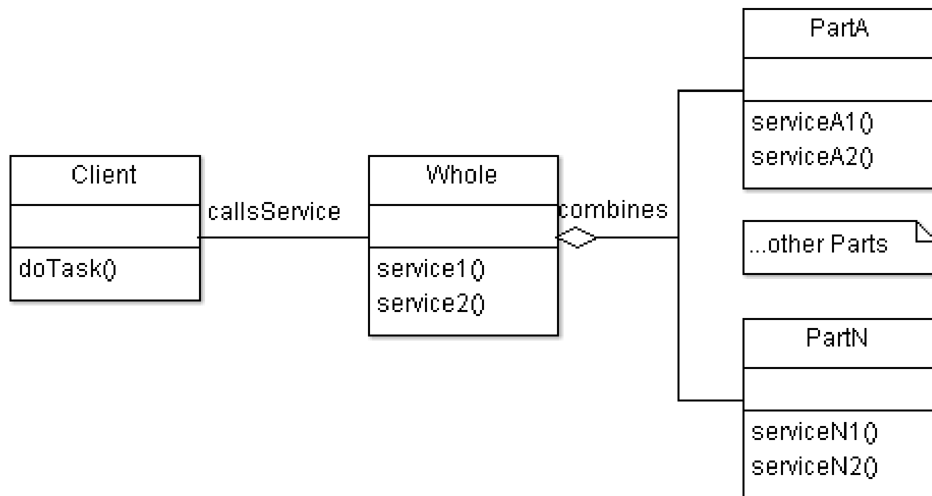


Figure 2.8: Class Diagram of Whole-Part Pattern. Image adapted from [Bus+96, p. 229].

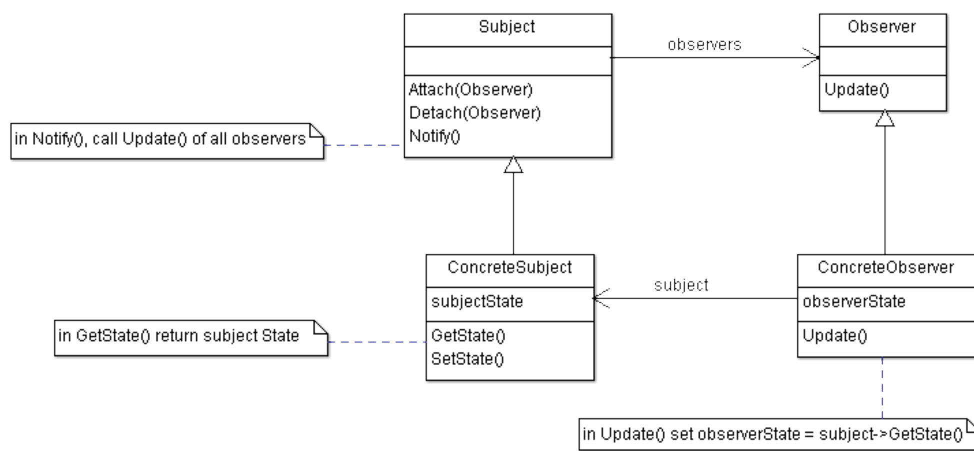


Figure 2.9: Class Diagram of Observer Pattern. Image adapted from [Gam+94, p. 294].

consultant and architect in various companies and explains the usage of design patterns in industrial projects. There are three types of design pattern usage:

The primary use is in the *communication* between the developers. When they talk about a problem, architecture or implementation they use a design pattern based vocabulary which makes communication much easier.

The second and obvious usage is in the *implementation* of the software. Design patterns "provide a more comprehensive "bigger" picture of the design and yet are specific enough to lead to code based on prior experiences." ([Rie11, p. 5]). One big advantage of the usage of design patterns in the code is, that other developers know them and therefore could understand the code faster.

The last field where design patterns are used is in the documentation. Here the advantage lies in the fact that the developers, who make the documentation, use the design pattern vocabulary and therefore others, who read it, know what is meant more quickly.

He also mentions that the projects benefited from design patterns and that firm-specific design language arose. Components of this language are firm-specific variations of design patterns, firm-specific patterns, the architecture of the firm's products and firm's programming practices.

Khomh and Guéhéneuce have a more critical view of the usage of design patterns. In their paper ([KG08]) they analyzed the impact of design patterns on different quality attributes of software development. The attributes are separated into three classes, namely those which are related to design, to implementation and to runtime. They estimated the impact of design patterns on those quality attributes by creating a questionnaire and analyzing the answers of twenty expert software engineers. They conclude that not all design patterns improve the quality of a software and therefore should be used with caution.

Ten years later, they published a retrospective paper ([KG18]) where they report and reflect studies on the impact of design patterns on software development. Khomh and Guéhéneuce give an overview of the different

types of design patterns and identified seven fields in software development where they occur.

1. Knowledge Sharing
2. Development Tools
3. Formalization
4. Forward Engineering
5. Reverse Engineering
6. Documentation
7. Quality

2.7 PImpl Idiom

In his book "Thinking in C++, Volume 1" [Eck00], Eckel explains a technique named Cheshire Cat. This technique is used to reduce compile time by moving data into a private structure. Similar to this, in the PImpl idiom [Ref18a] not only data is hidden in a private structure but a complete private class capsules private data and methods. This is used especially for libraries to make the Application Programming Interface (API) stable for changes of the memory footprint. Another advantage of the PImpl idiom is that the private members of a class are not visible in the class definition and in the interface, respectively.

In the implementation of Qt, this technique is widely used. Here it is called D-Pointer [Wik17] and primarily used for binary compatibility and also to hide implementation details, keep the header file clean and faster compilation.

2.8 Smart Pointers

Smart pointers are standard pointers where the memory management is done by the system. In Qt there exist different types of smart pointers which are described in their wiki [Wik18]. One of them is the QScoped-Pointer where the object is deleted when the pointer gets out of scope. A

detailed description of it could be found in the Qt Documentation Archives [Ltd17a].

In this project, the private pointer of the PImple Idiom is stored in a `QScopedPointer` to ensure that all data stored in the private class are deleted properly.

2.9 The Property System

Qt provides a property system [Ltd17b] which is based on its Meta-Object System and very useful for serialization. In the objects, the properties which have to be serialized are declared using the `Q_PROPERTY` macro. The corresponding getters and setters should be declared as well. Afterwards, all properties could be iterated using the QObject's member `metaObject()`. A detailed explanation of how this is used is given in Section 4.3.2.

2.10 Rule of Three/Five/Zero

In the C++ Reference [Ref18b], the rule of three states that if you need a user-defined destructor, it is almost certainly that also a user-defined copy constructor and copy assignment operator is needed.

Stroustrup and Sutter explain in their C++ Core Guidelines [SS18] that you should only implement default operations if you really need them. This is known as the "Rule of Zero". Additionally to this, the "Rule of Five" says that "If you define or =delete any default operation, define or =delete them all" ([SS18]).

3 Design

This chapter describes the design of TestPatternGenerator. The first section contains the specification given by Durst. It explains restrictions of the used technologies and defines additional requirements. Afterwards, use cases are identified and described. In the next section, the overall structure and concepts of the software are explained, followed by the definition of the printer configuration file and the pattern definition file. The chapter closes with the design of the graphical user interface.

3.1 Specification

There are several requirements given by Durst which have to be met. These requirements have been identified with the development manager and through talks with members of the quality and customer service department, which are the future users of the software.

The restrictions for the used software and libraries arise from the fact that they are currently used in these versions. By using them, there are no problems concerning licensing and other developers are used to it. As programming language, the C++ 11 standard was chosen with Qt in version 4.8 and LibTiff version 4.0.8. The coding was done using NetBeans IDE version 8.2 and cmake 3.11.2 as build tool. An overview of the used versions is shown in Table 3.1

As already explained in Section 1.3, the main issues which must be addressed by the software are usability, flexibility and expandability. To meet these requirements the design should allow to easily add new printers to the program without recompiling it. Also implementing new test patterns should not affect existing classes except where it is necessary.

Technology	Version
C++	11
Qt	4.8
LibTiff	4.0.9
cmake	3.11.2
NetBeans IDE	8.2

Table 3.1: Versions of Used Technologies

3.2 User Stories

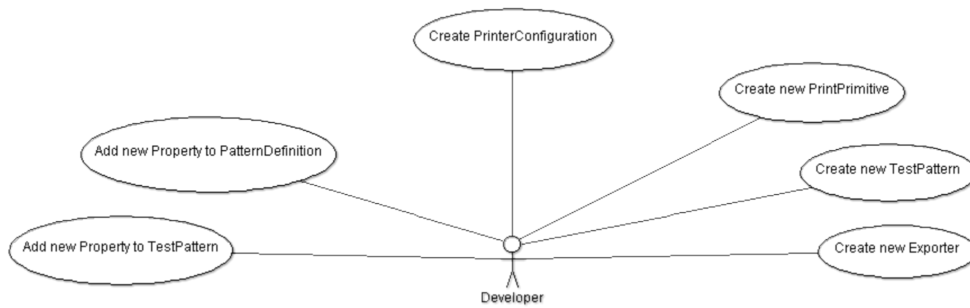
There are two different perspectives on the software which are illustrated in Figure 3.1. At first, there is the perspective from a user of the system (a) who needs to export test patterns to calibrate the printer. Another one is the developer's perspective (b) who is interested in an easy way to extend the software with new printers and test patterns and in adding new print primitives as easy as possible. A special case is where the TestPatternGenerator is used by the printer software to generate test patterns on the printer itself. These use case is covered by the last user story in Subsection 3.2.1 named "Run in Commandline".

3.2.1 User Stories for Users

The following user stories are from the perspective of the user of the TestPatternGenerator. These users are technicians from customer service, quality management and from the assembling department who need to adjust the printheads.

The different parts of the GUI which are referred in the following user stories are explained in Section 3.3.

(a)



(b)

Figure 3.1: User Stories

Import a Printer Configuration

Actuator User

Description After clicking on "Import Printer", a file dialog opens where the user can select the printer configuration file (.dpc). This file is then stored in the installation directory and an entry is added in the registry.

Consequences A new printer is available in the combo box for selecting printers.

Create a Pattern Definition

Actuator User

Description After clicking on "New Pattern Definition", a new pattern definition is created. This pattern definition is empty and all properties are set to the defaults. If another pattern definition was open, it is deleted and lost if not saved previously by the user.

Consequences A new and empty pattern definition was created.

Save a Pattern Definition

Actuator User

Description After clicking on "Save Pattern Definition", a file dialog opens where the user can specify the location and name at which the pattern definition should be stored.

Consequences A pattern definition file (.dpd) is created containing all test patterns and properties.

Load a Pattern Definition

Actuator User

Description After clicking on "Load Pattern Definition", a file dialog opens where the user can select the pattern definition file.

Consequences The specified pattern definition is loaded and shown on the GUI.

Add a Test Pattern

Actuator User
Description With a double-click on an entry in the patterns list on the left, the corresponding pattern is added to the pattern list from the pattern definition.
Consequences A test pattern is added to the pattern list.

Change Properties of the Pattern Definition

Actuator User
Description The properties of the pattern definition are shown in the table on the right. With a double-click on one entry, the user can change it. After clicking outside the table, the changed value is written to the property.
Consequences The property is changed.

Change Properties of a Test Pattern

Actuator User
Description To change the property of a pattern, the user needs to select the pattern in the pattern list. Afterwards, the properties of the selected pattern are shown under the properties of the pattern definition in the table on the right. Changing a property now works like changing a property of the pattern definition.
Consequences The property is changed.

Clear the Pattern List

Actuator User
Description With a click on the button ”Clear” all test patterns are removed from the pattern list from the pattern definition.
Consequences The pattern list is empty.

Change the Printer

Actuator User

Description Changing the printer is as simple as selecting it in the printer combo box.

Consequences A new printer is selected for which the pattern definition is exported to.

Export TIFF

Actuator User

Description After clicking on "Export TIFF", a file dialog appears where the user selects the place to export to. A progress dialog appears and the files are written to the specified location. If the pattern definition did not contain any test pattern, nothing happens.

Consequences A folder containing the tiff images and an expose (.exp) file is created.

Run in Commandline

Actuator User

Description Additional to the standard usage of the TestPatternGenerator it is also possible to run it in the command line. Here, no GUI is shown and the specified pattern definition is exported to the specified location.

Consequences A folder containing the tiff images and an expose file is created at the specified location.

3.2.2 User Stories for Developers

The developers extend the TestPatternGenerator with new printers, test patterns, exporters, etc. Their main focus lies on the software implementation of this tool and how to improve it. The user stories concerning them are explained in the following.

Create a Printer Configuration

Actuator Developer

Description To import a new printer, a printer configuration file (.dpc) has to be created. This is a simple XML file containing a name, the resolutions, mappings for color channels and the definition of the printheads. A detailed definition is given in Section 3.5.

Affected Classes none

Consequences A new printer configuration for importing into the TestPatternGenerator was created.

Create a new Print Primitive

Actuator Developer

Description For a new print primitive the base class PrintPrimitive has to be subclassed. The bounding rectangle must be set and the `getLine` method overwritten. This method returns an unsigned char*. Because the test patterns and print primitive are build together, it is ensured that all needed print primitives exist for the corresponding test pattern.

Affected Classes PrintPrimitive

Consequences The new printPrimitive could be used in the test patterns.

Create a new Exporter

Actuator Developer

Description Each exporter is derived from the abstract class `Exporter`. The only method which must be implemented is `exportTo` which contains the export logic. Each exporter has its own action in the export menu on the main window. To add a newly created exporter, a QAction has to be added and a method calling the `exportTo` method from the new exporter should be created.

Affected Classes Exporter, MainWindow

Consequences The newly created exporter is available under the export menu.

Create a new Test Pattern

Actuator Developer

Description A new test pattern must be derived from `PrintheadPattern` which is derived from Pattern. Since `Pattern` implements the interface `ISerializable`, the methods `serialize` and `deserialize` should be implemented and also the method type must be filled with the correct String. The most important method `init` must be implemented properly to add the required print primitives. After the new test pattern is implemented, it must be added to the `PatternFactory` in the method patterns, where a QStringList of all available test patterns is returned and in the method pattern where an instance of the desired test pattern is returned.

Affected Classes Pattern, PrintheadPattern, PatternFactory

Consequences The new test pattern is shown in the pattern list on the left and is available to use.

Add a new Property to the Pattern Definition

Actuator Developer

Description For adding a new property to the pattern definition, the developer creates it in the pattern definition header file using the `Q_PROPERTY` macro. In the private class, the corresponding member-variable as well as getter and setter methods must be created. The loading and storing using the serializer pattern is done automatically.

Affected Classes PatternDefinition

Consequences The added property is available in the properties for pattern definition on the right side.

Add a new Property to a Test Pattern

Actuator Developer

Description Similar to adding a property to the pattern definition, it could be done with a pattern. Create the definition with the `Q_PROPERTY` macro, add member-variable and add getter and setter methods respectively. As for the pattern definition, serialize and deserialize should

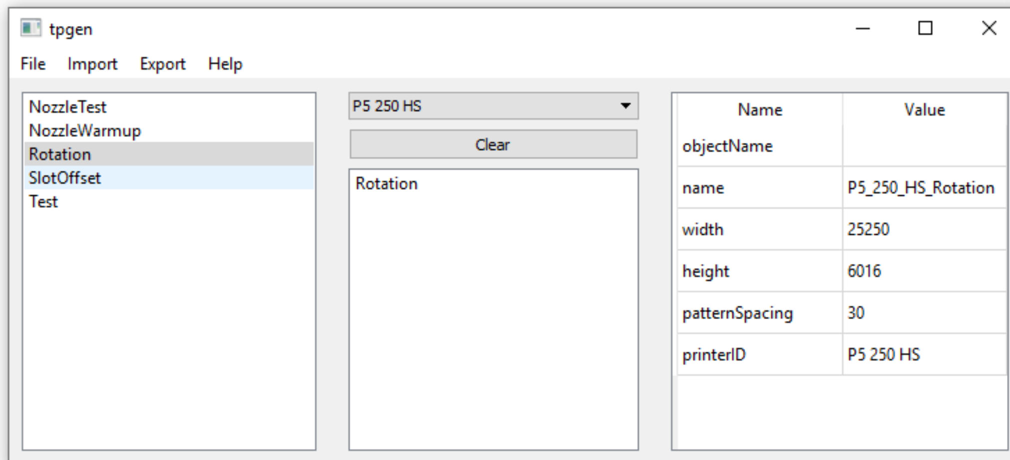


Figure 3.2: GUI of TestPatternGenerator

not be touched.

Affected Classes corresponding pattern class

Consequences The added property is available in the properties for the pattern on the right side.

3.3 GUI

Like the overall design of the system, also the GUI design follows the KISS principle. The main focus is on a clear structure to ensure an intuitive and easy use. The GUI, which is shown in Figure 3.2, consists of a menu bar and three sections.

The menu bar is separated into four menus. The first one named "File" provides the functionality for creating a new pattern definition, open an existing one and saving it. With import, a new printer configuration can be loaded into the TestPatternGenerator. Under the menu "Export" the different exporters can be found. In the first version, there is only the `TiffExporter`. Finally, with "Help" a dialog with additional information can be shown like in many other applications.

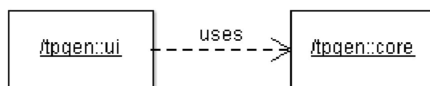


Figure 3.3: Components of TestPatternGenerator

The left section contains a list of the available test patterns. With a simple double-click, they are added to the pattern definition. The section in the center represents a pattern definition. On top, the target printer can be selected with the combo box. Underneath a button to clear the pattern list of the pattern definition can be found. The rest of the section contains the list of the test patterns which the current pattern definition contains. The last section is used to display and change the properties of the pattern definition and the selected test pattern.

3.4 System Architecture

The system is basically separated into two components as shown in Figure 3.3. The core lasses contain the classes for printers, print primitives, test patterns and the pattern definition. The UI classes on the other hand, contain the main window, the three sections for displaying a list of available test patterns, the pattern definition with the selected test patterns and for displaying their properties.

Figure 3.4 shows a diagram of the UI classes. `PatternDefinitionSection` and `PropertiesSection` need separate models for their list and table respectively. Detailed descriptions of the classes could be found in Section 4.2.

An overview of the core classes could be found in Figure 3.5. The classes `PrintPrimitive`, `Pattern` and `Exporter` are abstract base classes. The complete class diagram and detailed information about each class could be found in Section 4.3.

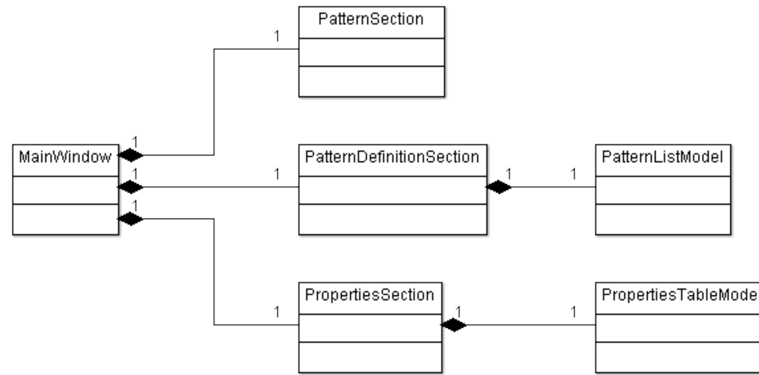


Figure 3.4: Overview of UI Classes

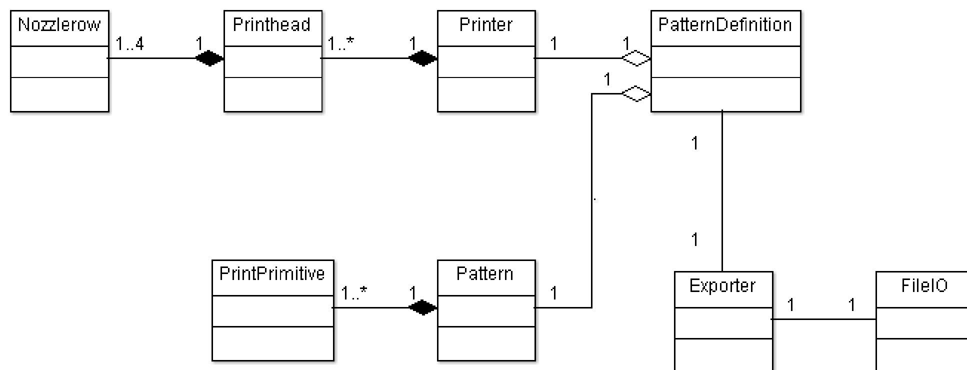


Figure 3.5: Overview of Core Classes

3.5 Printer Configuration File

Information about a printer, especially the geometry of their printheads is provided to the system with so-called printer configuration files. These files have to be written by a technician and are separated into three parts. The first contains general information about the printer itself such as name, resolutions and color channels. The second part defines enumerations which are used to simplify the definition of the printheads. Finally, the last part contains the information about the printheads itself. A part of a printer configuration (in this case for a P5 250 HS) is shown in Figure 3.6.

The name is needed in the printer selection on the GUI. The resolutions are written with the `TiffExporter` into the exported image files. The color channels contain a mapping to the different tiff channels used by the printer. This is a simplification to have more meaningful values for the colors of the different nozzlerows. Most printers have more equal printheads, one under another, with the same layout except the y position. In this case, the enumerations are used to simplify it. An enumeration has an id, a start- and end-number and a spacing between the previous printhead. Only the first printhead is specified, the rest of them are generated automatically. A printhead definition has an id, which could also contain an enumeration. The geometry defines the position and size of the printhead in pixels and pixelSpacing defines the spacing between the nozzles of one nozzlerow. Some test patterns need a reference for each printhead. these references are defined in the alignment-references section with a name and the id of the reference printhead. Finally, each printhead contains a list of nozzlerows with a specific color and nozzle count. They also have a geometry similar to the geometry of the printhead. Note, that the X-coordinates (of the nozzlerows and printheads) are not the real coordinates of the physical device. In this simplification, a nozzlerow has only a width of one pixel and there is no space between them. The last element is printheads, where only an id is specified (containing an enumeration), the reference printhead and a position where it is anchored.

```xml
<printer>
    <name>P5 250 HS</name>
    <resolution x="800" y="800" mediaAdvance=""/>
    <colorChannels>
        <channel color="cyan" tiffchannel="0"/>
        <channel color="magenta" tiffchannel="1"/>
        <channel color="yellow" tiffchannel="2"/>
        <channel color="black" tiffchannel="3"/>
        <channel color="lightcyan" tiffchannel="6"/>
        <channel color="lightmagenta" tiffchannel="7"/>
        <channel color="white" tiffchannel="4"/>
        <channel color="label" tiffchannel="3"/>
    </colorChannels>

    <enumerations>
        <counter id="#X1_" start="1" end="8" spacing="0"/>
        <counter id="#X2_" start="1" end="8" spacing="0"/>
        <counter id="#X3_" start="1" end="8" spacing="0"/>
        <counter id="#X4_" start="1" end="8" spacing="0"/>
        <counter id="#X5_" start="1" end="8" spacing="0"/>
        <counter id="#X6_" start="1" end="8" spacing="0"/>
        <counter id="#X7_" start="1" end="8" spacing="0"/>
        <counter id="#X8_" start="1" end="8" spacing="0"/>
        <counter id="#X9_" start="1" end="8" spacing="0"/>
        <counter id="#X10_" start="1" end="8" spacing="0"/>
        <counter id="#X11_" start="1" end="8" spacing="0"/>
        <counter id="#X12_" start="1" end="8" spacing="0"/>
        <counter id="#X13_" start="1" end="8" spacing="0"/>
        <counter id="#X14_" start="1" end="8" spacing="0"/>
    </enumerations>

    <printhead id="A7.1/#X1_">
        <geometry x="0" y="0" width="4" height="752" pixelSpacing="8"/>
        <alignment-references>
            <ref name="slotOffset" reference="A1.1/1"/>
        </alignment-references>
        <nozzlerow color="white" nozzleCount="94">
            <geometry x="0" y="0" width="1" height="94"/>
        </nozzlerow>
        <nozzlerow color="white" nozzleCount="94">
            <geometry x="1" y="4" width="1" height="94"/>
        </nozzlerow>
        <nozzlerow color="white" nozzleCount="94">
            <geometry x="2" y="2" width="1" height="94"/>
        </nozzlerow>
        <nozzlerow color="white" nozzleCount="94">
            <geometry x="3" y="6" width="1" height="94"/>
        </nozzlerow>
    </printhead>
    <printheads id="A7.1/#X1_" reference="A7.1/1" position="bottom"/>

```

Figure 3.6: Printer Configuration for P5 250 HS

```xml
<patterndefinition printerID="P5 250 HS" name="P5_250_HS_SlotOffset" patternSpacing="30">
  <pattern lineWidth="3" type="SlotOffset" border="0" spacing="500" nozzleWarmupWidth="250">
    <printheads/>
  </pattern>
</patterndefinition>
```

Figure 3.7: Pattern Definition of SlotOffset for P5 250 HS

3.6 Pattern Definition File

The pattern definition file is used for saving and loading generated pattern definitions including their test patterns and properties. These files are only generated by the TestPatternGenerator and are very short compared to the printer configuration files. An example of such a pattern definition file is shown in Figure 3.7.

The pattern definition contains a name which is used for the file- and folder names in the exporter, the id of the selected printer and since it could contain more than one test pattern, a spacing between the different test patterns is defined. The properties of the test pattern depend on the type of the pattern. In this case, a SlotOffset, the line width of the pattern, a border before and after the test pattern, the spacing between the different parts of the test pattern and the width of the nozzle warmup section is defined. The printheads section is not needed in the first version of the TestPatternGenerator, like explained in Section 4.3.6, but was kept to be prepared for future improvements.

4 Implementation

In this chapter, the implementation of the TestPatternGenerator is discussed. It starts with the development workflow, followed by descriptions of the UI classes and the core classes with the test patterns and the tiff-exporter. The chapter closes with a case study where different algorithms of the exporter are compared in terms of runtime.

4.1 Development Workflow

The development process for this software followed an iterative workflow where the KISS principle has always been kept in mind. First, the basic classes `FileIO`, `PatternDefinition`, the classes for the print primitives, the first patterns and the first exporter have been created. These parts of the software are explained in the following sections. To work with them properly, also a basic GUI was build which is explained in Section 3.3.

While for the first pattern, the `NozzleWarmupPattern`, just simple lines were needed, the next patterns need more than that. The RotationPattern consists of a very big number of lines. For the P5 250 HS, there are already 1000 lines for each printhead and there are 112 printheads. Needless to say, that this number of objects could hardly be managed with this software. To solve this problem, we create a new print primitive which combines a number of lines – the `LineBlock` was created. Like this, several `PrintPrimitives` had been created and also the software itself changed over time.

Another idea was to create a factory for the print primitives to reduce the number of parameters for the different print primitives. The factory should store the properties for media advance and direction and set them in the created `PrintPrimitive`. The reasons that it was not implemented are that

the signatures of the constructors for the different `PrintPrimitive` vary so much and it would indeed reduce the number of parameters by two, but it would also reduce the clarity of the patterns. If a developer has to change a pattern, it is easier to see these properties at first glance.

4.2 UI Classes

In the following subsections, all classes of the GUI are explained. An overview of how they are linked was already shown in Figure 3.4. The main class is the `MainWindow` which contains the menu bar and the three sections for the test patterns, the pattern definition and the properties.

4.2.1 MainWindow

This class is derived from QMainWindow and provides the functionality for creating, saving and loading pattern definitions, importing printer configurations and exporting into tiff files. These functions are provided through QMenus in a QMenuBar. The three sections the `MainWindow` contains are placed in a QSplitter and the communication between them is done via signals and slots, Qt's implementation of the publisher-subscriber pattern. The class diagram of this class is shown in Figure 4.1.

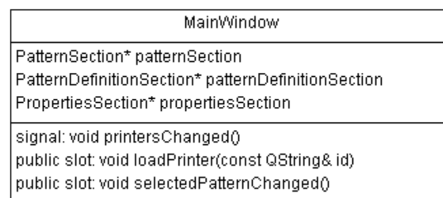


Figure 4.1: Class diagram of MainWindow

4.2.2 PatternSection

The `PatternSection` is a QWidget containing a QListWidget where all test patterns are represented. With a double click, a pattern is selected and the corresponding signal is emitted. This is a quite small class and its diagram is shown in Figure 4.2.

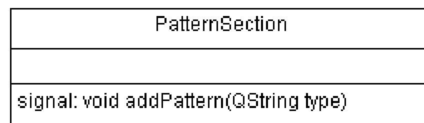


Figure 4.2: Class diagram of PatternSection

4.2.3 PatternDefinitionSection

The most interesting GUI class is `PatternDefinitionSection` which is shown in Figure 4.3. It holds members of the selected printer and the pattern definition. To display the containing test patterns, it needs a list model named `PatternListModel` which provides the data shown in a QListView. When the selected printer changes, the new printer is set to the test patterns of the pattern definition and also to the `PatternFactory`. This has to be done to ensure, that the correct test pattern is created.

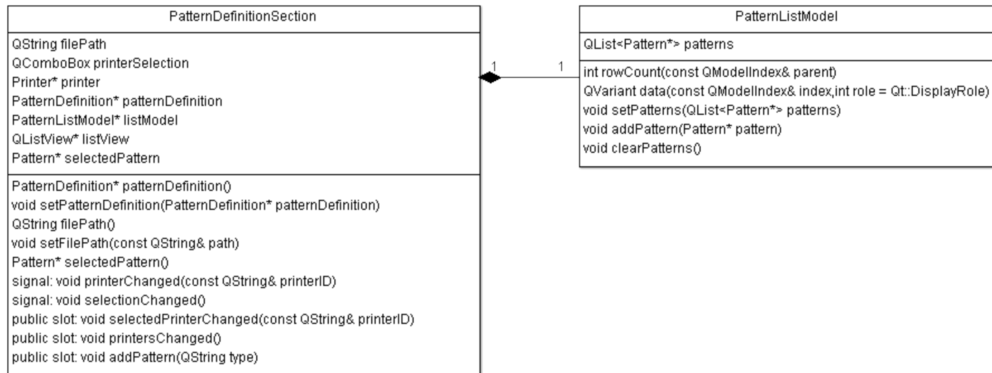


Figure 4.3: Class diagram of PatternDefinitionSection

4.2.4 PropertiesSection

The last component of the GUI is the `PropertiesSection`. As shown in Figure 4.4, it contains a QObject and uses the `PropertyTableModel` to display the object's properties in a QTableView. The PropertyTableModel provides all properties of the specified object using Qt's property system.

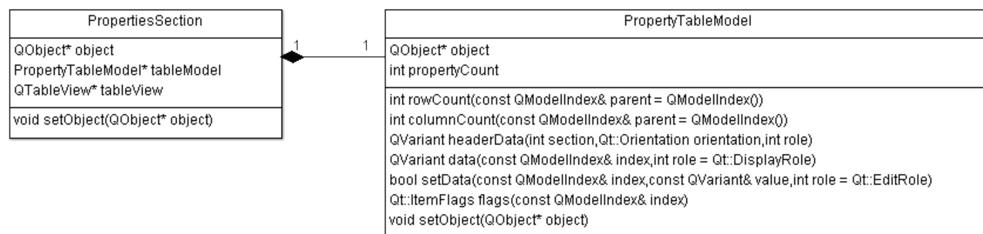


Figure 4.4: Class diagram of PropertiesSection

4.3 Core Classes

The following subsections explain various classes. Figure 4.5 gives an overview of the core classes. There are three separate hierarchies with the base classes PrintPrimitive, Pattern and Exporter. PrintPrimitive has five direct subclasses representing the basic units of which a pattern consist. Derived from Pattern, the PrintheadPattern is the base for four test patterns. PrintheadPattern is currently not used but was inserted to meet the future goal to only export a test pattern for only a few printheads. The third one is Exporter which is the base for TiffExporter. The Exporter uses the class FileIO to write the exported files. Another part is the Printer class which represents a physical printer. It consists of a number of printheads which again consist of up to four nozzlerows. The central class of the whole structure is the PatternDefinition. It contains a list of patterns, holds an instance of the selected printer and uses the Exporter.

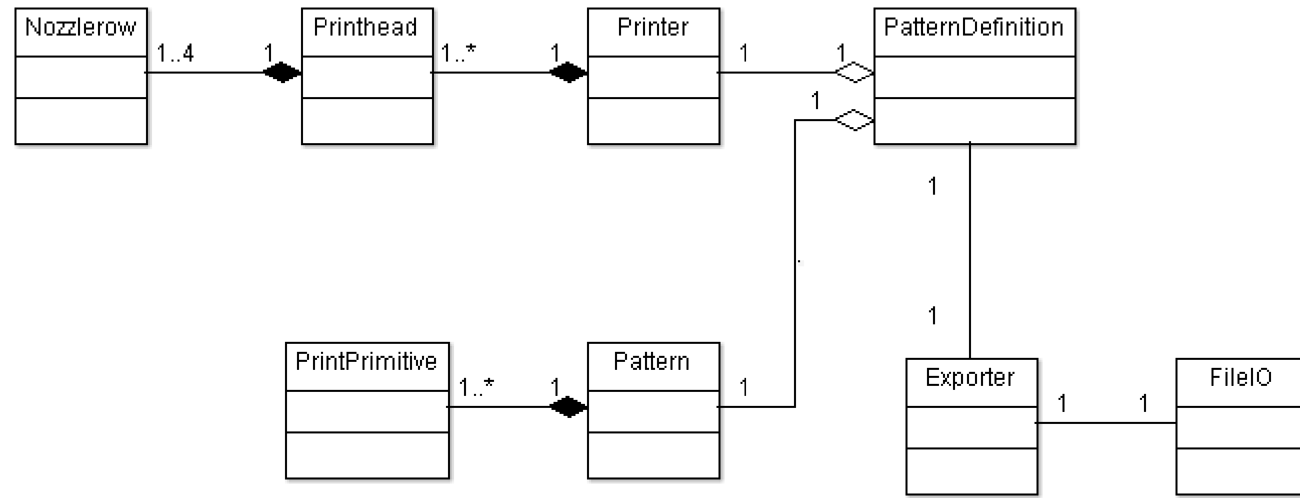


Figure 4.5: Core Classes

4.3.1 FileIO

The class where the whole input and output to the file system is implemented is `FileIO`. As shown in the class diagram in Figure 4.6, it has four methods for saving and loading a `PrinterConfiguration` and a `PatternDefinition`. Since creating a `PrinterConfiguration` is not supported yet, this method is still empty and just returns false.

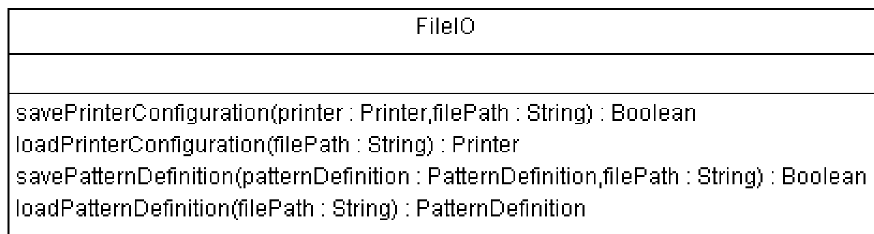


Figure 4.6: Class diagram of FileIO

Considerations have been made to implement this class as a singleton, but since no states or anything other needs to be stored it has not been necessary.

4.3.2 ISerializable

This simple interface defines methods to serialize and deserialize an object. The only parameter for both classes is a QDomElement. While in the serialize method, the QDomElement is filled with data from the object, the deserialize method sets the properties of the object according to the data from the retrieved QDomElement. The class diagram of this interface is shown in Figure 4.7.

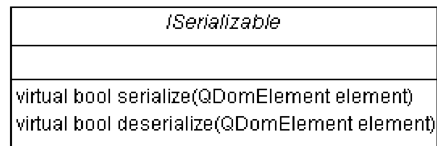


Figure 4.7: Class diagram of ISerializable

4.3.3 Pattern Definition

A `PatternDefinition` is the basic object the user works with. It holds properties like a name, a spacing between the single patterns, the id of the selected printer and also a list of patterns. The class diagram is shown in Figure 4.8.

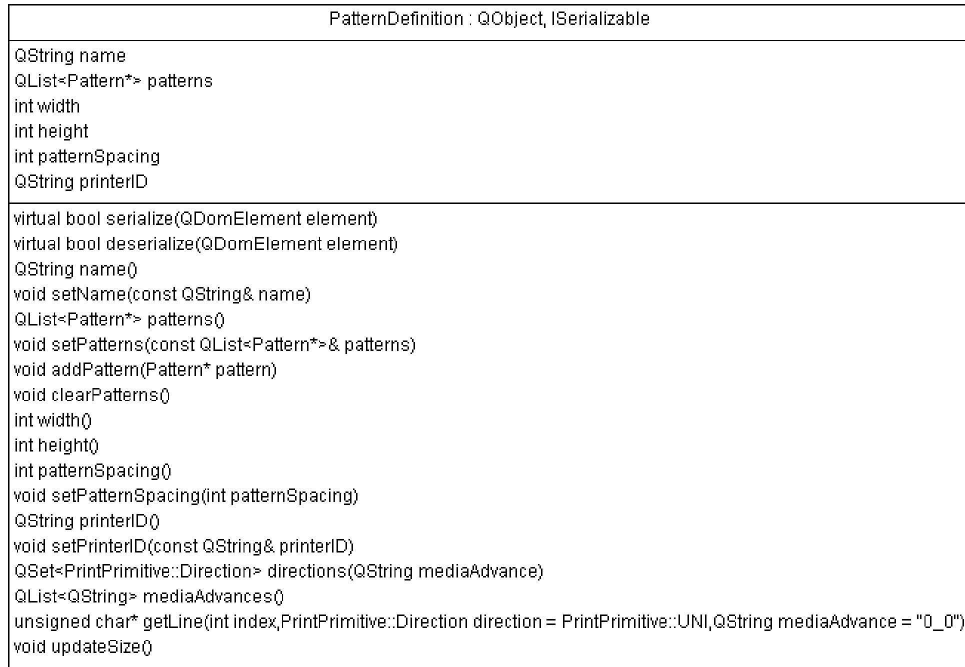


Figure 4.8: Class diagram of PatternDefinition

The `PatternDefinition` also implements the serialization interface. According to the definition in Section 4.3.2 the object data is serialized into the QDomElement or deserialized from it.

The serialization of the standard properties works straight forward, while the pattern list is iterated and for every single pattern, the corresponding serialization method is called. The pattern class itself implements the `ISerializable` interface. For the deserialization, the properties of the `PatternDefinition` are set and for each pattern-entry in the XML file, the `PatternFactory` creates a new object from which the deserialization method is called and this deserialized object is added to the `PatternDefinition`.

Exporting a pattern using the `TIFFExporter`, for each mediaAdvance and direction a different file is needed as described in Section 4.3.7. To get the

information, which directions and media advances are needed, the two methods directions and mediaAdvances are needed. The method directions returns a QSet containing the directions which are needed for the containing patterns. For example, the RotationPattern only needs to be printed in one direction while the SlotOffsetPattern contains parts for both directions, uni and bidi. It needs the mediaAdvance as a parameter since not for each mediaAdvance the same directions are needed. The method mediaAdvance does not need any parameters and returns a QList containing the strings of the needed mediaAdvances.

4.3.4 Printer

One of the most important parts providing the flexibility of this software is the `Printer` class. The user has the possibility to select between different installed printers. For this purpose, the classes shown in Figure 4.9 are used.

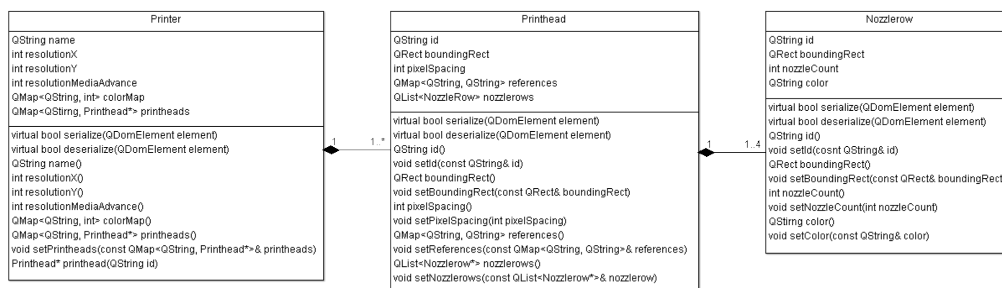


Figure 4.9: Class diagram of Printer and associated classes

Each printer has a name, resolutions in X and Y-direction and for the media advance, and a colormap where the mapping between the colors of the printer and the channel in the tiff is done. Also, each printer has a number of printheads with an id, a geometry stored as a boundingRect, pixelSpacing where the amount of pixels between two vertical nozzles is stored. It also could have a map of references, where it is defined for which

pattern which other printhead is used as a reference. And finally, it can consist of a number of so-called nozzlerows. These are the smallest unit of a printer in this software model and represent a collection of nozzles with the same color. They also have a boundingRect, the number of nozzles stored in nozzleCount and a color.

These information is loaded from the printer configuration file which was explained in Section 3.5 using the class `FileIO`.

4.3.5 Print Primitives

The basic units of the patterns are the so-called print primitives. The different print primitives implemented for the test pattern generator are explained in the following. The hierarchy of the print primitives classes can be seen in the class diagram in Figure 4.10.

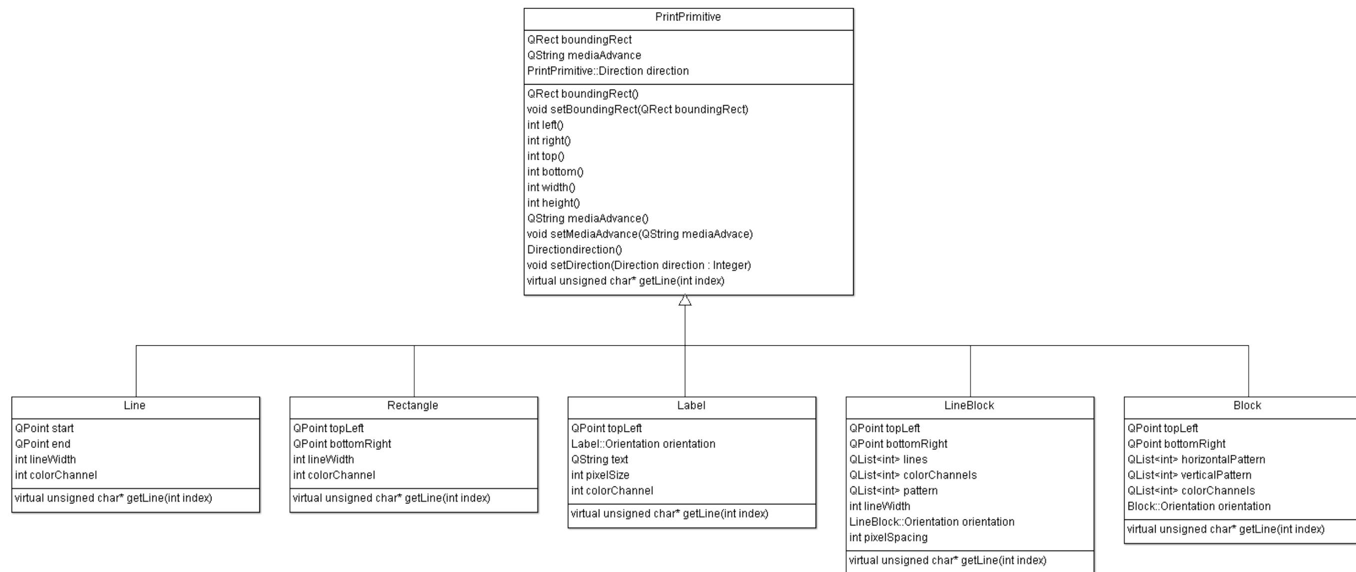


Figure 4.10: Class diagram of print primitives

PrintPrimitive

The root class for all print primitives is called `PrintPrimitive`. It contains the general properties for the print primitives, the media advance and an enum for the direction which could be unidirectional or bidirectional.

Additionally to the standard getter and setter methods, there are methods for getting values of left, right, top, bottom, width and height. These are just to simplify working with the print primitives by providing direct access to these properties.

One important method of all primitives is `getLine`. As mentioned before, the tiff files are written line by line and for this purpose, the virtual method `getLine` is used to retrieve the data at the given index.

Each derived print primitive must calculate their bounding rectangle themselves and set it properly.

Line

The first implemented print primitive is a simple line. It is defined by two QPoint members defining the start and the end of the line and two integers for the line width and the color channel on which it is going to be printed. Since all lines are parallel to the axis, there are only horizontal and vertical lines supported. If future demands occur, this could be easily adapted by simply modifying the `getLine` method.

Rectangle

The next print primitive, the rectangle, is mostly used to frame the different parts of the patterns to distinguish them from each other. The signature of this class is similar to that of the `Line`. The two QPoint members represent the top left and bottom right point of the rectangle. The two integers define the line width and the color channel.

Label

Since the adjustments are performed on all printheads, the different patterns consist of a lot of similar looking blocks of print primitives. E.g. the slot offset is measured for each printhead in both directions, unidirectional and bidirectional. Due to this, it is essential that the different parts of the pattern can be distinguished quickly. For this purpose, the `Label` is used.

A QPoint defines its top left position, the enumeration Orientation states whether it should be printed horizontally or vertically. The text is set as QString and two integers define the pixel size of the text and the color channel.

LineBlock

Most patterns consist of lots of single lines. The implementation of the first prototype has shown that creating a test pattern with a composition of single lines is very memory intensive since each line is an individual object. Due to this fact, the need for a more complex structure arose and the `LineBlock` was created.

Similar to Rectangle, two QPoints define the top left and bottom right position of the `LineBlock`. The property lines is a list of integers containing the indices of the lines starting by zero. `QList<int> colorChannels` define the color channels for the different lines. With pattern, which is also a list of integers, a pattern for the lines could be defined, e.g., to create dotted lines. The line width can also be defined as well as the orientation and the pixel spacing which defines when the pattern is repeated.

After implementing the next print primitive, the `Block`, this becomes obsolete and is removed from the test patterns to simplify them.

Block

The `Block` is quite similar to the `LineBlock`, but the developer has the possibility to define a horizontal and a vertical pattern. The orientation is

only used to define whether the list color channels are applied horizontally or vertically. The top left and bottom right coordinates are equal to those of the `LineBlock`.

Remarks on Print Primitive

Considerations have been made to create a primitives factory where general properties like direction and mediaAdvance could be defined only once they change and the individual properties of the primitives have to be set after creation. This approach would make the object creation simpler, but it would reduce the clarity in the pattern classes.

4.3.6 Test Patterns

The central and most important part of the test pattern generator are of course the patterns themselves. For all different types of adjustments, which have to be done for a printer, there are individual test patterns. In the first version, only a limited number have been implemented which are described in the following.

One important requirement was that it should be easy to add new patterns, and to ensure this, a pattern class hierarchy, which can be seen in Figure 4.11 and a factory for creating patterns was created.

PatternFactory

The `PatternFactory` is implemented as a singleton and is responsible for the pattern creation. It implements the prototype pattern and holds a map of test patterns. When a test pattern is retrieved, the factory returns a copy of the stored object if it exists, otherwise a null-pointer is returned.

After creating a new pattern, the `registerPatterns` method must be updated. The new test pattern just needs to be inserted into the map of test-patterns using a unique name.

Figure 4.11: Class diagram of patterns

Pattern

This is the base class for all patterns. It has properties for the user to specify the border left and right to the pattern and also for spacing between different parts of a pattern. It contains data structures for storing the different print primitives. These are two separate QMaps (for each direction) where the mediaAdvance is the key, and a QList of `PrintPrimitive` is the value.

As mentioned earlier in this paper, getting the data of a line is done using the plane-sweep algorithm, where only the print primitives which are currently (at the specified index) visible is considered.

PrintheadPattern

The class `PrintheadPattern` is needed to create a pattern only for a few individual printheads. For this purpose this class is derived from `Pattern` and contains a list of printheads. For the first version, the patterns are exported for all printheads but it is planned to expand this functionality in a future version.

NozzleWarmupPattern

The first implemented pattern is used for a simple nozzle warmup. Before printing patterns for the adjustments, it is necessary to print with all nozzles to get them to the operating temperature. A part of the created pattern is shown in Figure 4.12.

As can be seen, it consists of simple vertical lines for each color channel. It uses all eight channels from the tiff, but since they cannot be interpreted, four of them are shown white. The printers, however, interpret the different channels according to their setup, but it is still possible that some are not used.

How the printed pattern looks like is shown in Figure 4.13. The photo was taken with a magnification factor of 20.

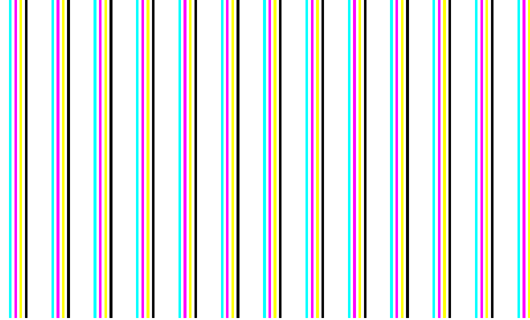


Figure 4.12: NozzleWarmup pattern

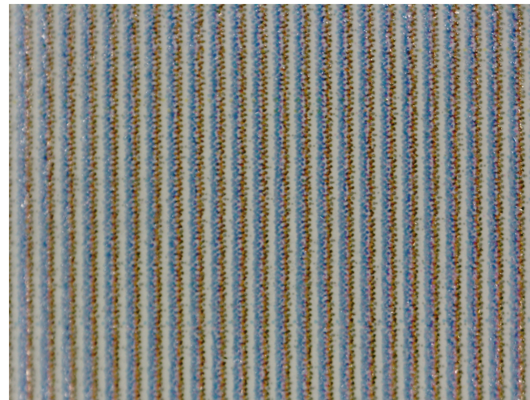


Figure 4.13: NozzleWarmup pattern printed with P5 250 HS

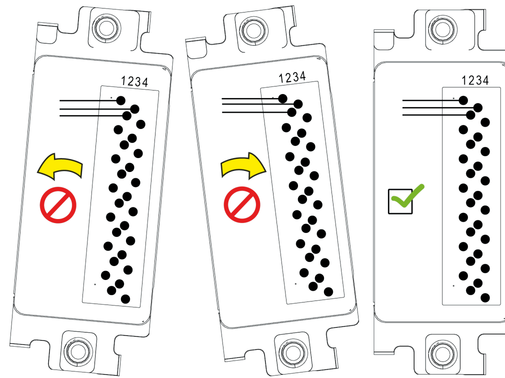


Figure 4.14: Adjusting Printhead Rotation. Image taken from [Dur18a].

RotationPattern

The next pattern is used to adjust the rotation of the single printheads. As described in Chapter 1, each printhead consists of different slots containing the nozzles. To check the rotation, lines printed by three nozzles are required. Two lines are the references and one measuring line, which have to be between the reference lines. If one looks at the individual nozzles in y direction (from top), they are arranged one after another. To see the rotation better, it is useful to choose the different nozzles in a way that their distance in X-direction is maximal.

Figure 4.14 gives an overview how the rotation of a printhead is done. If the measuring line is closer to the lower reference line, like in the left sketch, the printhead must be rotated counterclockwise. If it is closer to the upper reference line, the printhead must be rotated clockwise and if its right in between the measuring lines, the rotation is perfect.

A part of the generated pattern is shown in Figure 4.15. The label on the left side indicates which printhead is concerned, in this case, the first printhead of the second row on the left side. It is followed by four sections of lines. In the first one, lines are printed with all nozzles. The second section contains only the reference lines, followed by a section containing only the measuring lines. Finally, in the fourth section, the measuring and the referencing lines

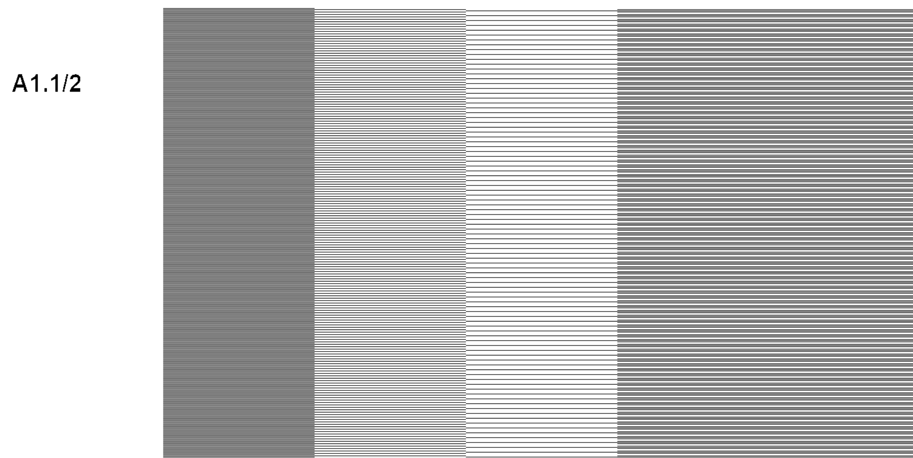


Figure 4.15: Rotation pattern

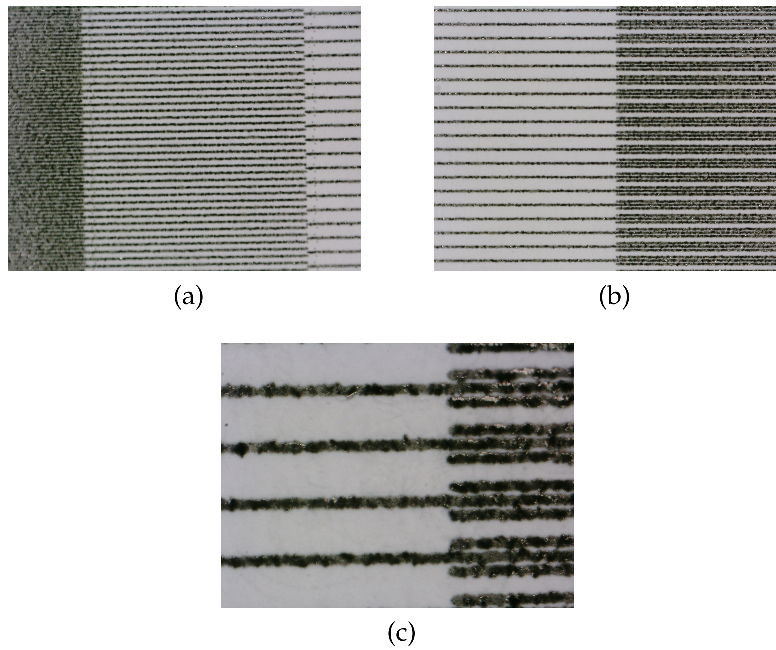
are printed. Because this is the most important one, it has twice the size of the other sections.

Figure 4.16 shows the printed pattern. In (a) and (b) parts of the four sections are shown with a magnification factor of 20, while in (c) a more detailed view (magnification factor of 200) of the transition from section three to four is shown.

The following Figures 4.17 show the pattern printed with a Rho 13xx. This printer has two different colors per printhead. Similar to Figure 4.16 the four parts are shown in (a) – (c) with a magnification factor of 20 and (d) shows the transition from section three to four with a magnifaction factor of 200.

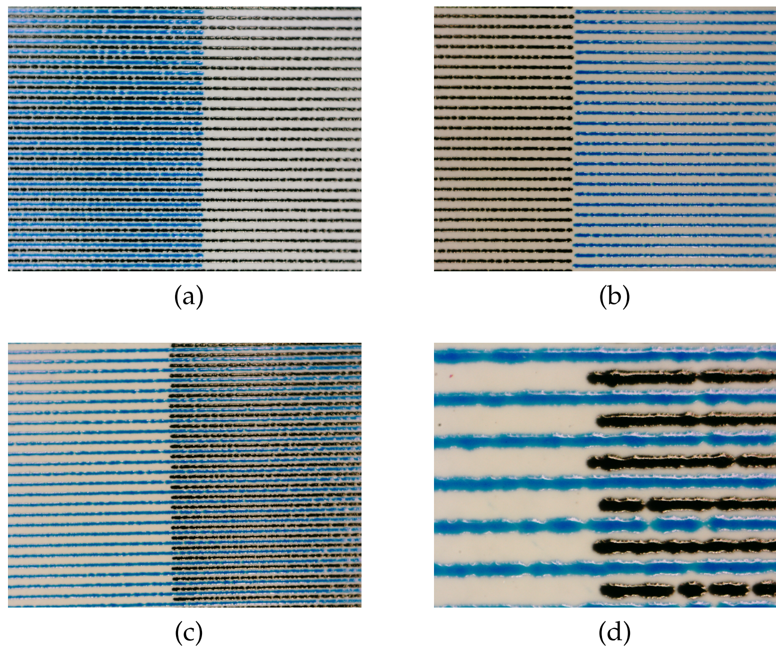
SlotOffsetPattern

In contrast to the RotationPattern, the SlotOffsetPattern is not used for a mechanical adjustment of the printhead. It is used to set the time adjustment when the different printheads fire their nozzles. For this purpose, a reference printhead is needed, e.g., for the P5 250 HS it is the first black printhead on the left.



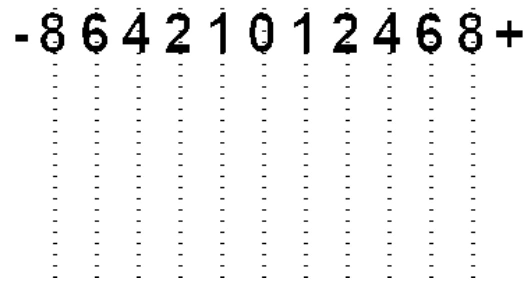
(a) (b) (c)

Figure 4.16: RotationPattern printed with P5 250 HS

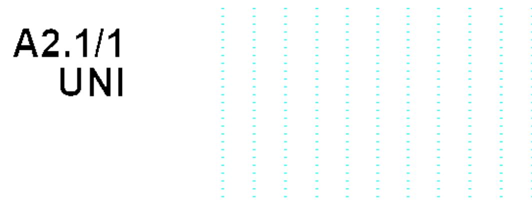


(a) (b) (c) (d)

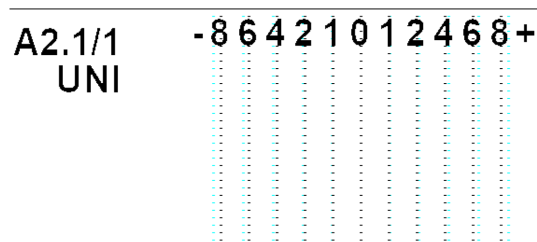
Figure 4.17: RotationPattern printed with Rho 13xx



(a) Reference



(b) Measurement



(c) Measurement

Figure 4.18: SlotOffset

Another difference to the RotationPattern is that this pattern cannot be produced with only one image file. For each row of printheads and for each direction, a separate file is needed to print the reference and after printing the references for all rows seen in (a) of Figure 4.18, another two files are needed to print the measurement for both directions (b). For the P5 250 HS, this sums up to eighteen files (two times 8 for the references and two measurements). Figure (c) shows the two parts overlapped.

How the printed pattern looks with a magnification factor of 20 can be seen in Figure 4.19. As can be seen, the adjustment for this slot would be +1. In Figure 4.20 the printed offset can be seen with a magnification factor of

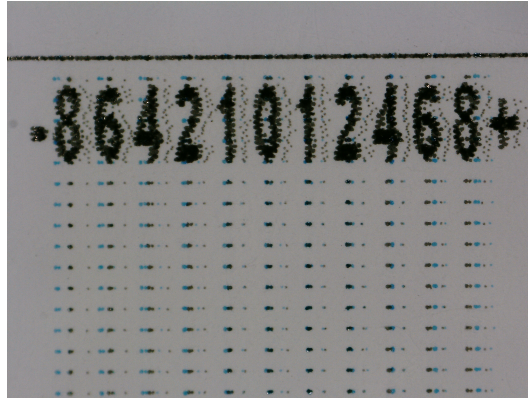


Figure 4.19: SlotOffset printed

200.

4.3.7 Exporter

After creating a pattern definition, it has to be exported for printing it on the different printers. For this purpose, there exists the abstract class `Exporter` which defines the virtual function `exportTo`. It takes a QString containing the export path and a pointer to the `PatternDefinition` which should be exported. In the first version, only the `TiffExporter` was implemented which is explained in the following.

In Figure 4.21 the class diagram of the exporters can be seen.

TIFFExporter

The `TiffExporter` was created to export a pattern definition into tiff images and an additional expose (.exp) file. There could be more image files, because for each different media advance and for each direction a different file is needed. For example, the reference for the slot offset must be printed separately for all printhead rows and also for each direction. The measurement is then printed on top of the references. The .exp file tells the printer

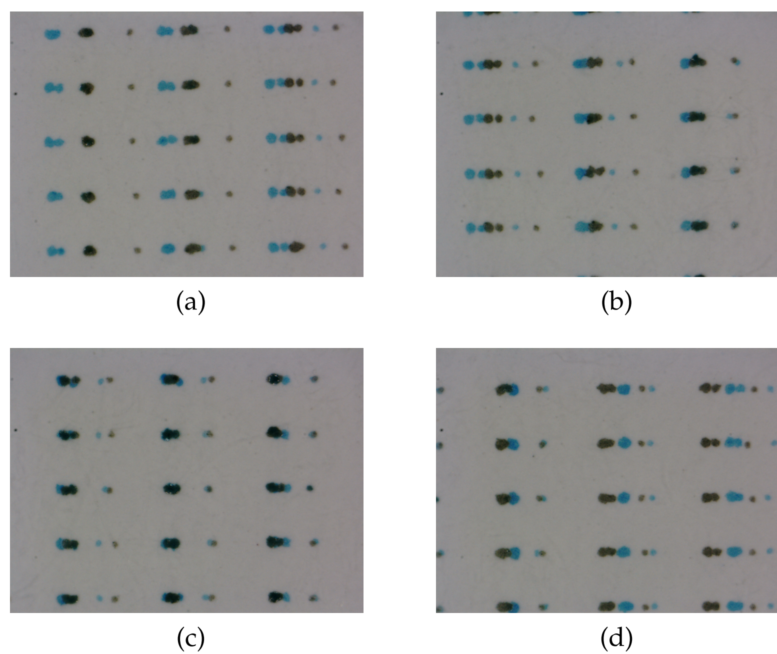


Figure 4.20: SlotOffsetPattern printed zoomed

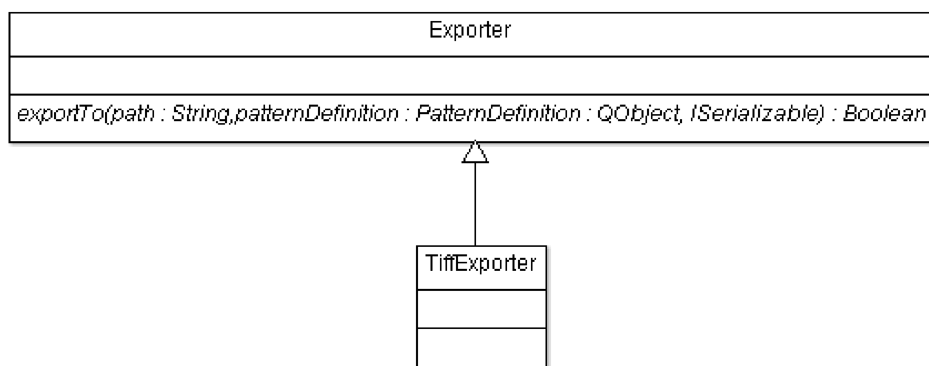


Figure 4.21: Exporter

which image to print and how much media advance is needed before the next image.

Figure 4.22 shows a simplified sequence diagram of how the exporter works. After creating the `Exporter` and calling the export method, it gets the information about the printer, i.e., resolution in X- and Y-direction and a list of the printheads. Additionally, the width, height, a list of the mediaAdvances and a list of the needed directions are fetched from the pattern definition. Now, for each mediaAdvance and direction, a separate file is created and in a loop, which iterates over the total height of the test pattern, the information for the specified index is written into the file. The pattern definition iterates over all containing patterns where also the `getLine` method is called and the results are concatenated to a line for the exporter which is written to the file. Finally, the different patterns also retrieve the necessary data from their print primitive.

Figure 4.23 shows the folder structure of the exported patterns. The exported tiff files have the following naming convention:

<name>_<mediaAdvance>_<direction>.tif

name Name of the exported pattern definition specified in the GUI or as a parameter in the commandline.

mediaAdvance The first number is the media advance in pixels, the second is a counter. The counter is needed because sometimes two images with the same media advance must be printed in a separate pass.

direction This indicates in which direction the file should be printed.

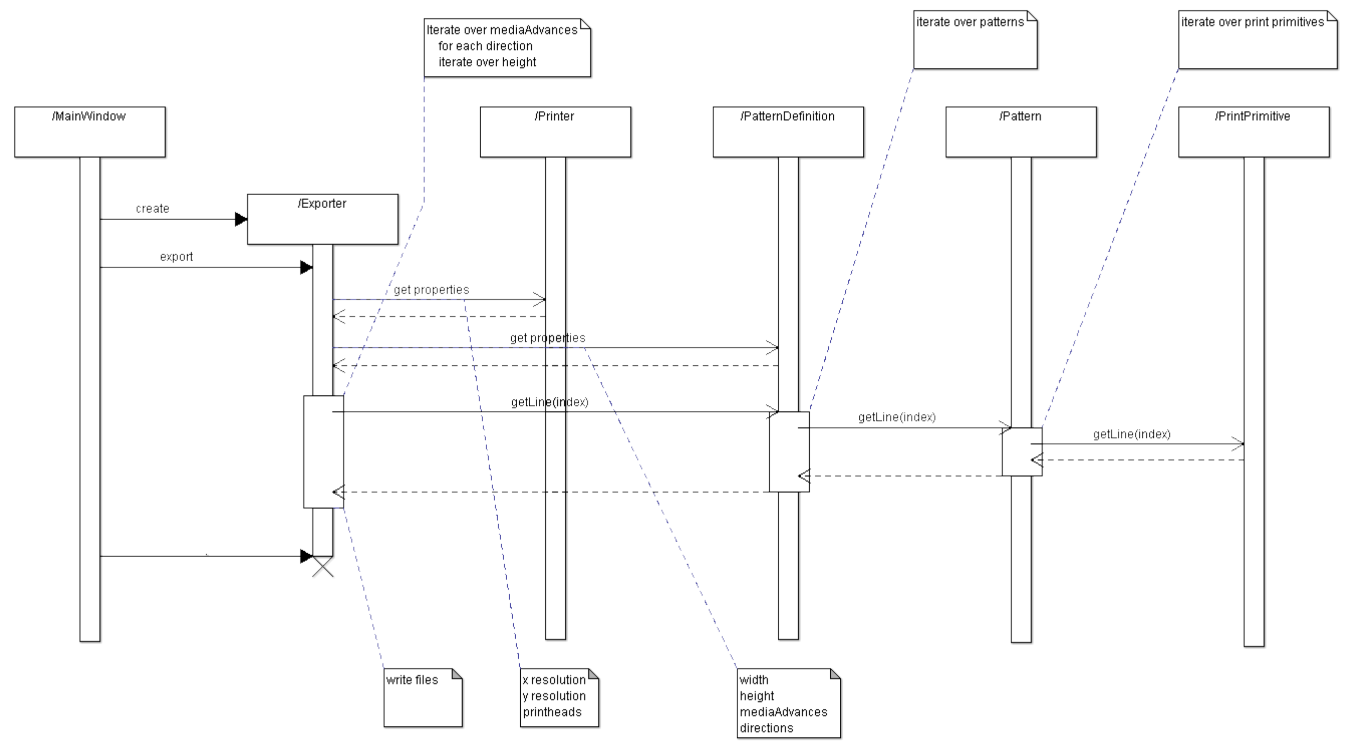


Figure 4.22: Sequence Diagram of Export Workflow

4.4 Main Function and Commandline Execution

Depending on the number of arguments, the main function decides whether the GUI is shown and the TestPatternGenerator starts as usual or the execution should be done without GUI. With the commandline execution, the user story "Export TIFF" defined in 3.2.1. In this case, several arguments are required. The command looks as follows:

```
tpgen.exe <path-to-pattern-definition> <output-path> <name> /
[printer-id]
```

The printer id is optional and could be used to overwrite the target printer specified in the pattern definition. After running this command, the given pattern definition is automatically exported to the specified output path with the specified name.

4.5 Case Study

The export algorithm essentially influences the duration of exporting a test pattern. In this case study, three algorithms are compared. The first one is a naive algorithm where for retrieving the information for each line every print primitive was taken into account. The second algorithm is based on the plane-sweep algorithm which was described in Section 2.5. Since this was still too slow for exporting big test patterns like the SlotOffset to printers with a lot of printheads, multithreading was implemented when fetching the line data from the pattern definition. Because the labels are generated using a QPainter and this does not work outside the GUI-thread, they are moved into a separate tiff file where multithreading was disabled.

The program was tested on a Lenovo T570 laptop with an Intel® Core™ i7-7500U CPU 2.70GHz with four cores and 16 GB DDR4 2400 RAM, running Windows 10 Pro. The measured execution times for the different algorithms and test patterns can be found in Table 4.1. Additional to this it was also tested on a workstation used for the P5 250 HS. It is a Dell Precision 5820 Tower with an Intel® Xeon™-2145 CPU 4.50GHz with eight cores with hyperthreading and 16 GB DDR4 2666 RAM, running Red Hat Enterprise

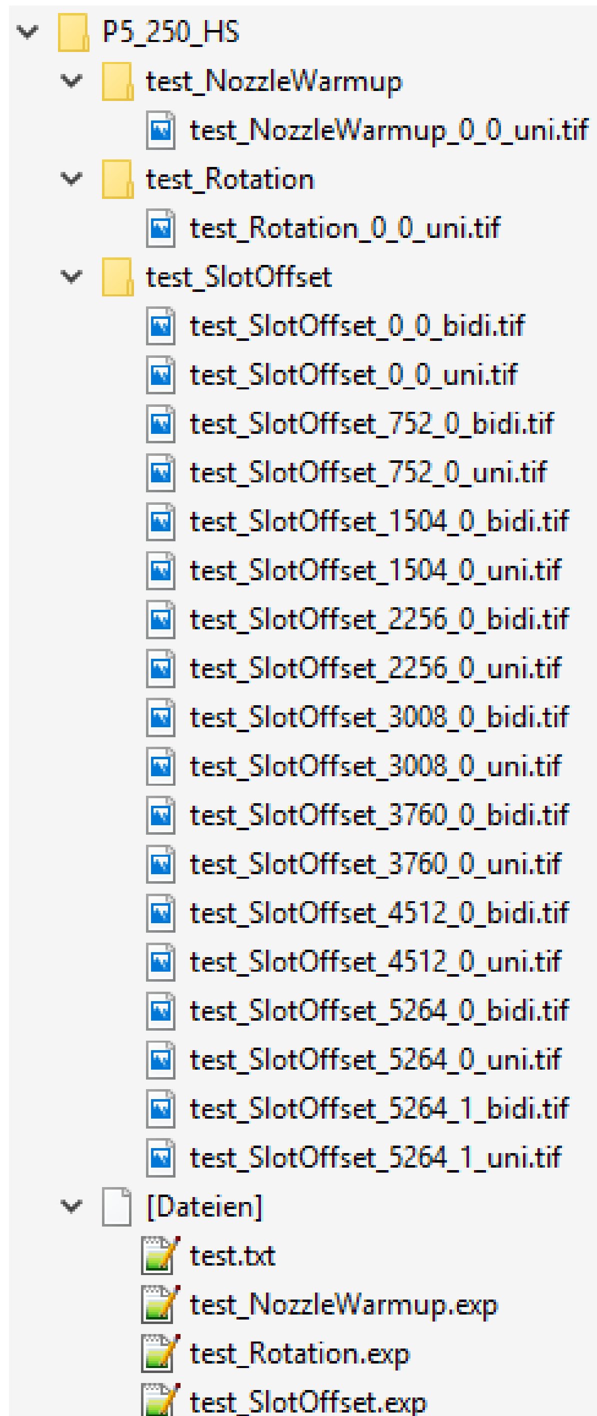


Figure 4.23: Folder Structure of Exported Files

Linux Workstation version 6.2 (Santiago). The measured execution times from the workstation can be found in Table 4.2. The format of the times is hours:minutes:seconds:milliseconds.

The difference between the number of the print primitive of the different test patterns vary very much and so does the execution time. For example, the NozzleWarmup has only a few print primitives so the execution time is in the range of milliseconds. Due to the small execution time of this test pattern, the values are not very meaningful and therefore not considered in further analysis.

The average improvement from the naive algorithm to the plane-sweep algorithm is about 2.5 on the laptop and 2.0 on the workstation. The biggest improvement can be observed for the export of the SlotOffset for the Rho 13xx. This is not only because the SlotOffset is the biggest test pattern regarding the number of print primitives, but also the Rho 13xx is the printer with the most printheads in this case study. But also for the other printheads and test patterns, a significant improvement by the factor of about 2.0 can be seen both for the execution on the laptop and also on the workstation.

The difference from the second to the third algorithm, where multithreading was used, is more significant on the workstation because the workstation uses hyperthreading and the TestPatternGenerator works on 16 cores compared to 4 cores on the laptop. The average improvement on the laptop is about 1.9 and on the workstation 5.8 where the biggest can be observed for the NozzleTest.

One important thing that can be seen when comparing the export of the Rotation for the P5 250 HS on the laptop and the workstation. With the naive algorithm, the workstation with much better hardware is only by a factor of about 1.2 faster than the laptop while the improvement of the plane-sweep algorithm has a factor of about 1.9. It can be concluded that the choice of the algorithm is much more important than the hardware.

Printer	TestPattern	Naive Algorithm	Plane-sweep Algorithm	Threaded Plane-sweep Algorithm
P5 250 HS	NozzleWarmup	0:00:00:616	0:00:00:484	0:00:00:356
P5 250 HS	NozzleTest	0:27:11:837	0:12:47:085	0:07:15:380
P5 250 HS	Rotation	0:12:22:955	0:06:23:246	0:03:11:468
P5 250 HS	SlotOffset	2:26:15:274	0:46:01:189	0:27:28:455
Rho 13xx	NozzleWarmup	0:00:00:578	0:00:00:203	0:00:00:141
Rho 13xx	NozzleTest	1:04:47:752	0:30:57:722	0:12:17:391
Rho 13xx	Rotation	0:28:55:352	0:14:57:365	0:06:40:820
Rho 13xx	SlotOffset	6:11:59:134	1:02:52:582	0:52:25:940
Rho 163 TS	NozzleWarmup	0:00:00:291	0:00:00:161	0:00:00:150
Rho 163 TS	NozzleTest	0:01:09:516	0:00:31:977	0:00:15:419
Rho 163 TS	Rotation	0:00:30:958	0:00:21:970	0:00:12:122
Rho 163 TS	SlotOffset	0:07:09:427	0:03:33:377	0:02:01:325

Table 4.1: Comparison of Algorithms on Laptop

Printer	TestPattern	Naive Algorithm	Plane-sweep Algorithm	Threaded Plane-sweep Algorithm
P5 250 HS	NozzleWarmup	0:00:00:082	0:00:00:080	0:00:00:032
P5 250 HS	NozzleTest	0:22:52:067	0:10:50:750	0:00:51:026
P5 250 HS	Rotation	0:10:04:819	0:05:21:375	0:00:55:002
P5 250 HS	SlotOffset	1:26:10:680	0:35:26:692	0:09:20:338
Rho 13xx	NozzleWarmup	0:00:00:154	0:00:00:079	0:00:00:029
Rho 13xx	NozzleTest	0:55:08:170	0:26:23:034	0:03:13:782
Rho 13xx	Rotation	0:23:47:654	0:12:36:798	0:01:58:348
Rho 13xx	SlotOffset	3:22:47:665	1:21:29:483	0:23:21:304
Rho 163 TS	NozzleWarmup	0:00:00:080	0:00:00:079	0:00:00:029
Rho 163 TS	NozzleTest	0:00:52:708	0:00:26:014	0:00:05:965
Rho 163 TS	Rotation	0:00:22:055	0:00:17:077	0:00:05:826
Rho 163 TS	SlotOffset	0:04:09:472	0:02:46:282	0:00:59:096

Table 4.2: Comparison of Algorithms on Workstation

5 Use Cases

In this chapter, the four use cases which are most likely to occur are explained. The most important one is definitely adding a new printer because Durst produces printers for five segments and permanently improves and adapts them. Other use cases are adding a new property to an existing test pattern or creating a new test pattern. With the creation of a new print primitive, a less common one is also explained.

5.1 Adding a new Printer

The first use case to be described is the need for a new printer support. The software was primarily developed for the latest printer developed, namely the P5 250 HS. As a reference, to show that a large variety of printers could be supported, the Rho 13xx series was also included. During the work on this thesis, a test pattern for a different printer, the Rho 163 TS was needed. Since it was a major requirement to this software, to easily add new printers, this was a very welcome use case to test the flexibility of this tool.

The slot arrangement of this printer is shown in Figure 5.1. As can be seen, it has four different printhead layouts where the colors of the slots are mirrored. On the first printhead, only the slots three and four with the colors red and blue are present, on the second one, all four slots with yellow, magenta, cyan and black are present. The third printhead has the same slot arrangement as printhead two, but the colors are mirrored, the fourth one only has the slots one and two with the colors blue and red.

In total, this printer has eight printheads positioned in two rows as shown in Figure 5.2.

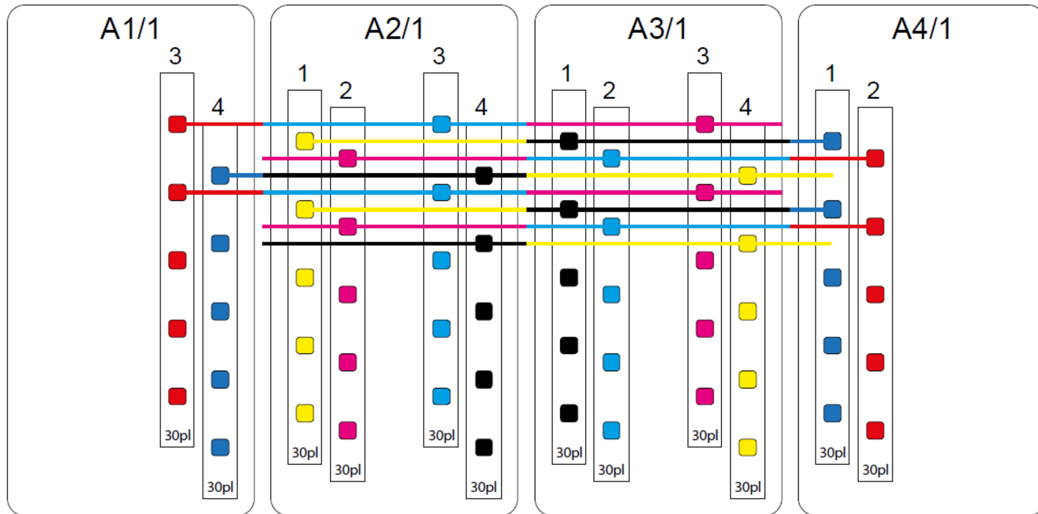


Figure 5.1: Slot Arrangement Rho 163 TS. Image taken from [Dur18c].

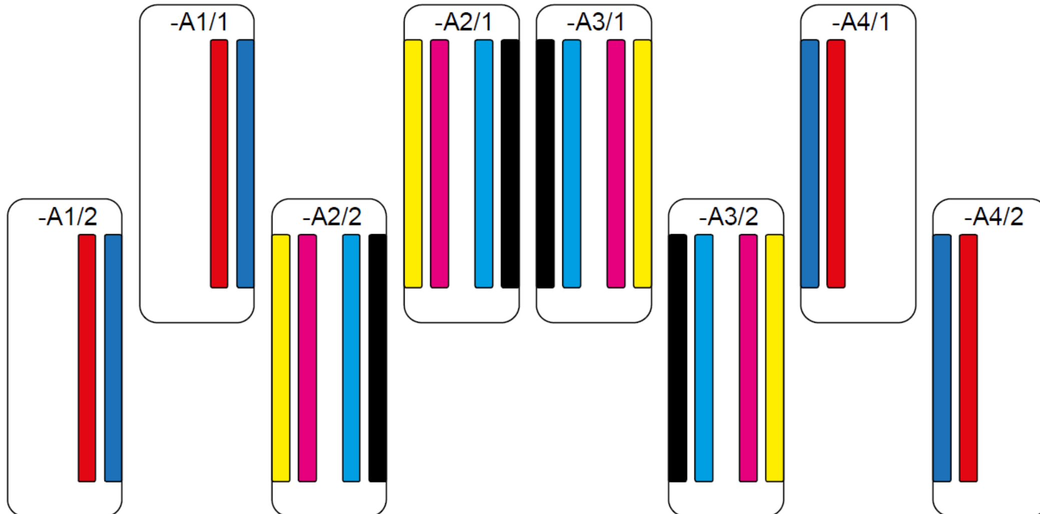


Figure 5.2: Printhead Configuration Rho 163 TS. Image taken from [Dur18c].

The printer configuration for the first two printheads is shown in Figure 5.3. As can be seen, the X position of the first slot from the first printhead is not zero, as in the other configurations, but two. This is because there are only the slots two and three available on this printhead. The other printheads are defined equivalently.

Adding this printer to the TestPatternGenerator is as easy as clicking on the "Printer Configuration" in the Import menu and selecting the new printer configuration file in the file dialog.

Creating this printer configuration took about two hours including the testing if everything is correctly configured and all patterns work properly with this configuration.

5.2 Adding a Property to a Test Pattern

Another use case that appeared after finishing the implementation was to add the possibility to set the width of the lines from the SlotOffsetPattern. The original pattern (a) and with the one-pixel lines (b) is shown in Figure 5.4.

Fortunately, changing the pattern was not that much effort. The costly part, the calculation of the positions of the different parts of the test pattern was not affected and setting the lines for the `LineBlocks` just needed a small change namely adding a loop over the lineWidth and subtracting the lineWidth from the space between the different lines. The listing with the changes can be found in Listing 5.1.

```
22  <printhead id="A1/#X1_">
23      <geometry x="0" y="0" width="4" height="2048" pixelSpacing="8"/>
24      <alignment-references>
25          <ref name="slotOffset" reference="A2/1"/>
26      </alignment-references>
27      <nozzlerow id="RB#X1_S1" color="red" nozzleCount="256">
28          <geometry x="2" y="0" width="1" height="256"/>
29      </nozzlerow>
30      <nozzlerow id="RB#X1_S2" color="blue" nozzleCount="256">
31          <geometry x="3" y="6" width="1" height="256"/>
32      </nozzlerow>
33  </printhead>
34  <printheads id="A1/#X1_" reference="A1/1" position="bottom"/>
35

37  <printhead id="A2/#X2_">
38      <geometry x="4" y="0" width="4" height="2048" pixelSpacing="8"/>
39      <alignment-references>
40      </alignment-references>
41      <nozzlerow id="YM#X2_S1" color="yellow" nozzleCount="256">
42          <geometry x="0" y="2" width="1" height="256"/>
43      </nozzlerow>
44      <nozzlerow id="YM#X2_S2" color="magenta" nozzleCount="256">
45          <geometry x="1" y="4" width="1" height="256"/>
46      </nozzlerow>
47      <nozzlerow id="CK#X2_S1" color="cyan" nozzleCount="256">
48          <geometry x="2" y="0" width="1" height="256"/>
49      </nozzlerow>
50      <nozzlerow id="CK#X2_S2" color="black" nozzleCount="256">
51          <geometry x="3" y="6" width="1" height="256"/>
52      </nozzlerow>
53  </printhead>
54  <printheads id="A2/#X2_" reference="A2/1" position="bottom"/>
55
```

Figure 5.3: Printer Configuration of Rho 163 TS

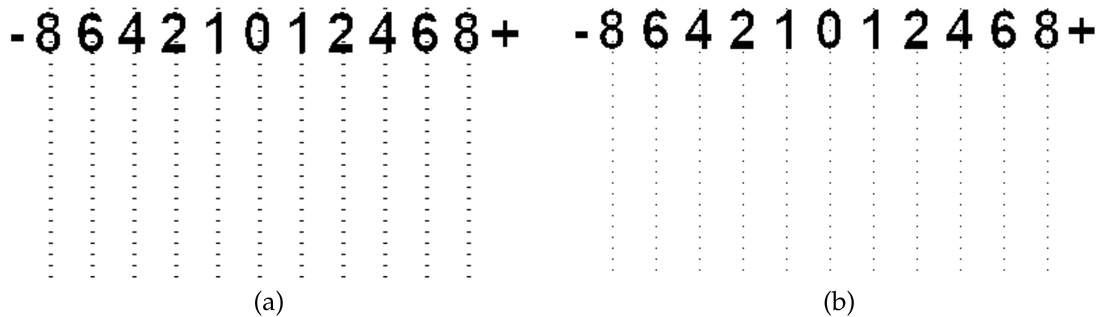


Figure 5.4: SlotOffsetPattern original and with 1px lines

```
for (int i = 0; i < 11; ++i) {
  for (int k = 0; k < m_lineWidth; k++) {
    lines << j++;
    colorChannels << colorChannel;
  }

  j += (31 - 2 * offset);
}
```

Listing 5.1: Changes in SlotOffset Pattern

Since the serialization and deserialization of the different classes are done using Qt's property system, adding a new property did not entail changes in other classes as the pattern itself.

5.3 Creating a new Print Primitive

A new print primitive should be derived from `PrintPrimitive`. The only methods which has to be implemented are `getLine` which returns the data for exporting, and `calculateBoundingRect` which is used to set the dimensions for the bounding rectangle. It is also necessary to call the setters for direction, mediaAdvance and boundingRect from the base class in the constructor with the proper values.

There is no need to register it somewhere. Since the different patterns are part of the software and have to be compiled with it (no dynamic loading), it is guaranteed that all print primitives, used by the pattern, are available. The only time, where it could be critical is when a pattern definition is stored with a newer version of the TestPatternGenerator containing a new pattern, and it is opened later by an older version where this specific pattern is not available. But this problem is targeted with the `PatternFactory`. When deserializing a pattern definition, the different patterns specified in the definition file are created with the factory which returns no object if the specified pattern is not registered.

5.4 Creating a new Test Pattern

The last use case which occurred was creating a new test pattern. The aim of the NozzleTest is to check all nozzles if they work properly. The implementation of this test pattern is straightforward, subclass it from PrintheadPattern, implement the serialization interface and the initialization method.

The pattern itself is quite easy, as for each nozzle just one line is printed. For testing purposes, it is important, that the different printheads can be identified. This is done by adding labels and the first and last nozzles of a printhead print longer lines. A part of the test pattern exported for the Rho 163 TS is shown in Figure 5.5.

Adding this newly generated test pattern is as easy as adding a new entry in `registerPatterns` in the `PatternFactory` as shown in Listing 5.2.

```
QString name =
    NozzleTestPattern::staticMetaObject.className();

m_patterns.insert(
    name.split(":").last().split("Pattern").first(),
    new NozzleTestPattern());
```

Listing 5.2: Pattern Registration in PatternFactory

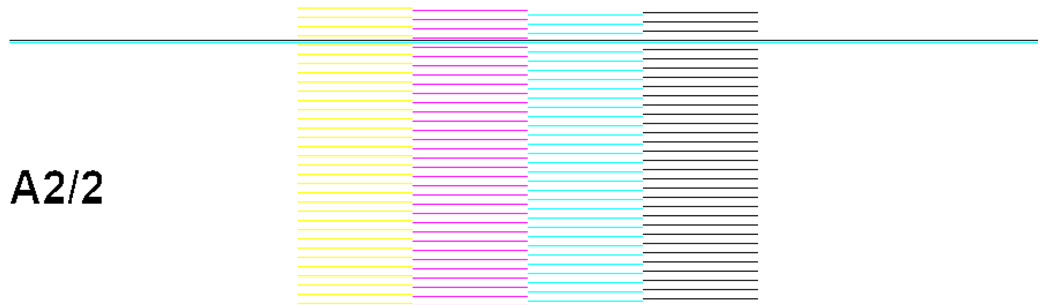


Figure 5.5: NozzleTest Pattern

The member variable `m_patterns` is a QMap where the key is a QString and the value is of the base type `Pattern`. Since the key must be unique, the class name of the test pattern, without the namespaces and the appendix "Pattern", was selected.

Implementing this new pattern (including the new print primitive, timestamp, and testing it) took about half a day.

6 Conclusion and Future Work

In this master thesis, we have designed and implemented a TestPattern-Generator used for printhead adjustments of various digital printing systems. The main target was to create a flexible and easy to use software. This was achieved by separating the information of the printers and the test patterns. To have the possibility to add new printers without recompiling the software, their definition is provided through configuration files in XML format. The test patterns are composed of different print primitives which contain the main functionality for retrieving the data to export it. The central part is the pattern definition which contains the test pattern and can be stored and loaded.

Since during this thesis the design and the implementation of the basic system for test pattern creation was developed, there are still some parts open to implement.

The most important one is to implement the missing test patterns for the printhead adjustment. Only with a complete set of test patterns the solution gains the whole benefit. The next testpatterns to implement are the following:

- Y-Alignment
- Y-Distance
- Voltage Match

The next extension would be the possibility to create your own patterns using the print primitives. For this purpose, a new component, e.g., pattern designer, would be suitable. With this designer, an experienced technician could create his own patterns. With this possibility, it would be easy to target future needs without recompiling the whole software. And if there is also the possibility to export and import new self-defined test patterns, it

would be possible to share them with other users or, if the test pattern is generally needed, to quickly integrate it into the software.

One last extension is another export function. Currently, effort at Durst is made to automate the evaluation of some patterns. For this purpose, camera systems are used to take photos of the printed pattern. To analyze and get information out of them, a definition of the printed pattern is needed. This definition is currently supplied by XML files with a specific structure. An XML-exporter which creates this definition files should be added to the software.

The integration of the TestPatternGenerator into the existing printer software is also a pending task. For this purpose, all required test patterns should be implemented and tested and the printer configurations for the different systems have to be created.

Bibliography

[Aic15] Oswin Aichholzer. “Entwurf & Analyse von Algorithmen.” University Lecture. 2015 (cit. on pp. 11, 12).

[Ass92] Adobe Developers Association. *TIFF*. [Online; accessed 25. Sep. 2018]. June 1992. URL: https://www.adobe.io/content/udp/en/open/standards/TIFF/_jcr_content/contentbody/download/file.res/TIFF6.pdf (cit. on p. 8).

[BEN09] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. [Online; accessed 2. Oct. 2018]. Oct. 2009. URL: http://yaml.org/spec/1.2/spec.pdf (cit. on p. 7).

[Ben86] Jon Bentley. *Programming Pearls*. New York, NY, USA: ACM, 1986. ISBN: 0201500191 (cit. on p. 11).

[Bra+08] Tim Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. [Online; accessed 2. Oct. 2018]. Nov. 2008. URL: https://www.w3.org/TR/xml (cit. on p. 7).

[Bus+96] Frank Buschmann et al. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 978-0-471-95869-7 (cit. on pp. 20, 21).

[Des18] Object Oriented Design. *Factory Pattern*. [Online; accessed 26. Sep. 2018]. May 2018. URL: https://www.oodesign.com/factory-pattern.html (cit. on p. 14).

[Dur18a] Durst. “P5 250 HS - Service Manual Print head GMA.” 2018 (cit. on pp. 2, 57).

[Dur18b] Durst. “Rho 1012, Rho 1030, Rho 1312, Rho 1330 - Service Manual Print head.” 2018 (cit. on p. 3).

[Dur18c] Durst. "Rho 163 TS, Rho 163 TS HS - Service Manual Print head." 2018 (cit. on p. 72).

[Eck00] Bruce Eckel. *Thinking in C++, Volume 1, 2nd Edition*. Addison-Wesley Professional, Jan. 2000. ISBN: 978-0321334879 (cit. on p. 23).

[Fou18] Interaction Design Foundation. *KISS (Keep it Simple, Stupid) - A Design Principle*. [Online; accessed 25. Sep. 2018]. Sept. 2018. URL: https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle (cit. on p. 6).

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0-201-63361-2 (cit. on pp. 14–21).

[Int17] ECMA International. *The JSON Data Interchange Syntax*. [Online; accessed 2. Oct. 2018]. Dec. 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (cit. on p. 7).

[KB11] Kaveh Keshavarzi and Thomas Bayer. *XML, JSON und YAML im Vergleich*. [Online; accessed 7. May 2018]. July 2011. URL: https://www.predic8.de/xml-json-yaml.htm (cit. on p. 7).

[KG08] Foutse Khomh and Yann-Gaël Guéhéneuc. "Do Design Patterns Impact Software Quality Positively?" In: *2008 12th European Conference on Software Maintenance and Reengineering*. Apr. 2008, pp. 274–278. DOI: 10.1109/CSMR.2008.4493325 (cit. on p. 22).

[KG18] Foutse Khomh and Yann-Gaël Guéhéneuc. "Design patterns impact on software quality: Where are the theories?" In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pp. 15–25. DOI: 10.1109/SANER.2018.8330193 (cit. on p. 22).

[Ltd17a] The Qt Company Ltd. *QScopedPointer Class | Qt 4.8*. [Online; accessed 26. Sep. 2018]. Dec. 2017. URL: http://doc.qt.io/archives/qt-4.8/qscopedpointer.html (cit. on p. 24).

[Ltd17b] The Qt Company Ltd. *The Property System | Qt 4.8*. [Online; accessed 26. Sep. 2018]. Dec. 2017. URL: http://doc.qt.io/archives/qt-4.8/properties.html (cit. on p. 24).

[Nor00] James Norton. "Dynamic Class Loading in C++." In: *Linux J.* 2000.73es (May 2000). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=348902.349180 (cit. on p. 6).

[NP82] Jürg Nievergelt and Franco P. Preparata. "Plane-sweep Algorithms for Intersecting Geometric Figures." In: *Commun. ACM* 25.10 (Oct. 1982), pp. 739–747. ISSN: 0001-0782. DOI: 10.1145/358656.358681. URL: http://doi.acm.org/10.1145/358656.358681 (cit. on p. 11).

[Nur+09] Nurzhan Nurseitov et al. "Comparison of JSON and XML data interchange formats: a case study." In: *Caine* 9 (2009), pp. 157–162 (cit. on p. 7).

[PJ98] Jens Palsberg and C. Barry Jay. "The essence of the Visitor pattern." In: *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*. Aug. 1998, pp. 9–15. DOI: 10.1109/CMPSAC.1998.716629 (cit. on p. 20).

[Ref18a] C++ Reference. *PImpl*. [Online; accessed 11. Oct. 2018]. Sept. 2018. URL: https://en.cppreference.com/w/cpp/language/pimpl#cite_note-1 (cit. on p. 23).

[Ref18b] C++ Reference. *The rule of three/five/zero*. [Online; accessed 12. Oct. 2018]. Oct. 2018. URL: https://en.cppreference.com/w/cpp/language/rule_of_three (cit. on p. 24).

[Rie+97] Dirk Riehle et al. *Serializer*. [Online; accessed 27. Sep. 2018]. 1997. URL: https://www.ubilab.org/publications/print_versions/pdf/plop-96-serializer.pdf (cit. on p. 13).

[Rie11] Dirk Riehle. "Lessons Learned from Using Design Patterns in Industry Projects." In: 2 (Jan. 2011), pp. 1–15 (cit. on pp. 20, 22).

[SS18] Bjarne Stroustrup and Herb Sutter. *C++ Core Guidelines*. [Online; accessed 12. Oct. 2018]. Sept. 2018. URL: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rc-zero (cit. on p. 24).

[Sys17] Simple Systems. *LibTIFF - TIFF Library and Utilities*. [Online; accessed 25. Sep. 2018]. Dec. 2017. URL: http://www.simplesystems.org/libtiff (cit. on p. 8).

[SYS18] AWARE SYSTEMS. *TIFF Tag Reference*. [Online; accessed 25. Sep. 2018]. May 2018. URL: https://www.awaresystems.be/imaging/tiff/tifftags.html (cit. on pp. 8, 9).

[Wik17] Qt Wiki. *D-Pointer*. [Online; accessed 8. May 2018]. Aug. 2017. URL: http://wiki.qt.io/D-Pointer (cit. on p. 23).

[Wik18] Qt Wiki. *Smart Pointers*. [Online; accessed 11. Oct. 2018]. Oct. 2018. URL: https://wiki.qt.io/Smart_Pointers (cit. on p. 23).