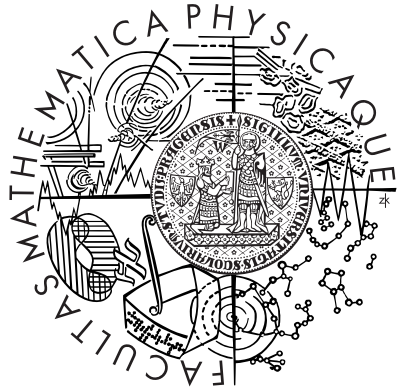
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Černý

Sewing pattern editor

Department of Software Engineering

Supervisor: RNDr. Ondřej Šerý

Study program: Computer science, General Computer Science

2009

I would like to thank the supervisor of this work, RNDr. Ondřej Šerý, for being capable of asking the right questions in the right moment and for his tolerance to my poor time management skills.

I thank Jana Minaříková for the very idea of this work and for support and consulting during the development.

I hereby declare that I wrote the thesis myself using only the referenced sources.

I agree with lending the thesis.

Prague, May 28, 2009

Martin Černý

Table of Contents

1 Introduction....6
1.1 Goal statement....7
1.2 Existing solutions....7
1.3 Chapter summary....8
2 Analysis....9
2.1 Basic concepts....9
2.1.1 Initial observations....9
2.1.2 Drawing and grading....10
2.1.3 Dependency....10
2.1.4 Measures and distances....11
2.2 Objects to represent....12
2.3 Construction layer....13
2.3.1 Overview....13
2.3.2 Point dependencies....14
2.3.3 Conclusions....15
2.4 Pattern Layer....16
2.5 Tracing the outline....17
2.6 Pattern adjustments....18
2.7 Transformations....19
2.8 Analysis conclusions....19
3 Developer documentation....20
3.1 General....20
3.2 Namespaces....20
3.3 Utilities....21
3.3.1 Serialization....21
3.3.2 Structured cloning....23
3.3.3 Other utilities....24
3.4 Pattern objects....25
3.4.1 The dependency model....26
3.4.2 Other general properties....27
3.4.3 User objects....27
3.4.4 Pattern, pattern part, layers, sublayers....28
3.4.5 Layer bound objects....29
3.4.6 Points....30
3.4.7 Defined points, point definition....32
3.4.8 Point proxy....35
3.4.9 Semantical equivalence, point chain....36
3.5 Pattern adjustments....37
3.5.1 Splitting parts....37
3.5.2 Preserving structure, transformations....38
3.6 Graphic and geometric algorithms....40
3.7 User interface....42

3.7.1 Controls and forms....42
3.7.2 Selection, Highlights....43
3.7.3 Editor tools....44
3.7.4 Actions....44
3.7.5 Undo/redo....45
3.7.6 Printing....45
3.8 Resources....45
4 Conclusion....47
4.1 Achievements....47
4.2 Directions for future development....47
5 References....48
6 Appendixes....50
6.1 Appendix A - CD Content....50
6.2 Appendix B - Introduction to sewing pattern design....51
6.3 Appendix C - Terminology....54

Název práce: Editor textilních střihů
Autor: Martin Černý
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Ondřej Šerý
e-mail vedoucího: ondrej.sery@dsrg.mff.cuni.cz

Abstrakt: Konstrukce střihů je základní součástí oděvního návrhářství. Předmětem této práce je implementace editoru textilních střihů, který by vyhovoval potřebám nezávislých oděvních návrhářů a výrobců zakázkového šatstva. Práce v editoru je velmi blízká ručnímu navrhování střihu a program je tedy pro návrháře snadno použitelný. Implementace pokrývá nejdůležitější oblasti navrhování střihů. Střihy v programu vytvořené mohou být použity pro různé zákazníky, protože se automaticky přizpůsobují zadaným tělesným mírám.

Klíčová slova: Střih, oblečení, oděvnictví, CAD, šití

Title: Sewing pattern editor
Author: Martin Černý
Department: Department of Software Engineering
Supervisor: RNDr. Ondřej Šerý
Supervisor's e-mail address: ondrej.sery@dsrg.mff.cuni.cz

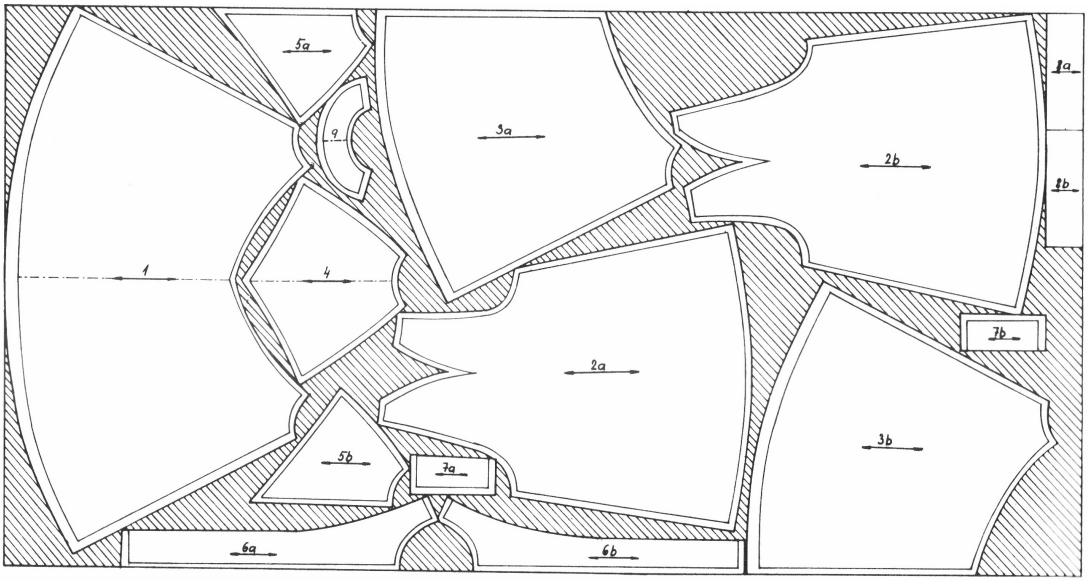
Abstract: Sewing pattern construction is a fundamental part of garment design. The subject of this thesis is the implementation of an editor of sewing patterns, that would suite the needs of independent apparel designers and custom apparel manufacturers. The editor follows closely the procedure of manual pattern design and thus is easy to use for the designer. The final program covers the most vital steps in pattern design. Patterns created in the program can be reused for different customers, since they automatically adjust to given body measures.

Keywords: Sewing pattern, clothing, apparel, CAD, sewing

1 Introduction

The notion of sewing pattern was introduced to european apparel production in the 17th century [1]. It allowed the dressmakers to produce garments that fitted better the shape of human body and thus were more comfortable for the customer. It has opened the way for serial production of clothing, because it made it possible to produce garments that would fit a specific range of body sizes without measuring the actual customers. In today's practice, production of virtually all world's clothing starts with a sewing pattern.

The basic aspect of a sewing pattern is the outline of one or several pieces of fabric that constitute the final garment - these are the **parts** of the pattern. These are accompanied with several kinds of marks and other notes that provide further information for the garment maker (Fig. 1).



26/ Položení dílů halenky na tkaninu

legenda: šíře materiálu – 120 cm
spotřeba – 233 cm
1 – spodní část zadního dílu
2a, b – rukávy
3a, b – spodní část předního dílu
4 – zadní sedlo
5 – přední sedla
6a, b – krajové podsádky
7a, b – nástavky k podsádkám
8a, b – manžety
9 – průkrčníková podsádka

Fig. 1: A sewing pattern, picture taken from [2].

One of the key concepts of pattern design is the **dart**. Darts are triangular areas, usually with curved sides (as seen above). The opposite sides of a dart are sewn together to provide a three-dimensional shape to the garment. Although the garment itself is three-dimensional, the whole pattern design process is done in two-dimensional space. Mainly because it is much more simple and people with pattern design skills have no trouble projecting how the fabric will behave when sewn together.

At least here, in Czech Republic, every apparel design student learns how to design patterns by hand. Although computers are used in pattern design, they are mostly utilized in large companies, where the patterns have to be digitized, so that they could be used as an input to fabric cutters and other machines involved in mass apparel production. In custom apparel production, many designers still construct sewing patterns by hand. For a more detailed introduction into the pattern design process, see Appendix B.

1.1 Goal statement

The goal of this work is to create a program for sewing pattern design, aimed at custom apparel manufacturers. The program has to allow the user to work as close as possible to manual pattern design, while preserving the benefits of digital processing. The key features from the user's point of view are sloper design, pattern adjustments, printing, and generic CAD benefits (e.g. undo/redo, ...).

Since the process of constructing a sewing pattern is parametrized by body measures of the customer, the program should be able to resize a designed pattern to different measures, without further assistance from the user.

1.2 Existing solutions

Because software for sewing pattern design is not used massively among general public, the software manufacturers usually do not release demo versions of their products. Thus the product functionality can only be inferred from a provided product data sheet or documentation and the comparison below is hence lacking exact information.

Existing solutions for sewing pattern design can be divided into three categories:

1) Complex products for big companies

   Software aimed at mass production of clothing. The products have rich feature set and are designed to integrate well with the clothing industry machines. Three main players in this field are Gerber technologies [3], Lectra [4], and Optitex [5]. Aside from standard pattern making procedures, all of these offer products for 3D pattern visualization on virtual human body and simulation of the fabric fit and drape. Optitex even advertises a product that is capable of simulating and visualizing the garment behavior on moving humans.

   The features of program discussed here cannot be compared with such products, but this is not its aim. Although 3D visualization is a very fancy feature, it is not targeted at pattern designer, but on the company management, since the designer always knows how the garment he is designing will look like in reality.

2) Products with comparable target audience

    1. Audaces Apparel 8 [6]

       Although Audaces is still more focused on the clothing industry than custom manufacturers, it provides features for custom fitting. It also offers "automatic grading" of a pattern. Whether this means that the pattern adjusts to given measures of a customer without any further input from the user is unclear from the documentation.

    2. Wild Ginger Cameo 4 [7]

       The only complex program that aims specifically at custom clothing manufacturers the author of this work has found. It is also the only complex solution that has more than a simple product sheet publicly accessible. The program lacks the whole notion of path – every pattern is only a set of lines and curves. Every time the user wants to perform an adjustment, he has to explicitly select segments he wants the adjustment to be applied to. Only the set of slopers bought with program is capable of changing to measures of the customer. For adjusted patterns or patterns created from scratch, grading rules (i.e. modifications for each size specifically) have to be defined.

    3. Several other products exist that vary in both functionality and price between the two mentioned above.

3) Smaller products

   These products do not provide full fledged functionality of a CAD system, they are instead only a library of slopers/patterns that adjust to given measures, but cannot be altered.

1.3 Chapter summary

The text is organized into following chapters:

Chapter 2 discusses in depth the specifics of pattern design and possible approaches to the implementation of the program. It is shown that structures proposed for representation of a scalable pattern are sufficient.

Chapter 3 contains the developer documentation, i.e. the way the structures proposed in Chapter 2 are implemented in the program.

Chapter 4 summarizes the achievements and suggests direction for further improvements.

2 Analysis

The nature of pattern design may suggest that it could be implemented as simple vector graphics with a few specific functions. However, the fact that the pattern should adjust automatically to modified body measures has far reaching implications on the approach that may be chosen to successfully implement such a design program. The specific requirements and methods to satisfy them are discussed in this chapter.

2.1 Basic concepts

2.1.1 Initial observations

Consider two basic use cases for the program:

1) Drawing a sloper from scratch
2) Modifying a sloper/pattern

Drawing a sloper by hand starts with drawing a **construction net**. This is always a simple ruler and compass construction. Main "net" usually consists of important lines on human body, both horizontal (waistline, bustline) and vertical (back line, side line). The net contains only points and line segments, there are no curves or arcs. A line segment that is part of the construction net is a **construction line.**

Then the **outline** of the pattern is traced over the construction net. The outline is always a closed vector path. While many segments of the outline are lines that simply copy a part of the construction net, some are curves. The curved segments always connect two vertices of the construction net. The pattern design literature usually contains only an image of the final sloper and does not describe the exact shape of the curved segments. The pattern designer relies on his or her experience to trace the curves properly. Outline of multiple pattern parts of the pattern may share the same construction net (e.g. the front and back part of a skirt).

Adjustments of a pattern part are easily described in terms of geometry. A typical adjustment starts with dividing the pattern in two pieces, applying a transformation on one of the parts and joining the parts back together (not necessarily in the same way they were joined before), possibly adding some extra fabric in between.

Some other data may be incorporated in the pattern (e.g. marks to indicate where parts of the pattern are sewn together). This data consists mostly of textual and/or graphic notes and modifications to the visual appearance of individual objects in the pattern. Such information may be easily incorporated in any model chosen for the outline of the pattern and the construction net, and thus will not be discussed here.

An important feature of patterns of the same garment for different sizes is that they are quite similar in shape. So it can be assumed that following properties of objects in the construction net and/or pattern outline are preserved over all real world measure values:

1) The number of intersections of a pair of lines, line segments, and/or circles.
2) Two distinct points will remain distinct
3) A point in half plane defined by a construction line will remain in the same half plane

2.1.2 Drawing and grading

If a subset of a pattern (or the whole pattern) adjusts automatically to given measure values, such a subset will be called **scalable**. The set of measure values that are associated with the pattern will be referred to as **scale**.

The process of drawing a sloper in the program should be as close as possible to the way it is drawn by hand, thus the pattern editor has to provide the user with tools to create the construction net and to draw the outline tracing the net.

The pattern should be scalable. If the construction net was scalable, then only little extra information on the shape of curved segments needs to be added, and the outline itself would be scalable. The most straightforward way of automatic computation of the construction net is to store the complete geometric construction that defines the net. Since this approach can be easily implemented, it is the method that will be used.

While the editor adjusts a pattern, it should preserve its scalability. Once again, the most straightforward approach is to retain the exact modifications that were used. I.e. retain the adjustment method that transformed the pattern and parameters of the adjustment. Whenever the underlying pattern changes, the same adjustment is applied to the modified pattern. Although this cannot be implemented as easily as with the construction net, it is the method that will be used. Any attempt to further simplify the calculation of the resulting shape requires the program to be able to operate symbolically with analytical descriptions of the pattern outline and/or construction net. The program would thus require a strong mathematical engine, which is far beyond the scope of this work.

2.1.3 Dependency

Dependencies among objects are a general concept that will be used all along this text. Let us define several terms:

An object *A* is **dependent** on an object *B* if the state (especially position) of *A* depends directly on the state of *B*, i.e. there are no “intermediate” objects. In this situation the object *B* is **superior** to *A*.

If the dependence is not direct (there are intermediate objects) then *A* is **indirectly dependent** on *B* and *B* is **indirectly superior** to *A*.

If *A* is not directly or indirectly dependent on *B*, then *A* is **independent** of *B*. Consider a set of objects *S* and $A \in S$, *A* is **independent** in *S*, if *A* is independent of all objects $B \in S$. If the context is unambiguous, the term independent will be used without specifying the set *S*, meaning that the object is independent in the whole pattern.

Let us have a set of objects *S*. If there is such *A* and $B \in S$, that *A* is (indirectly) dependent on *B* and vice versa, *S* contains a **circular dependency** on *A* and *B*. If *S* contains no circular dependency, it is **dependency-consistent.**

2.1.4 Measures and distances

For the purpose of this text, **measure** will be regarded as a numeric parameter of the pattern. The pattern has to be scalable, so the objects in the pattern are always indirectly dependent on a set of measures. In apparel design, there are two types of measures used. The actual measures taken from the body of the customer (e.g. full bust, hip size) will be referred to as **standard measures.** There are also helper measures that are calculated from the standard measures, because measuring them would be difficult or inconvenient or they do not directly correspond to a distance on human body (e.g. width of the armscye). These will be referred to as **compound measures**. A compound measure is either a linear combination of standard measures plus a constant (i.e. independent on other measures) addition, or it may refer to a distance between two points in already constructed portion of the pattern.

The algorithms for creating a construction net express distances (length of line segments, radii of circles) either as a linear combination of measures or as a multiple of distance between two points already constructed. The former case will be referred to as a **distance from measures**, the latter as **distance from points**. Both of these types of distances may also contain some constant addition. Noting the similarity between compound measure and distance, it is obvious that a compound measure can be fully defined by a distance.

This representation allows for quite complex structure of distances and measures. If utilized properly, a well crafted hierarchy of measures may significantly simplify the designer's work. From mathematical point of view, only linear combinations of the standard measures can be expressed, as long as the pattern does not contain a circular dependency and does not use a distance from points. If the distance from points is taken into account, any constructible number can be expressed by the means of distances and measures.

In the terms of dependency: A standard measure is independent (i.e. dependent only on its actual value, which is independent). Compound measure is dependent on the associated distance. Distance from measures is dependent on a subset of pattern measures and the constant addition. And distance from points is dependent on two points, the coefficient and the constant addition.

2.2 Objects to represent

Summing up the initial observations, the editor needs to represent these geometrical objects: point, line segment, and curve. Bézier splines will be used to represent curves in pattern, because adjusting the shape of a Bézier curve through its control points is simple and intuitive task and leads quickly to good results. In addition, Bézier curves are standard in graphics software; many users are already familiar with Bézier curves from other products.

All of the above mentioned geometrical objects can be described by one or several points. The start and end point for a line segment, and four control points for cubic Bézier curve. Moreover, affine transformations can be applied on lines and Bézier curves by applying the respective transform to their defining points [8]. For this reason, point will constitute the base object. All other objects will be represented by points and will implement most methods by calling the corresponding method on their defining points.

Apart from representing these basic objects, the editor also needs to group line segments and curves to form paths and to group paths and other objects to form pattern parts. The pattern as a whole is then a group of pattern parts. All objects mentioned above are objects the user can "see" and will think of, they will be referred to as **user objects**. Every user object is accompanied with a name. However, the name is only a label the user associates with the object and carries no meaning for the program itself.

The objects that constitute the pattern will be divided into three logical **layers**. First one is the **construction layer**, which contains the construction net, then the **pattern layer**, containing the outline of the pattern, and the **metadata layer**, which contains special marks and other data, that are important for the pattern, but are not an integral part of the pattern shape. Since pattern parts are independent entities that should contain their whole description, this division will be realized at the pattern part level. In other words, every pattern part contains three sets of objects, one for each layer, but the pattern as a whole does not recognize these layers. The point, construction line, path segment and path are **layer bound objects**. Every layer bound object is a member of exactly one pattern part and exactly one layer.

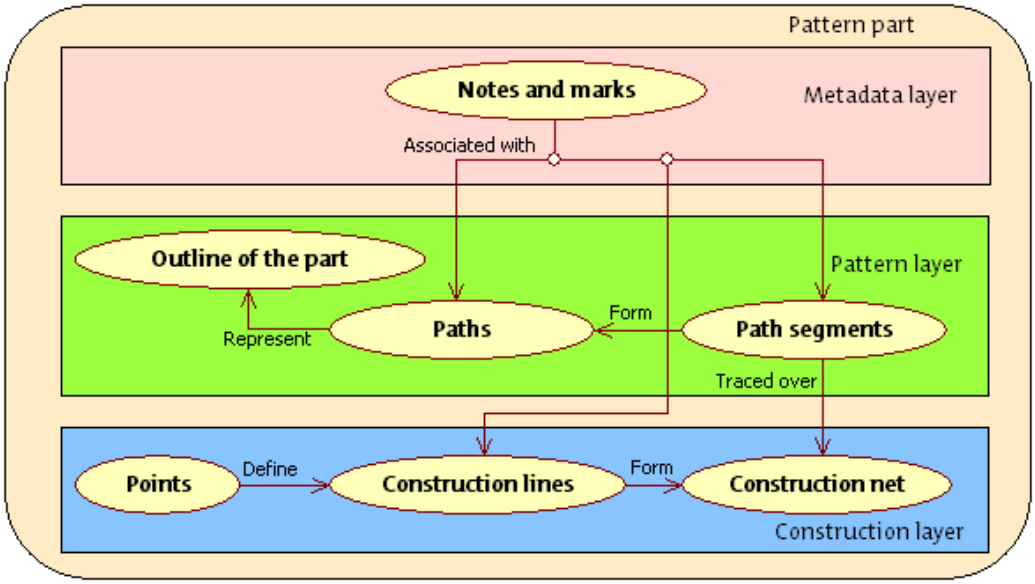


Fig. 2: The layer structure

2.3 Construction layer

2.3.1 Overview

The construction layer has to describe a geometrical construction for the construction net of a pattern part. It will be assumed that the construction starts with a single point whose position is already defined. This point will be referred to as **anchor point** of the pattern part. Since main focus is on points, two types of steps in the construction will be considered. First, constructing a new point using points already constructed, measure values, and possibly other parameters. Point constructed in this way is called **dependent point**. The second type of step consists of drawing a construction line between two points. A construction line is fully described by its endpoints. The two endpoints of the construction line are not interchangeable, because for a subset of operations it is useful to regard the construction line as having an orientation. Thus the line has a specified **start point** and **end point.** However, when the context is unambiguous, both of them will be referred to as endpoints. In terms of dependency, construction line is dependent on its endpoints.

While drawing a construction line is indeed a simple step, the construction of a new point requires further discussion. The exact way how the new dependent point is constructed is described by a **point dependency**. Point dependency is a standalone object, that is dependent on one or several points and/or construction lines. Since a construction line is only a group of two points, it is possible to think of a point dependency as being dependent directly on the endpoints of the construction line.

Because the construction net and hence the points on it have to be scalable, the point dependency may be dependent on one or several distances and possibly on some other data provided by the user (e.g. angle magnitude). Now the dependent point itself is dependent on only one object, the point dependency. The point is **defined** by the dependency and the dependency is the point's **defining dependency** (see Fig. 3). From the user's point of view, a dependent point can be moved only by modifying the properties of its defining dependency or the measures.

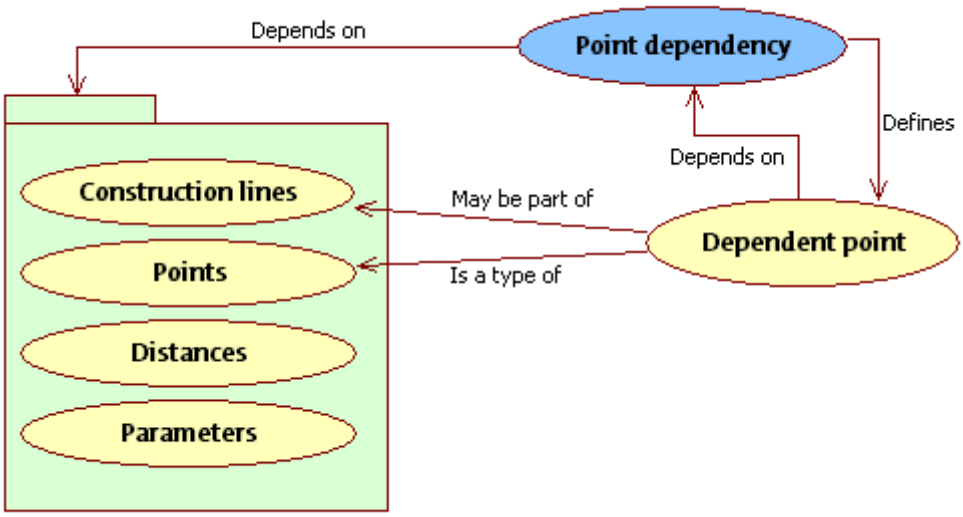


Fig. 3: Point dependency and its relationships to other objects

2.3.2 Point dependencies

In the construction layer, four types of point dependencies are required to represent various steps in the construction. These were chosen to follow closely the construction steps the user is familiar with. The former three of these dependencies are enough to allow for all basic ruler and compass constructions [9], although a circle is always expressed only implicitly through its radius and center. The types of dependencies related to construction layer are listed below:

1) **Line intersection point dependency**

   The point is described as an intersection of two construction lines. For the sake of simplicity and consistency, the resulting intersection is not limited to lie only on the segments specified by construction lines but on the whole lines. In other words, if the construction lines are not parallel, their intersection is always defined, regardless of their length.

2) **Line and circle point dependency**

   The point is described as one of intersections of a construction line and a circle. Among two possible results the right one is described as lying closer to negative or positive infinity in relation to the

orientation of the construction line. This distinction was chosen because it gives intuitively expected results if the position of the line or the circle changes. In practice, there is no sloper design where the orientation of a line would vary more than slightly between construction nets for different measures, supposing these are measures a human can physically have.

3) **Triangle point dependency**

   The point is described as one of the intersections of two circles - or equivalently, as being a third vertex of a triangle with given two vertices and distances from these to the third vertex. Among the two possible results, the right one is described as being on the left hand or right hand side of a ray going from the first point to the second. Reasons for this distinction are similar to these mentioned in connection with line and circle point dependency.

4) **Angle and distance point dependency**

   The point is described as lying on a line in a specified angle (relative to a construction line or y-axis) from another point, in a given distance from the point. In the context of geometry, this dependency is superfluous as it can be derived from the former three if the user is provided with at least two distinct points at the beginning of the construction. However, this dependency allows for simplification of: drawing a line with a specified length, orthogonal to another line; finding a point on a line in specified distance from one of the endpoints; and finding the center of a line (in combination with distance from points).

2.3.3 Conclusions

It was shown that a construction net consisting of an anchor point, a set of dependent points with corresponding dependencies, and a set of construction lines is able to describe any ruler and compass construction and may be scalable.

The construction algorithm is not contained explicitly in the pattern structure, but that is an advantage, because it implies only the really necessary restrictions on the operations the user can perform on the construction net. It is easier to implement as well. Consider a model, where the construction algorithm is represented explicitly by a list of actions with some parameters. Every action takes as input the output from the previous action and adds some objects to it. Such an implementation could have basically the same behavior as the one that was chosen here. However, it would need to introduce similar concepts of dependencies between objects (e.g. to check whether an object may be safely deleted), and the objects representing a step in the construction would correspond with point dependencies mentioned here or a combination thereof. So the explicit

expression of the geometrical construction would constitute only inconvenient overhead while bringing no real value to the user, who does not think about the construction net as a result of a construction but as a set of lines and points.

2.4 Pattern Layer

The most important object in the pattern layer is the **path**. As discussed above, the path consists of linear and cubic Bézier segments. Because a line segment can be easily expressed as a cubic Bézier curve, all path segments will be considered Bézier segments, for the sake of easier implementation. Every Bézier segment is defined by four points, these will be referred to as the **start point**, **control point 1**, **control point 2** and **end point** of the segment.

The path has to be continuous, i.e. the endpoint of a segment must be identical to the start point of the next segment. A path with no segments (only with a start point of the first segment), should be defined, because it will be needed upon the beginning of the tracing process, when the user has chosen the first point of the path and has not chosen the second point yet. Moreover, it makes sense to retain a control point beyond the end of the path (i.e. the control point 1 of a segment not yet appended to the path) so that the newly appended segment will continue "in the same direction" as the last segment. For similar reasons (e.g. when the path is being closed), it is useful to save the control point before the beginning of the path. And, last but not least, in most cases the path should have continuous derivation at the segment endpoints, so both control points have to stay collinear with the segment endpoint. This implies that control point 1 of a segment must "know" about the control point 2 of the previous segment.

Thus the segment will not directly contain all of its points. Instead, **Bézier point** is introduced. A Bézier point is a point that has two other points associated with it, the **in control point** and the **out control point**. The path owns its first point (which is a Bézier point) and a list of segments. Every segment owns only its end point (also a Bézier point) and if it needs the start point, it asks the path and the path provides either the previous segment's endpoint (if there is any) or the first point of the path. The control point 1 is then the out control point of the start point and control point 2 is the in control point of the end point.

The start and end point of a segment will be referred to as the **core of the segment**. Union of the path's first point and all segment's endpoints form the **core of the path**.

In the terms of dependency, a Bézier point is independent of its control points, the path segment is dependent on its start point and end point and also on their out control point and in control point respectively.

2.5 Tracing the outline

The path has to trace the construction net, this means that every point in the core of the path is identical to some point constructed in the construction net. However, it cannot be the same object, because no object can be part of more than one layer. For this reason new kind of dependency is introduced, the **identical point dependency**. It simply states that the dependent point defined by this dependency is identical to another point. Although this could be easily accomplished with angle and distance point dependency, it is useful to have a dedicated object, because this relationship is special and will be handled in a special way by many algorithms modifying the pattern (see Section 3.4.9).

For linear segments of the path, the control points are simply identical to some of the points in the core. If the segment is not linear, the control points will be represented by a **relatively positioned point**. The position of a relatively positioned point is defined by an anchor point and a translation vector, called the **delta**. Note that while this could be expressed by point dependencies, a relatively positioned point is not a dependent point, it is presented to the user in a different way. A dependent point cannot be changed directly, its dependency must be modified instead. Relatively positioned point can be changed on its own, it appears to the user as an independent object.

Problems may arise from the fact that relatively positioned point does not change appropriately when the scale of the pattern is adjusted (e.g. if the distance between endpoints of the segment grows, the distance between the endpoints and their respective control points remains the same and the curve is thus rendered a little more “flat”). However, since the possible real world values for measures are quite limited, the difference is small enough to be covered by the flexibility of the fabric or imprecision in sewing.

In certain constructions it is important that a construction line is the tangent to the path at a specified point. In this case, another type of point is used – the **constrained point**. Constrained point is an independent point, whose position is limited to certain values. The limitation is realized by a **point constraint** object, specifically **line point constraint** in this case. The line point constraint limits the points position so that it always lies on a given construction line, but it still may be moved freely by the user. If this line contains the core point, then the line is a tangent to the path segment.

Constrained point and point constraint are very similar to dependent point and point dependency, however, analogously to the relatively positioned point, it is presented to the user as an object the user can directly manipulate with. Point bounded with line point constraint has its coordinates stored as a coefficient of the distance from line's start point to its end point. So the point scales along with the line when the measures are changed and thus provides slightly better behavior over relatively positioned point when the measures are modified.

2.6 Pattern adjustments

As was already mentioned, all of these adjustments can be separated into two steps: first, the pattern part is split into two **fragments**, than a transform (affine for the former three) is applied to one of the fragments. Except for dart transfer, all adjustments split the part along a construction line, the dart transfer may need to split the part along an angle formed by two construction lines.

Following types of pattern adjustments are considered:

1) Splitting the part in two

   The cut is always a straight line. For simplicity, a construction line will be considered, because in most cases the part is cut along some important line in the construction net, so there is no problem in finding appropriate construction line.

2) Dart transfer

   One of the fragments is rotated along a point on the border of both fragments. The amount of rotation is chosen such that the fragments overlap. The fragments are then joined together.

3) Pleats

   The fragments are moved apart and new fabric is inserted between them. The user specifies how many pleats he wants to insert and their size. The places where the pleats should be folded are marked on the pattern. This adjustment is not implemented in the current version of the program.

Since the whole sequence of the adjustments has to be preserved for the pattern to be scalable, the part is organized into **sublayers**. Every sublayer contains objects in all three layers and represents complete state of the pattern part. The topmost sublayer is presented to the user as the resulting pattern part. The initial sloper lies in the first sublayer. Every pattern adjustment than produces two new sublayers. The first one contains the part after it was cut into fragments, the second contains the transformed part.

To preserve as much as possible of the relationships between layers and to produce a neat structure, the new sublayers are constructed such that:

1) Every layer bound object is a member of exactly one sublayer.
2) Points in construction layer and control points of paths are dependent only on objects in the same sublayer or on objects in the sublayer directly below.
3) Layer bound objects except for those mentioned in 2) are dependent only on points in the construction layer in the same sublayer.

If the above conditions are met, the pattern part is **sublayer-consistent.**

Now the adjustment is done in a simple way. The part is cut – that means most of the points in the new sublayer are identically dependent to their counterparts in the sublayer below, new points have been created in the construction layer for the intersections of construction lines and paths with the cut. The relevant paths and construction lines have been split in two.

After that, one fragment is once again transferred identically to the second new sublayer. The other has all of its points transformed and then inserted to the second new sublayer. Copies of construction lines and line segments that contained the original points are created over the transformed points and inserted into the second new sublayer.

2.7 Transformations

Pattern adjustment require new type of point dependency – the **transformed point dependency**. This is a dependency that applies a transformation on the given point, where the transformation is given by a **transformation context**. While multiple points cannot share common point dependency, they can share a common transformation context. For example, the dart transfer produces only one transformation context describing the rotation of the whole fragment round the dart center. Now only one value may be updated to change the amount of rotation of the fragment.

2.8 Analysis conclusions

It was shown that proposed structures are strong enough to represent a pattern that can be adjusted and preserve scalability. If the user for some reasons does not want to preserve this ability (e.g. the user does not want to express all distances by measures), the structures proposed allow for a static design as well by simply omitting some of their features.

3 Developer documentation

Developer documentation discusses concrete implementation of structures proposed in Chapter 2 as well as the implementation of GUI and tools of the editor. Objects, structures, relationships and processes of the program are discussed on conceptual level. For detailed description of concrete methods and properties of implemented objects, see reference documentation on the attached CD.

3.1 General

The program is written in C#, using Windows Forms library for GUI. Target platform is .NET Framework 2.0 [10]. The project was developed and compiled under Microsoft Visual C# 2008 Express edition [11].

The program makes use of XPTable library [12] - an advanced list component for .NET by Mathew Hall. Redistribution, modification and use of the library is permitted by the author, if the redistribution retains original copyright notice and disclaimer. Minor modifications were applied to the library – several bugfixes were introduced and the handling of number columns has been extended.

3.2 Namespaces

The program is organized into several namespaces:

1) **PatternEditor**

   The general namespace. Contains the top level GUI components and their helper classes.

2) **PatternEditor.Actions**

   Actions the user can perform in the editor – only actions that are not provided by editor tools use this mechanism. Contains the `Action` class and its descendants and helper classes.

3) **PatternEditor.Controls**

   Contains custom GUI controls (descendants of `Windows.Forms.Control`) developed for the project.

4) **PatternEditor.Forms**

   Contains forms (dialogs) designed for the editor.

5) **PatternEditor.Geometry**

   Contains classes that perform geometrical computations, most important being the `Plane` static class.

6) **PatternEditor.PatternObjects**

   The largest namespace. Contains the structure that represent all objects that are part of the pattern, as well as several helper classes. All objects mentioned in the Chapter 2 reside there. The Points subdirectory is introduced only for organizational purposes and carries no semantical meaning, all objects in this subdirectory are part of the `PatternEditor.PatternObjects` namespace.

7) **PatternEditor.Resources**

   Namespace managed by the VS 2008 IDE. Contains wrappers to resources such as localized strings, cursors and images.

8) **PatternEditor.Tools**

   Contains classes that represent the tools the user is provided with.

9) **PatternEditor.Utils**

   Contains classes that are of general usability - although they were developed for this project, they may be reused in many other applications.

3.3 Utilities

Several classes that are not strictly connected to pattern design were implemented. Since they will be referred to in parts discussing the more important objects, they are introduced first. All of the objects mentioned below can be found in `PatternEditor.Utils` namespace.

3.3.1 Serialization

In early versions of the program, native .NET serialization interface was used. However, its many limitations appeared soon (difficult handling of changes in the object structure, inability to produce human readable output...). Thus new serialization mechanism was implemented that would suite the needs of the program. The mechanism is based on XML and takes advantage of good XML support provided by .NET Framework (`XmlReader` and `XmlWriter` classes).

Object graphs of arbitrary complexity are stored and restored, while their structure is left intact. For every object instance an element is written to the XML with unique id. The properties of the instance are written as child elements. The properties are output only the first time the instance is written to XML. If the instance is written for second and every subsequent time, it is only referenced by its id.

Every object that should be serializable implements the `IXmlPersistent` interface. This interface defines methods `WriteContentsToXml` and `ReadContentsFromXml`. An instance of `XmlObjectWriter` and

`XmlObjectReader` respectively is passed as a parameter to the methods. The serializable object also has to have a parameterless constructor, as will be explained below.

The serialization process starts with establishing a mapping between XML element names and classes. This mapping is represented by an instance of a descendant of abstract class `TypeToXmlElementMapping`. The program uses `AttributeTypeToXmlElementMapping`, a `DictionaryTypeToXmlElementMapping` descendant, that adds an entry to the dictionary for each class that implements `IXmlSerializable`. (Enumerating all such classes is done through the .NET reflection API.) The element name of a class is either the value of `XmlElementAttribute`, if it was applied to the class, or the name of the class.

An instance of `XmlObjectWriter` is created. Then the method `WriteObject` is called with the root object as a parameter. `XmlObjectWriter` writes a start tag for the object in the form <ElementName id='*x*'> where *x* is the unique object id. Than it calls the object's `WriteContentsToXml` method. In this method the object writes its properties to the XML. If these are `IXmlSerializable` descendants, the method `WriteObject` or `WriteElementObject` is called for each child object. `XmlObjectWriter` keeps track of objects that have already been written to the XML and their id. If an object was already output, only an empty element with the id attribute of the object is written. `XmlObjectWriter` is also responsible for closing the elements it has opened.

The deserialization is done symmetrically, but `XmlObjectReader` has a more difficult task, because it has to create new instances of objects. If `XmlObjectReader` encounters a start tag for an object that was not already read (i.e. the id is not yet mapped to an initialized object), it creates a new instance of the type associated with the element name in `TypeToXmlElementMapping`. This is achieved via the .NET reflection API and this is why every `IXmlPersistent` descendant has to have a parameterless constructor. After the new object was constructed, its `ReadContentsFromXml` method is called, so that the object may initialize.

Under certain circumstances, an object may need to have a reference to its newly created child object before the child object reads its contents. If this is the case, the `ReadObjectHeader` method is called. It returns an instance of `XmlObjectReader.ObjectHeaderResult` inner class. This class contains a reference to possibly not yet initialized object and a flag, whether it was initialized (i.e. whether this is the first occurrence of an object with this id). The parent object saves the reference to the child object and calls `ReadObjectContents` method, which initializes the object if needed.

All methods for reading objects have a `System.Type` parameter to validate the object that is read against, or are of a generic form so that type safety is enforced during the deserialization process.

3.3.2 Structured cloning

Since the editor should support undo and redo operations, there arises a need to create an in-memory copy of the graph of pattern objects. In theory this could have been done by serializing the pattern to an in-memory XML stream and deserializing it from the stream. But that would not be an elegant solution. Apart from that, creating a copy of the object graph, possibly with some modifications on certain type of objects is useful for some other operations. To achieve this, structured cloning mechanism is introduced. Note that this mechanism will bear many resemblances to the serialization mechanism.

Every object that is to be cloned has to implement the `IStructuredCloneable` interface and have a parameterless constructor (for similar reasons as the serializable objects need it). To create a copy of the graph of objects, an instance of a `CloneFactory` descendant is created. If a simple clone is needed, it would be `CopyCloneFactory`. The `GetClone<T>` generic method of `CloneFactory` is called (the genericity enforces type safety). The clone factory checks whether this object was cloned before, and if it was, the clone is returned. If it was not, a new clone is created by calling the `CreateClone` pure virtual method of `CloneFactory`. This is where the `CloneFactory` descendants come into play. As of `CopyCloneFactory`, the clone is created by calling the parameterless constructor of the appropriate type. The clone is represented by an instance of `CloneFactory.Clone<T>` descendant. Aside from the cloned instance, the `Clone` object contains a flag, whether it is initialized. `CreateClone` method may in certain implementations return already initialized clones, which is a little counter-intuitive, but useful (e.g. transformed cloning, see Section 3.5.2).

If the clone is not initialized, it is then filled with copy of the data by calling the `IStructuredCloneable.CloneMemberData` method. A reference to the `CloneFactory` is passed to the method along with the source object, which is guaranteed to be of the same type as the target object or a descendant thereof. In most implementations the class calls `base.CloneMemberData` first. If the member data are themselves `IStructuredCloneable`, the `CloneFactory.GetClone<T>` method has to be called to clone the data. Otherwise the newly created object graph would intersect with the original graph and the whole structure would soon pass to an invalid state.

If an object needs the pointer to its child object sooner than the object is initialized it may do so by calling `CloneFactory.GetNonInitializedClone<T>` and initializing the clone with `CloneFactory.InitializeIfNeeded` method.

3.3.3 Other utilities

Several other utility classes were implemented. Since their function is quite intuitive and they exhibit no surprising features, they will not be discussed here in much detail.

Because .NET Framework 2.0 does not have these collection types, `HashSet` (backed by a `System.Collections.Generic.Dictionary`) has been implemented, as well as `OrderedSet` (simple implementation, backed by `System.Collections.Generic.List`).

Surprisingly enough, the .NET Framework implementation of `System.Collections.ReadOnlyCollectionBase` is only capable to wrap up a list. So a simple wrapper class that prohibits modification to the underlying collection was implemented – `ReadOnlyCollection<T>`.

A simple graph representation was introduced with `IDirectedGraph` interface and static class `TopologicalSort` provides functionality to topologically sort such a graph.

3.4 Pattern objects

All objects that are integral part of the pattern reside in the namespace `PatternEditor.PatternObjects` along with some helper classes. For deeper background on structure of these objects see Chapter 2. An overview of the pattern objects inheritance structure may be seen in Fig. 4

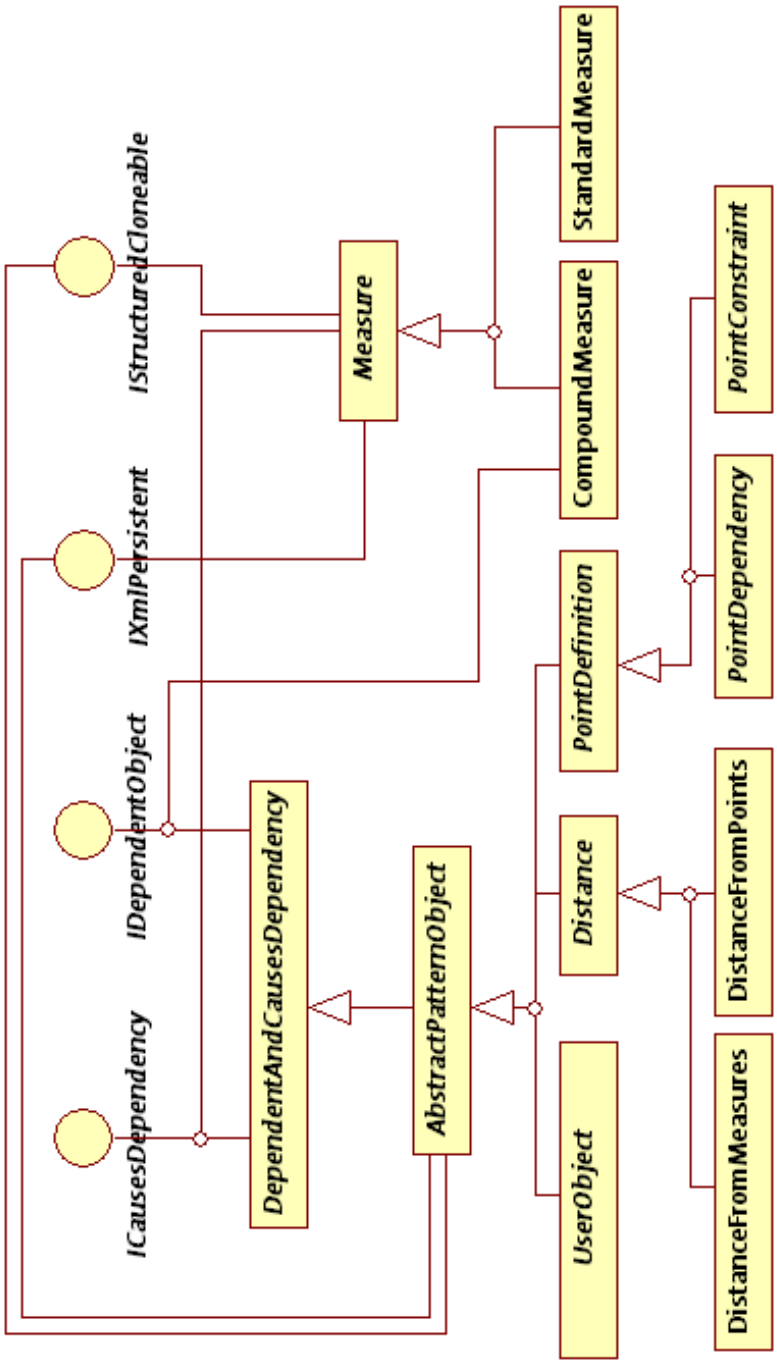


Fig. 4: Overview of pattern objects inheritance structure. Some less important descendants of AbstractPatternObject were omitted for clarity.

3.4.1 The dependency model

Since the program needs to somehow keep track of dependencies between pattern objects, two interfaces are introduced: `IDependentObject` and `ICausesDependency`. The former keeps track of its superiors and has a method `OnChanged` which is called whenever one of these objects has changed. The latter only keeps track of objects it is superior to.

To reuse code, all operations on descendants of these interfaces are managed by one singleton class – `DependencyManager`. Aside from managing the actual lists of dependent and superior objects (the `SetProperty` method), `DependencyManager` handles **change events.** A change event is realized by a call to the `OnChanged` method on every object that is dependent on the event sender. This is done by calling the `DependencyManager.RaiseEvent` method. Under certain circumstances, especially when a massive change is applied to the object structure (e.g. deserialization), the objects may be in an invalid state during the process. In this case `DependencyManager` may be **suspended**. In the suspended state, the events are not executed (the `OnChanged` method is not called) and are stored in a list instead. When the manager is resumed, all events are raised simultaneously. `DependencyManager` may be suspended multiple times, the same number of calls to `Resume` are then needed for the manager to return to the normal state and raise the waiting events.

While raising change events, `DependencyManager` needs to ensure that when object *A* updates its state, all objects that are superior to *A* are already updated. Thus it has to topologically sort the objects according to their dependency. In addition, possible cyclic dependencies are detected.

Let us note, that upon removing an object from the pattern structure, it is important to ensure not only that other objects do not depend on it, but also that the object has unregistered from its superiors. Otherwise, a reference to the object will be kept in the superior and the removed object will never be seen as garbage, resulting in a memory leak. Not to mention that the dependent object may easily be in an invalid state when its `OnChanged` method is called. To accomplish this, `DependencyManager` provides `RemoveFromDependencyGraph` method.

Two helper classes are present to simplify implementation of the `IDependent` and `ICausesDependency` interfaces. The `DependentAndCausesDependency` class has a simple implementation of both. Generic class `CausesDependency<T>` is a wrapper for object of type T. `CausesDependency<T>` implements `ICausesDependency` and thus allows any standard object (e.g. number, enum,...) to be inserted in the dependency graph.

3.4.2 Other general properties

Aside from being a part of the dependency graph, all pattern objects have to be serializable and structured cloneable to allow for save/load and undo/redo. Thus they all implement `IXmlPersistent` and `IStructuredCloneable`. To simplify this, an abstract class `AbstractPatternObject` is provided. `AbstractPatternObject` inherits from `DependentAndCausesDependency`, `IXmlPersistent` and `IStructuredCloneable`. Majority of the pattern objects is derived from `AbstractPatternObject`.

Native unit for all distances and coordinates is millimeter. The pattern coordinate space is similar to screen coordinate space - the origin of the coordinates is at top left point of the pattern. X-axis is horizontal and x coordinates grow towards right hand side, y-axis is vertical and y coordinates grow towards the bottom of the pattern.

3.4.3 User objects

Base class for all user objects (for deeper background see Section 2.2) is the `UserObject`, which introduces several basic features. First is the **parental structure**. This structure helps to keep the objects organized. A **parent** is responsible for managing its **child** objects (e.g. painting, ...). Except for the pattern as a whole, every `UserObject` has at least one parent. However an object may have more than one parent. This is necessary, since a point in the construction layer may be contained in several construction lines. The point has to know to which lines it belongs and all of the lines have to treat the point as their child. Other objects than points in the construction layer, are in an invalid state if they have more than one parent. The parental structure is also reflected by the `Children` property. Note that the value of `Children` property is not stored directly in the object, and is dynamically generated in the derived classes instead.

Then there are members related to the position of the object – the `BoundingBox` property, and the `GetObjectsAt` method. The latter returns `OrderedSet` of objects that lie at specified coordinates (these are ordered from top to the bottom of the pattern) and are of specified `UserObject.ObjectType`. `UserObject.ObjectType` enumeration introduces several categories of objects that a tool in the editor might be interested in.

Every object is responsible for adding itself to the output of the `GetObjectsAt` method, if it satisfies the conditions specified. A parent object is responsible for calling this method on all of its children and gather together the output values. The return value has to be a set, since an object may have multiple parents, and thus could be added multiple times to the resulting collection.

The `GetTransformableObjects` method returns `HashSet` of `ITransformable` descendants. The `ITransformable` interface is a base for all objects that may be transformed by an affine transformation. As of current implementation, only translation is supported. If the pattern is scalable, most of the objects are dependent on other objects and thus they are not themselves transformable. For example a pattern part returns only its anchor point as a transformable object. Another transformable object is a relatively positioned point.

The `UserObject` also contains some UI-related members. The only one of interest is the `OnPaint` method. The `OnPaint` method is provided with an `EditorPaintEventArgs` object as its only argument. `EditorPaintEventArgs` is derived from `System.Windows.Forms.PaintEventArgs` and introduces several other members that affect the drawing of user objects. Most notably it provides a transformation between the pattern coordinate space and the screen coordinate space. This cannot be achieved by applying a transformation to the `System.Drawing.Graphics` object, because some visual objects (e.g. boxes that indicate the presence of a point) should have a fixed pixel size independent of the scale the pattern is viewed in. There is a special transformation provided for these objects, so that they may be magnified if the user wants to do so - most notably when printing the pattern.

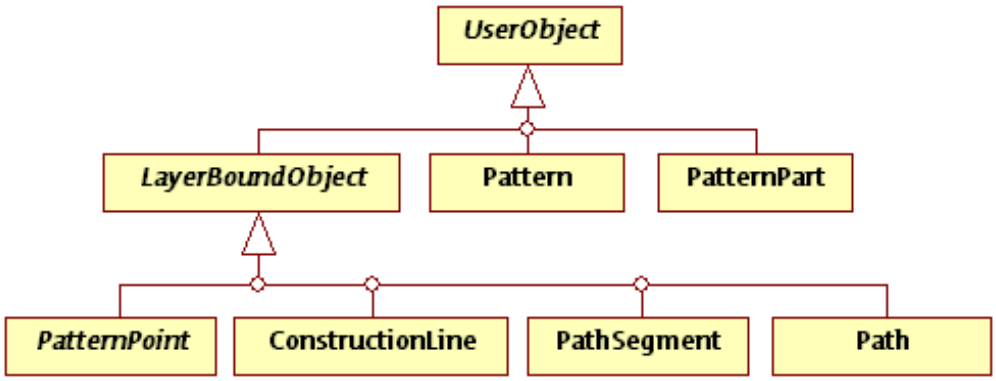


Fig. 5: Inheritance structure of user objects

3.4.4 Pattern, pattern part, layers, sublayers

`Pattern` is the topmost object in the parental structure. It manages a list of `PatternPart` objects that represent the individual parts of the pattern. It also keeps a list of `Measure` objects involved in the pattern. Some helper methods that simplify the usage of `UserObject.GetObjectsAt` are introduced.

`PatternPart` organizes its child objects into layers and sublayers. The layers are represented by a `Layer` object. Since there are always exactly three layers in a pattern, it is useful to have only three instances of `Layer` for the whole pattern (or whole program). To achieve that, a structure like Java 5's enum would be suitable, but C# does not provide such a mechanism. This behavior is simulated instead. The `Layer` class is created as a "tripleton" and provides static method to query for the individual `Layer` instances using the `Layer.Type` enumeration. `Layer` contains its localized name and colors, objects in the layer should be painted with.

`PatternPart` has an associated list of `SubLayers` which the `PatternPart` contains. `SubLayer` is simply a label applied to objects and has no other functionality. The child objects of `PatternPart` are stored in a dictionary from `Layer` to dictionary from `SubLayer` to list of objects.

Both `Pattern` and `PatternPart` are not involved in the dependency graph of the pattern. They are only containers of other objects.

3.4.5 Layer bound objects

Base class for all layer bound objects is `LayerBoundObject`, which provides information about `Layer`, `Sublayer` and `PatternPart`, the object is contained in. All children of a `LayerBoundObject` have to lie in the same `Sublayer` as the parent object. Most important of classes derived from `LayerBoundObject` is `PatternPoint`, to which the whole next section is dedicated. The others will be discussed here.

The `ConstructionLine` class represents a construction line. It is defined by two points. Aside from these two points, `ConstructionLine` also manages a list of points that will always (regardless of scale) lie on the `ConstructionLine` (the `PointsOnLine` property and `AddPoint` and `RemovePoint` methods). This connection between `ConstructionLine` and a point is important for the editor tools. However, this list is not computed automatically (it cannot be, unless the program would be capable of solving equations describing geometrical relationship between objects, which may be quite complicated). The editor tools are responsible for adding only such points that semantically have to lie on the line (e.g. intersection of the line with another object). The user may add a point directly to this list as well. The program assumes that if the user decides to do so, he is certain, that the point will always lie on the `ConstructionLine`.

The `Path` object contains a list of `PathSegments`, that form the path. As was discussed in Section 2.4, `Path` contains the first point of the path and each segment contains its endpoint. `PathSegment` is dependent both on its end point and its start point but it cannot intercept changes of its start point, because start point is not a property owned by `PathSegment`. Therefore, the path segment exposes the `RefreshStartPointDependency` method that is called by the preceding `PathSegment` or the `Path`, whenever the start point of the segment was changed (i.e. it was replaced with another instance,

not that it has raised a change event – change events are handled properly by the dependency mechanism). `Path` is not directly involved in the dependency graph, it is only a container for other objects.

`PathSegment` represents a Bézier curve. To save work on algorithms for the curve, the `System.Drawing.Drawing2D` library is called for most calculations. This is not limited to drawing, but it applies for example to checking whether a point is closer than specified radius to the path segment: A `System.Drawing.Drawing2D.GraphicsPath` that represents the segment is created. It is then widened by a call to `GraphicsPath.Widen` method to the specified radius and then `GraphicsPath.IsVisible` method is called to check whether the point lies in the widened region.

The user does not distinguish between the first and the last point of `Path`, so `Path` exposes `Reverse` method and thus may be transformed seamlessly to the same path with the first and end point swapped.

3.4.6 Points

Points are the most important objects in the pattern. All objects that represent points are derived from the `PatternPoint` abstract class. This has two descendants, `PointProxy`, which is discussed in Section 3.4.8, and the `VisualisedPoint` abstract class. While `PointProxy` is only a wrapper class, `VisualisedPoint` is a base class for all actual points. As its name suggests, `VisualisedPoint` implements only the `OnPaint` method. `RelativelyPositionedPoint`, `DependentPoint` and `ConstrainedPoint` are counterparts of eponymous objects discussed in Chapter 2. Since `DependentPoint` and `ConstrainedPoint` both delegate some of their functionality on an instance of `PointDefinition`, they have a common base class, `DefinedPoint`. A diagram of the inheritance hiearchy of the point objects can be seen in Fig. 6.

`SimplePoint` is a point, that is independent on any other objects. As of current version, only its descendant `PatternPartAnchorPoint` is actually utilized in the pattern structure. `PatternPartAnchorPoint` represents an anchor point of a part of the pattern, and therefore introduces some specific functionality related to this role. Even in case that the user wants to drop all dependencies in the pattern, the points are not converted to `SimplePoint`, but to `RelativelyPositionedPoint` – relative to the anchor point of the part. This approach provides better behavior for moving the pattern part. `PatternPartAnchorPoint` is also an exception from the general rules of the pattern, because it does not belong to any layer or sublayer.

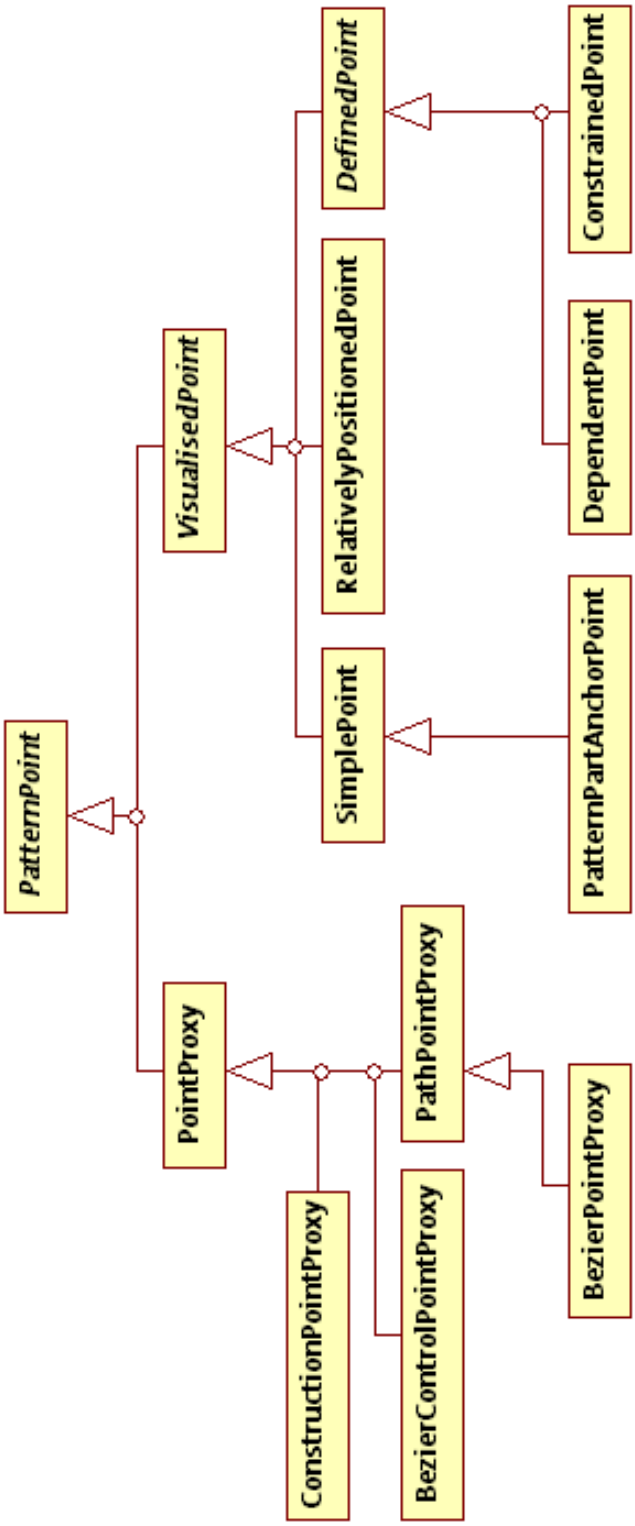


Fig. 6: Pattern points inheritance structure

3.4.7 Defined points, point definition

`PointDependency` and `PointConstraint` have two common properties, that are delegated to the `PointDefinition` class. Both calculate their positions from other objects and both are directly bound to a point that owns them. Thus a common ancestor was declared for the respective point types as well.

Furthermore, `PointDefinition` may be affected by transformations and thus it should be notified of them. `PointDefinition` also has an `OnPaint` method, which allows `PointDefinition` to graphically indicate some property of the point (e.g. `LineAndCirclePointDependency` draws a short arc indicating the compass trace to distinguish such a point from a point defined by `AngleAndDistancePointDependency`).

`PointDefinition` is along with `CurvesIntersection` the only object in the pattern structure, that caches its state/position. This is however sufficient, because `DefinedPoint` takes its position directly from `PointDefinition`, and `ConstructionLine` and `PathSegment` actually don't have a notion of state. Moreover there are usually not long chains of `RelativelyPositionedPoint` instances and the only other object dependent on points, `DistanceFromPoints`, is not present often in the pattern structure. Thus the caching in `PointDefinition` is enough to prevent exponential complexity threat, that would arise if no object cached its position and calculated it from the position of its superiors every time the position would be needed.

A common ancestor for all point dependencies and constraints respectively is `PointDependency` and `PointConstraint`. The only type of `PointConstraint` currently implemented is `LinePointConstraint`, which was discussed in Section 2.5. The construction point dependencies were discussed in Section 2.3.2. `IdenticalPointDependency` and `TransformedPointDependency` were discussed in the analysis as well (Sections 2.5 and 2.7 respectively). Since their implementation does not bring any non-obvious ideas, they will be left aside. For details on the actual realization, see the reference documentation. For a diagram of the inheritance structure of point definitions, see Fig. 7.

Several other classes derived from `PointDependency` were introduced that were not explicitly mentioned in the analysis. A list of these with a brief summary of their function follows:

1) `AnchorPointDependency`

   A special type of dependency that binds together `PatternPartAnchorPoint`, which does not belong to any layer, with an initial point in the construction layer.

2) `PointPointDependency`

   A common ancestor for all point dependencies that somehow transform one point to another point, without being dependent on other user objects.

3) `IdenticalRelativeToAnchorPointDependency`

   A point dependency used when a pattern part is split in two. It takes a point in different part of the pattern as input and transforms it to coordinate space of the part that contains the point it defines. In other words, an image of set of points with this dependency will mirror the set in the original pattern part, but will move along with the part it belongs to.

4) `CurvesIntersectionDependentPointDependency`, `CurvesIntersectionPointDependency`, `BezierCurveSplitControlPointPointDependency`

   Together with `CurvesIntersection` provide a mechanism for representing points created by cutting a path segment by a construction line or another path segment. Used when splitting the part into fragments with `PartCutter`, see Section 3.5 for further details.

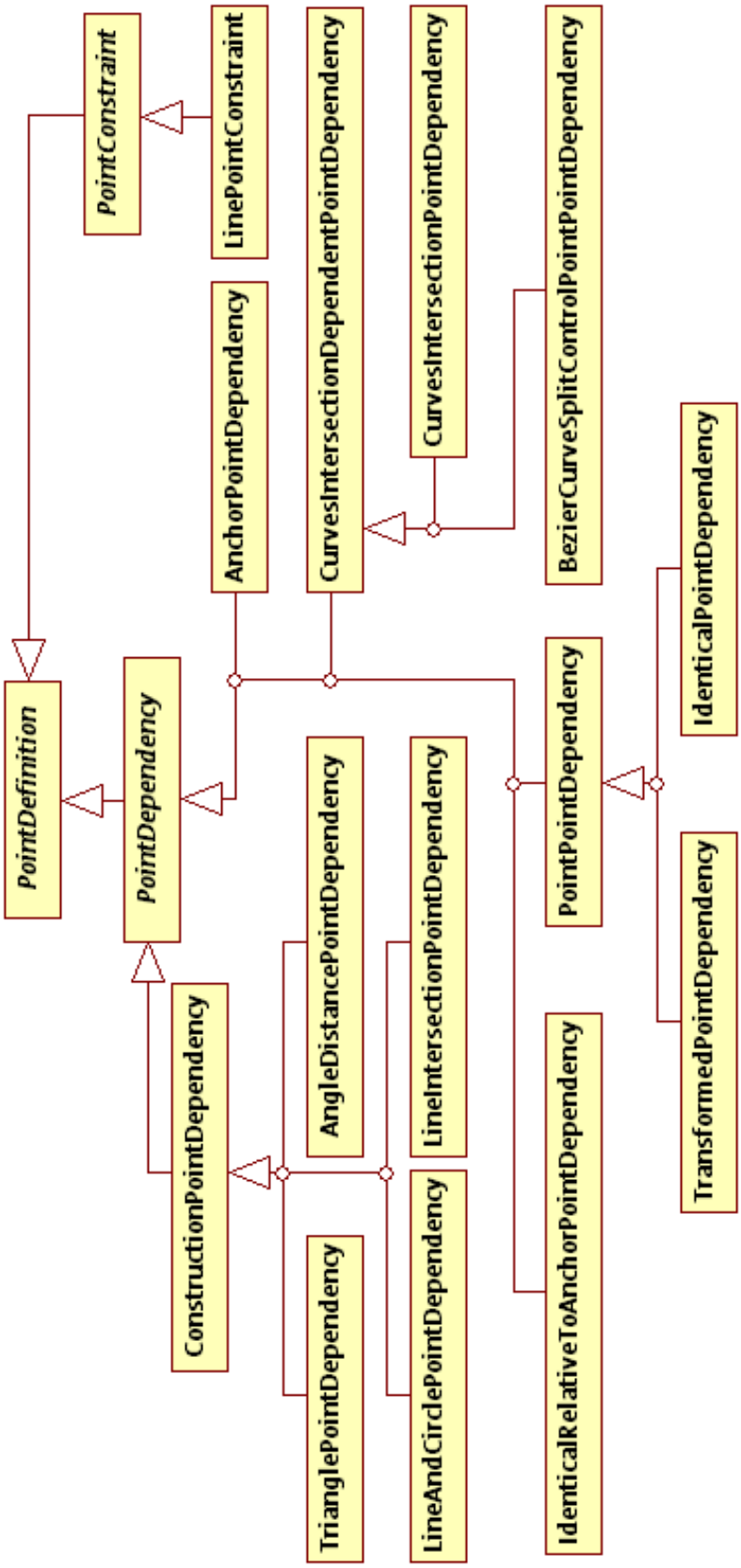


Fig. 7: Point definitions inheritance structure

3.4.8 Point proxy

Consider a situation, that the user wants to change the type of point (e.g. change `DependentPoint` to `RelativelyPositionedPoint`). All objects that are dependent on the original point should update their respective properties. There arise two possibilities how to handle such a situation.

First, a new type of event could be created that would be raised when an object should be replaced with another instance. This would require every object to implement a specific handler for this event, that would check all applicable properties of the object and would update them if necessary. Using advanced features of the .NET reflection API, it might even be feasible to implement a generic handler for this event that could update any type object by enumerating all of its properties and checking whether these are of appropriate type.

Second, an intermediate wrapper object could be created. Let us call such an object **point proxy.** Every object dependent on a point would depend on a point proxy instead and the point proxy would be the only object holding direct reference to the actual point (the **referenced point**). Then the type of point could be changed by updating only one object, the proxy.

The former solution would require either a lot of additional coding or a very unclear implementation via the reflection API. Although the latter solution also has its drawbacks, it allows for an elegant implementation of Bézier points and thus was chosen.

The proxy is represented by a `PointProxy` class. In early versions of the program, `PointProxy` delegated nearly all of its methods and properties (except for serialization and cloning) directly to the referenced point. `PointProxy` was transparent for the parental and dependency structure. However, this has exhibited a lot of problems, and in current implementation, `PointProxy` is a standalone object, that is equal member of both dependency and parental structures. No point can be referenced by more than one instance of `PointProxy`.

`PointProxy` has several descendants. `ConstructionPointProxy` brings no new functionality and exists simply to distinguish the points in the construction layer. All objects that refer to points in construction layer have `ConstructionPointProxy` properties. `PathPointProxy` represents a point in the core of a path and exhibits the benefits of the proxy approach: The proxy makes it possible to link additional data or functionality (the reference to owning `Path` or `PathSegment` in this case) to any type of point.

`PathPointProxy` object itself is deprecated in the current version of the software and only its descendant `BezierPointProxy` (which represents a Bézier point) is present in the pattern structure. `PathPointProxy` was not merged with `BezierPointProxy` to keep compatibility with pattern files

saved in older versions of the program – a `PathPointProxy` instance is read by normal deserialization process and converted to `BezierPointProxy` if needed.

`BezierPointProxy` further extends `PathPointProxy` by adding references to the control points of the Bézier point and methods to handle them. These are of the `BezierControlPointProxy` class, another `PointProxy` descendant. The `BezierControlPointProxy` class provides the underlying point with different visual appearance and exposes methods specific to handling control points of a Bézier point. `Path` and `PathSegment` reference only `BezierPointProxy` instances.

Although linking additional functionality to points is very useful, the proxy approach brings some disadvantages. While some actions of the user affect the proxy (e.g. changing the type of the referenced point) some affect the referenced point (e.g. modifying the defining dependency of dependent point). And it is possible to think of a scenario where a proxy references another proxy (e.g. a point wrapped in `PointProxy` is to be added to `Path`. The proxy might be referenced from outside so it must be kept and thus it has to be wrapped again in a new `BezierPointProxy` instance)

So a mechanism that would allow for retrieving the right type of point is needed and thus `UserObject.Target` method is introduced. This method returns an object of a specified category. The `UserObject.TargetType` enumeration represents these categories. As of current version, all objects return themselves regardless of `TargetType` specified, only `PointProxy` and its descendants provide different functionality for retrieving either the underlying point (`TargetType.Default`), the proxy itself (`TargetType.Proxy`), the proxy that directly references the underlying point (`TargetType.LowestProxy`) or the topmost `BezierPointProxy` (`TargetType.BezierPointProxy`).

3.4.9 Semantical equivalence, point chain

Another difficulty arises from the chosen model: What the user perceives as a single point may in fact be represented by several distinct objects in the program. The program should, however, handle these points as identical. This is overcome by introducing the notion of semantical equivalence. The idea is simple: two points are semantically equivalent if they lie in the same location regardless of scale. Since the program cannot decide this in most cases, the program's definition would be a little narrower:

Points *A* and B are **semantically equivalent** if either:

1) *A* is `PointProxy` and *B* its referenced point or vice versa.
2) *A* is `DependentPoint` and *A*'s defining dependency is `IdenticalPointDependency` on *B* or vice versa.
3) There exists a path between *A* and *B* consisting of steps 1 and 2.

The **semantical equivalence graph** of a pattern is a directed graph, whose vertices are all points contained in the pattern. For points *A* and *B*, the directed edge *(A,B)* is present in the graph if *A* is `PointProxy` and *B* its referenced point, or *A* is `DependentPoint` and *A*'s defining dependency is `IdenticalPointDependency` on *B*.

Note that if the pattern does not contain a cyclic dependency, the semantical equivalence graph is a forest and every tree in it is rooted. This leads to a definition of point chain:

A **point chain** of point *P* is a directed path in the semantical equivalence graph starting in *P* and ending in the root of the tree that contains *P*.

In other words, point chain of point *P* starts with *P* and continues by the point chain of its referenced point, if *P* is `PointProxy`, or by the point chain of a point *P* identically depends on, if there is such a point. This is the algorithm implemented by `PatternPoint.GetPointChain` method. It can be easily shown, that two points are semantically equivalent if and only if their point chains intersect, which is the algorithm presented by `PatternPoint.SemanticallyEquivalent` static method.

3.5 Pattern adjustments

As discussed in Section 2.6, the pattern adjustments consist of splitting the part into two fragments and then transforming one of the fragments.

3.5.1 Splitting parts

The classes derived from `PatternEditor.Tools.PartCutter` abstract class are responsible for splitting the part. `PartCutter` enumerates all `ConstructionLine` and `Path` instances in the active sublayer of the `PatternPart`. It delegates the actual splitting of individual `ConstructionLine` and `PathSegment` instances to its descendants. The `PartCutter` then connects the split `PathSegments` into final `Path`. The `PartCutter` does provide several protected methods that ease the implementation of its descendants though.

As of current implementation, it is assumed, that a `Path` is split in no more than two pieces, i.e. all split `PathSegment` instances of the same `Path` that belong to the same fragment are connected during the cutting process. The original objects are not modified, the newly created objects are inserted into new `SubLayer`.

To provide sublayer-consistency when a curved `PathSegment` is split, points that represent the location of the split and associated control points must be created in the construction layer of the new sublayer. The intersection, as calculated by `Geometry.CubicBezier.CurveIntersections`, is wrapped in a `CurvesIntersection` descendant. This objects represents both the location of the split and the locations of associated control points. Then the actual points are created. These are defined by

`CurvesIntersectionDependentPointDependency` descendants. The algorithms that calculate the actual intersection location are discussed in Section 3.6.

3.5.2 Preserving structure, transformations

The points that define the split lines and path segments are transformed images of points in the sublayer below (although the transformation is identity in many cases). Let us have a point *A* that is identically dependent on a point *B* in the original sublayer (e.g. point in the core of a `Path` and an underlying point in the construction layer). Then *A'* denotes the transformed image of point *A* in the new sublayer and *B'* denotes the image of point *B*. The straightforward approach would be to simply put *A'* dependent on *A*. However, to preserve semantical equivalence, *A'* should be identically dependent on *B'* (see Fig. 8). The structure of `PointProxy` instances should be preserved as well.

To achieve that, the structured cloning mechanism is extended to create copies of points after a given transformation, while preserving this inner structure. This mechanism is provided by `PatternEditor.PatternObjects.TransformedCopyCloneFactory`. A `PointTransformer` instance is associated with `TransformedCopyCloneFactory`. `PointTransformer` performs the acutal transformation on given points, in this case `IdenticalPointTransformer` would be used.

In contrast to `CopyCloneFactory`, `TransformedCopyCloneFactory` does not create new instances of objects that are not layer bound, (e.g. transformed points still indirectly depend on the same set of measures) and it does apply special processing to points. When a point, that was not cloned already is encountered, its `PatternPoint.TransformedCopy` method is called and the result is stored as the cloned image of the point. By default, this method creates a new point with a call to `PointTransformer.TransformPoint`. If a point was created this way, `PointTransformer` is responsible for copying all relevant data (e.g. name) from the original point to the clone. The clone is then returned as initialized and the `CloneMemberData` method is not called on it. `PointProxy` and `DependentPoint` override this behavior to provide more structured results of cloning as described above. To maintain sublayer-consistency, `TransformedCopyCloneFactory` does translate the sublayer of original objects to the new sublayer as well.

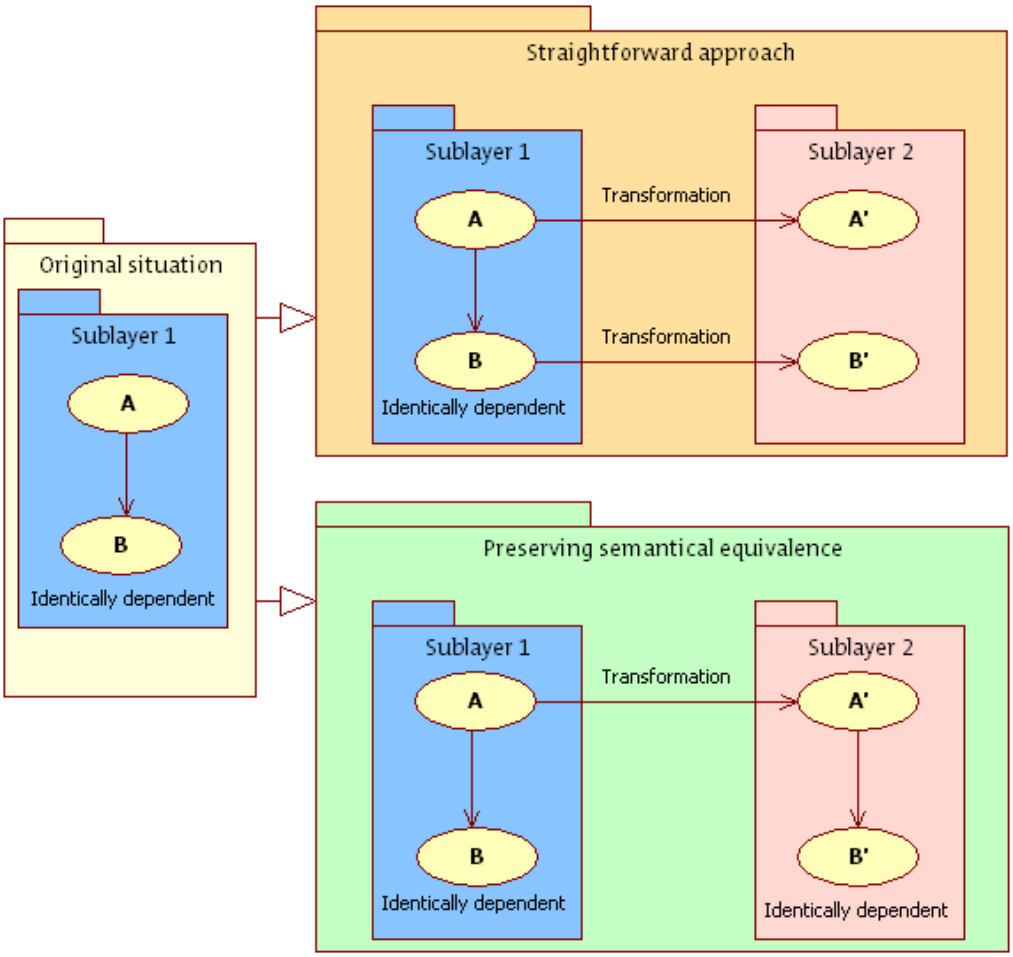


Fig. 8: Comparing straightforward approach to transformations of points for pattern adjustment to approach preserving semantical equivalence.

Although this extension to `CloneFactory` is useful, it requires that type safety is sacrificed at several places in the code. The main problem is that a type parameter of a generic class cannot be constrained at runtime. To be more specific:

In the `TransformedCopyCloneFactory.CreateClone<T1>` method, a check is performed, whether the source object is a descendant of `PatternPoint`, and if it is, `PatternPoint.TransformedCopy<T2>` is called. The `T2` type parameter cannot be replaced with a type constant – it is not known whether `T1` is not a more specific class than `PatternPoint`. So `T2` must be `T1`. Thus the type parameter of `PatternPoint.TransformedCopy<T>` cannot be constrained to be a `PatternPoint` descendant in the method declaration, otherwise the program would not compile.

Moreover the compiler does not allow conversions between type parameters and statically defined types. However, such conversions are needed because of the missing type constraints. So they must be performed by converting the

object to the `Object` class first and then converting it to the desired type. The resulting code of `TransformedCopyCloneFactory.CreateClone<T>` and `PatternPoint.TransformedCopy<T>` looks very inelegant.

To perform the actual adjustment, one of the fragments is transformed by the `TransformedCopyCloneFactory` mechanism, only different `PointTransformer` descendant, such as `TransformPointTransformer` is used.

3.6 Graphic and geometric algorithms

Several graphic algorithms had to be implemented for the program. All of these are located in the `PatternEditor.Geometry` namespace. The `Plane` static class provides methods to analytically solve many simple geometrical problems in two dimensional plane (e.g. two circles intersections, line intersection,...). These are of no big interest, for detailed description see the reference documentation.

The most complicated geometrical object involved in the pattern is the Bézier curve. As was already noted in Section 3.4.5, the .NET graphics library is used as much as possible to simplify the implementation of Bézier curves. However, the pattern adjustment process involves computing the intersections of a Bézier curve with a line or another Bézier curve. The latter results in set of two 3rd degree equations for two variables and thus cannot be solved analytically, an approximation is calculated instead. Although the former can be solved analytically, regarding it as a special case of the latter situation saves implementation time and avoids the situation when the respective 3rd degree polynomial is ill-conditioned.

Note that Bézier curves present in the pattern outline are quite “neat” – they contain no self-intersections and are usually not curved “much”. Moreover, in the situations, when intersections are calculated, the curves involved usually have only one intersection and they have very different “direction”. Furthermore it the deviations are less than a millimeter, then the result is considered more then sufficiently precise. Thus an approximation of the curve with a polyline provides sufficiently good results.

To approximate the curve with a polyline, the curve is recursively split using the de Casteljau's algorithm [13] until the parts are small enough. The endpoints of the curve parts are then used as the vertices of the polyline. The exact condition for “small enough” is that the summed distance from the start point to end point through control point 1 and control point 2 is smaller than a given value. This assures that the line is approximated with shorter line segments in the regions where it is more curved. Within the application, the `BezierCurveApproximationMaxLineLength` application setting is used. By default, it is set to 10 millimeters, which has shown to be enough to approximate the intersection with 1 millimeter precision even in cases that are extreme compared to standard pattern outline.

When the intersections of a polyline with a line are computed, the intersection of the line with all segments that form the polyline is calculated. If two polylines intersections should be calculated, a brute force approach (i.e. calculating the intersection of every possible pair of segments) would lead to quadratic complexity and because the polyline may consist of a large number of segments, this is not bearable. A planar partitioning algorithm is implemented instead.

The algorithm makes use of the fact, that both polylines were created as approximations of Bézier curves and thus the maximum length of a segment is limited. The bounding box of the two curves is partitioned into squares the size of a maximum segment length. Now every segment belongs to at most four partition squares. Only the segments that belong to the same partition square are checked for intersections.

Since it is needed to split the Bézier curve at the point of intersection without changing the shape of the path, the control points associated with the new Bézier point must be retrieved. These can be calculated by the de Casteljau's algorithm. So the parameter value that yields the location of intersection in the de Casteljau's algorithm must be found.

During the process of converting the curve to a polyline, the values of the parameter are associated with the endpoints of the segments. The initial value of the parameter is then approximated linearly from the parameter value for endpoints on the appropriate segment of the polyline. This initial value is then refined by finding the closest point to the approximate intersection on the curve using Newton's method [14]. The first and second derivative are calculated using the de Casteljau's algorithm, since a derivative of Bézier curve can be expressed as a Bézier curve of lower order [15].

Although this algorithm provides good results for the purposes of the program, it may in general yield wrong results and is not guaranteed even to find the correct number of intersections as seen on Fig. 9.

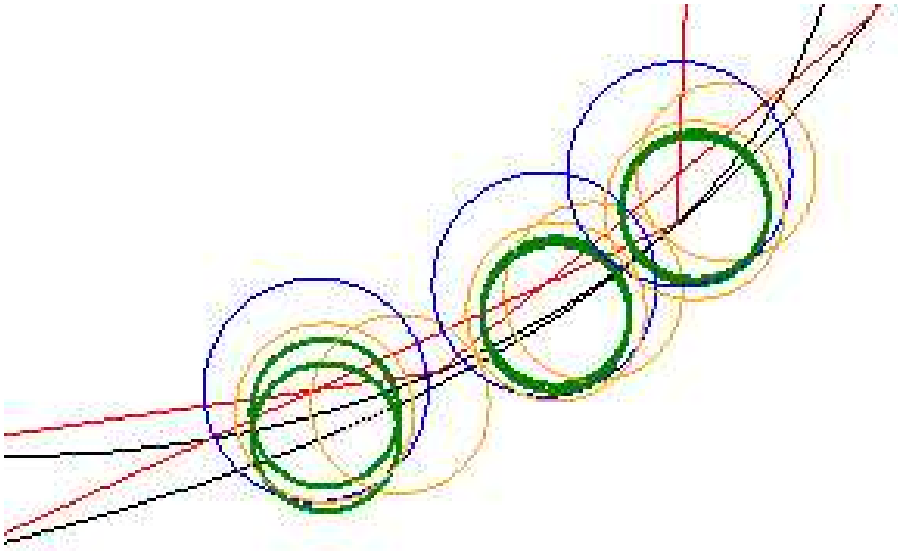


Fig. 9: Approximate algorithm for the intersections of two Bézier curves returns three intersections instead of one. Black lines are the actual curves, red lines are the approximations. Blue circles indicate the intersections of polylines, orange circles indicate locations on the original curves corresponding to initial parameter value, green circles represent the refined solutions.

Limit on the line length was 40mm, the radius of the green circle is 8mm.

3.7 User interface

The application is written using the MDI (multiple documents interface), so that it could be easily extended to multiple document editor (i.e. multiple patterns open at the same time). However, as of current version, multiple open documents are not supported.

The main editor window is represented by the `Editor` singleton class, which is also the MDI container for the whole application. Child windows for the individual open documents are represented by the `EditorWindow` class. The `EditorPanel` class is the control, that actually displays the pattern and owns the respective `Pattern` object. The event handlers for main menu events (e.g. open file, new file) are present in the `Editor` class. All of the classes mentioned are located in the `PatternEditor` namespace.

3.7.1 Controls and forms

Controls and forms the program uses are kept in `PatternEditor.Controls` and `PatternEditor.Forms` namespaces respectively. Most of the forms inherit from the `ToolForm` or `ToolFormWithButtons` classes, but this is only for copying visual

appearance of these. The form's functionality (which is usually quite simple) is not separated from the form, most of the processing happens directly in its event handlers. In a typical situation `Tool` or `Action` (see Sections 3.7.3 and 3.7.4) gathers data from the mouse events, then feeds the data to the form and shows the form. The form is then responsible for carrying out the desired operation, if the user presses the OK button.

The controls are written such that they may be reused as much as possible. Most of the controls are **editors**. An editor is a control, that provides the user with means to edit one type of objects in the pattern, this is the **edited type** of the editor. The editor controls bear names that indicate which type of object they edit (e.g. `LineAndCirclePanel` is an editor for `LineAndCirclePointDependency`). Every editor has a public property to set and get the object that corresponds to the state of the editor. The original object is not kept within the editor, all of its state is stored in the state of editor's child controls.

There are several aggregate editors (e.g. `DependencyPanel` or `DistancePanel`) which manage editors for all classes derived from their edited type. Only one of such child editors is active at a time and indicates the actual type of object that will be returned upon querying for the edited object. Since the point dependency is quite a large family of classes, child editors of `DependencyPanel` are not handled statically, but instead a list of types and their associated editors is introduced. Every time a new type of point dependency is added to the program, it has to be added to the list in `EditDependencyPanel.PopulateContexts`, otherwise the program would present the user with an error, whenever he will try edit a `DependentPoint` with the `PointDependency` not present in the list.

3.7.2 Selection, Highlights

During the editing of a pattern, `UserObject` may be selected and/or highlighted. The `Editor` class manages two corresponding sets of objects: `Editor.Selection` and `Editor.Highlights`. Objects are selected and deselected using the `Tools.MoveAndSelect` family of tools. If the object is selected, it indicates that the user specifically wants to perform some action on the selected object. A graphical emphasis is applied to objects that are selected. The object appears selected if one of its predecessors in the parental structure is selected. Thus an object may appear selected to the user even though it is not contained in the `Editor.Selection`. This does not involve any confusion to the user, because actions that are carried out on the selection usually affect child objects of the target object as well (e.g. movement, deletion).

Objects are highlighted to indicate that a certain action can be performed on them – most of the editor tools highlight possible target objects under the mouse cursor. Highlighting is also more flexible, since the color, with which the objects should be highlighted with, may be specified. Highlights may as

well be time-limited. In general, selected status is regarded as being more fundamental than the highlighted status. Highlighted status brings stronger but more short term emphasis.

3.7.3 Editor tools

Most of the functionality of the editor is managed by tools. All tools and their helper classes are located in the `PatternEditor.Tools` namespace, base class for the tools is the `Tool` class. When the users selects a tool, the `Tool.Register` method is called, with the active `EditorPanel` as a parameter. The tool then subscribes itself to mouse events of `EditorPanel`. All of the work of the tool is done in mouse event handlers. When another tool is selected, the `Unregister` method of the previous one is called to ensure the tool no longer receives the events.

A typical tool is a finite automaton with up to four states. Except of the tools that perform pattern adjustments, the specific function of tools is simple addition/modification of objects in the pattern and will not be discussed further in this documentation. The mechanisms for pattern adjustments are discussed in Section 3.5. For in depth insight on how the tools work and how they are implemented, refer to the user documentation or the reference documentation respectively.

3.7.4 Actions

Actions are operations on the pattern, that are not carried out by tools. In typical situation, actions appear in the context menu. Classes that represent actions are located in the `PatternEditor.Actions` namespace. The base class for all actions is the `Action` abstract class. When the user opens the context menu, the targets of the possible actions are determined. If the `Selection` is not empty, it is the contents of the `Selection`, otherwise it is the contents of `Highlights`. Then, for each action registered in the editor (i.e. for each action in `Editor.ActionList`), the `Action.IsApplicable` method is called to check, whether this action could be applied to current set of targets. The context menu is then populated with the applicable actions. If an action is selected, its `Execute` method is called.

Actions provide very simple processing or they only show an appropriate form and thus they will not be discussed further in this documentation. For detailed description of the function or implementation of individual actions, refer to the user documentation or the reference documentation.

The `ActionWrapper` class allows any `Action` to be inserted into arbitrary menu.

3.7.5 Undo/redo

The undo/redo functions of the editor are located in the `PatternEditor.History` class. For every step, `History` saves a copy of the entire pattern obtained from `CopyCloneFactory`. Although this approach is not exactly memory efficient, the usage of memory for undo/redo is still bearable, compared to the memory consumed by .NET Framework and UI.

The application consumes about 30MB of memory right after it has started and loaded a pattern. (about 7MB of memory is allocated when the "Open file..." dialog is opened for the first time) Every undo step in a typical pattern allocates about 50-150Kb of memory. Although infinitely many undo steps are possible, pattern creation in practice consists of at most hundred steps and only a few steps that create sublayers (effectively multiplying the size of pattern in memory). Thus the total memory allocated for undo/redo is in practice less than the memory for the program itself. However, there is room for improvement in future versions.

3.7.6 Printing

The printing of a pattern is realized via the native .NET Framework printing support. The process is managed by `PatternEditor.PatternPrinter`. The printing is done in a simple way: the whole area of bounding box of all pattern parts is printed. On every printed page, the boundaries of printing area are marked, and the page index is printed. The page index is of a form "Page NX" where N is a letter indicating the horizontal coordinate of the page and X is a number indicating the vertical coordinate of the page. Thus the pages can be easily sticked together to form the entire pattern.

3.8 Resources

The program was written to be fully localizable. Thus every string except of internal error messages* is stored in a resource. The localization of resources is then managed by the .NET Framework and is transparent for the application. Except of resources associated with specific forms, the resources are stored in `PatternEdtior.Resources` namespace. Brief description on the contents of the files in this namespace follows:

- **ActionNames.resx:** Names of editor actions that are presented to the user. The names of the entries in the resource correspond to class names.
- **Common.resx:** Miscellaneous strings. Names of measuring units are also present.

* Internal errors are errors that are of a debugging nature and should not occur in production environment, and even if they were localized, the error message would not tell anything comprehensible to the user.

- **CreateConstructionLine.resx:** Messages associated with the `PatternEditor.Forms.CreateConstructionLineForm` (deprecated, not accessible in current version, although may be reused).
- **DependencyNames.resx:** Names of dependencies as they are presented to the user. The names of the entries in the resource correspond to class names.
- **Editor.resx:** Messages and texts associated with the `Editor` object in general (such as default names of newly created objects).
- **ErrorMessages.resx:** Various error messages.
- **SelectObjectPanel.resx:** Messages associated with the `PatternEditor.Controls.SelectObjectPanel.`
- **ToolCursors.resx:** Mouse cursors for editor tools.
- **ToolIcons.resx:** Icons for editor tools. The names of the entries in the resource correspond to class names.
- **ToolMessages.resx:** Messages presented to the user by various editor tools.
- **ToolNames.resx:** Names of editor tools as they are presented to the user. The names of the entries in the resource correspond to class names.

4 Conclusion

In this chapter, the main achievements of this work as well as suggestions for further improvements will be discussed.

4.1 Achievements

The core functionality of the editor and pattern objects was successfully implemented. The UI was designed to be intuitive. When test users (students of apparel design) were presented with the program, they could quickly draft slopers, even though most of the users consider their skills with computer as basic or lower intermediate. Several of the test users claimed, that sloper creation with the program is faster than creation by hand, while the sloper created in the program can automatically adjust to any measures and thus is unlimitedly reusable. The test users were not yet presented with pattern adjustments features due to lack of time of the developer.

Printing support is simple, but considered sufficient for the current use cases of the program. The program is quite stable and supports simple auto-saving.

While writing the code of the program, focus was kept on formal cleanliness (except of the problem discussed in Section 3.5.2) and well-formed structure. Further modifications and/or enhancements were held in mind throughout the whole development process. Several reusable components for various purposes have been developed. The program is fully localizable.

4.2 Directions for future development

The problem of scaling the pattern according to given measures has posed unexpected obstacles in combination with pattern adjustments and large amount of helper structures had to be written. Thus not all basic adjustments that were planned were implemented (gathering and pleating). However, solid grounds for implementation of these were created. These adjustments should be implemented in short term. Less standard adjustments should be added as well (e.g. cutting along curves, dart transfer to a curve...)

Automatic computation of seam allowance should be added, along with more complex marks in the metadata layer.

The program should provide more extensive work with measures – saving measure values for specific customers, custom measure tables and loading measure values for industry standard body sizes.

The ultimate goal of the development process is to set up an Internet community portal dedicated to open source sewing patterns, created in the program.

5 References

[1] Technická univerzita v Liberci, Textilní fakulta: Střihové konstrukce, http://www.kod.tul.cz/info_predmety/Kmd/doc/prednasky_navrhari/9_Historie-%20Konst_pojmy_p%C5%99%C3%ADdavky.pdf, retrieved on: 2009-05-24

[2] Kalábová M., Pallayová H., Zdeňková H. (1993): Odívání III. Fortuna, Prague

[3] Gerber technologies, http://www.gerbertechnology.com/default.asp?contentID=56 , retrieved on: 2009-05-24

[4] Lectra - pattern making, http://www.lectra.com/en/fashion_apparel/solutions/pattern_design.html , retrieved on: 2009-05-24

[5] Optitex, http://www.optitex.com/ , retrieved on: 2009-05-24

[6] Audaces Apparel, http://www.audaces.com/index.php?pgn=15&lang=2 , retrieved on: 2009-05-24

[7] Wild Ginger Cameo 4, http://www.wildginger.com/products/cameo.htm , retrieved on: 2009-05-24

[8] Weisstein, Eric W.: Bézier Curve, http://mathworld.wolfram.com/BezierCurve.html, retrieved on: 2009-05-11

[9] Wikipedia - Compass and straightedge constructions, http://en.wikipedia.org/wiki/Ruler_and_compass, retrieved on: 2009-05-20 10:16

[10] Microsoft Corporation: .NET Framework 2.0, http://msdn.microsoft.com/cs-cz/library/bb966997(en-us).aspx, retrieved on: 2009-03-18

[11] Microsoft Corporation: Visual C# 2008 Express edition, http://www.microsoft.com/express/vcsharp/, retrieved on: 2009-01-15

[12] Hall M.: XPTable, http://www.codeproject.com/KB/list/XPTable.aspx, retrieved on: 2009-05-22

[13] Wikipedia - de Casteljau's algorithm, http://en.wikipedia.org/wiki/De_Casteljau%27s_algorithm, retrieved on: 2009-02-23

[14] Felcman J.: Numerická matematika, http://www.karlin.mff.cuni.cz/~felcman/nm.pdf, retrieved on: 2009-04-21

[15] Shene, C.-K.: Derivatives of a Bézier curve, http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-der.html, retrieved on: 2009-04-16

[16] Kalábová M., Zdeňková H., Pallayová H. (1993): Odívání I. Fortuna, Praha

[17] Technická univerzita v Liberci, Textilní fakulta: Konstrukce oděvů, http://www.kod.tul.cz/info_predmety/Kso/kso.htm, retrieved on: 2009-05-20 12:03

[18] Technická univerzita v Liberci, Textilní fakulta: Konstrukce základních typů oděvů, http://www.kod.tul.cz/ucebni_materialy/konstrukce/konstrukce.html, retrieved on: 2009-05-20 15:01
[19] Kalábová M., Zdeňková H., Pallayová H. (1993): Odívání II. Fortuna, Prague
[20] Wikipedia, http://en.wikipedia.org/ , retrieved on: 2009-05-24

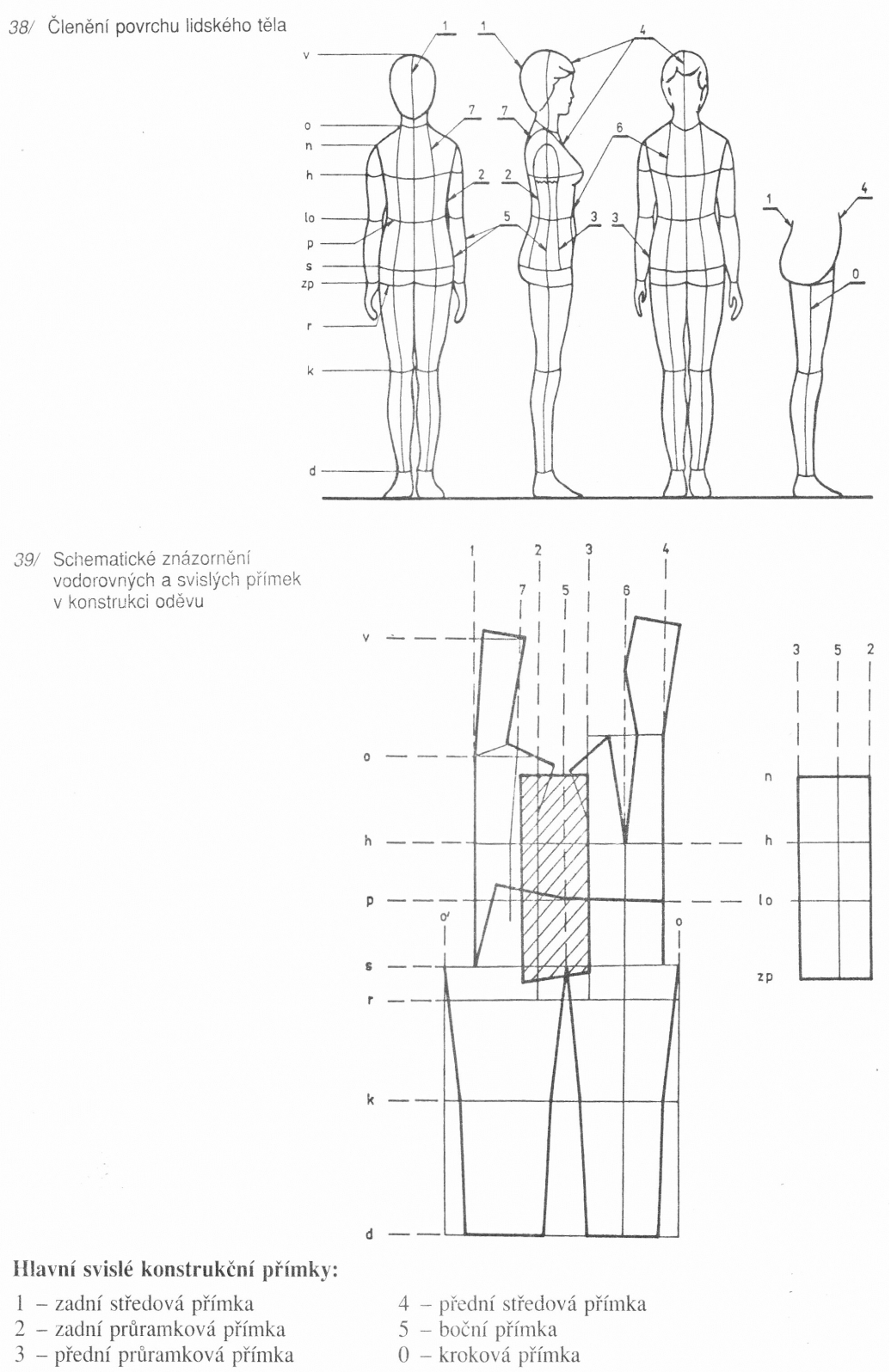
6 Appendixes

6.1 Appendix A - CD Content

The attached CD contains following files and folders:

- `PatternEditorInstall.exe` - an installer program containing binary distribution of the program
- `SewingPatternEditor.chm` – reference documentation in MS HTML Help format
- `src` – directory containing the source distribution of the program
- `src/PatternEditor.sln` – VS 2008 solution file for the program. To compile the project, open the solution in the IDE and choose Build->Build solution.
- `src/lib/XPTable` – directory containing source code of the XPTable library. Modifications made specifically for this projects are included.
- `UserGuide` – directory containing the user documentation of the program along with notes on program installation. The user documentation is written in Czech, because not all current users of the program speak English.
- `UserGuide/UzivatelskaPrirucka.html` – main file of the user documentation
- `Samples` – directory containing sample sewing patterns mentioned in the user documentation
- `Thesis.pdf` – electronic version of this thesis
- `dotnetfx.exe` – redistributable package of the .NET Framework 2.0
- `readme.txt` – short summary of the CD contents

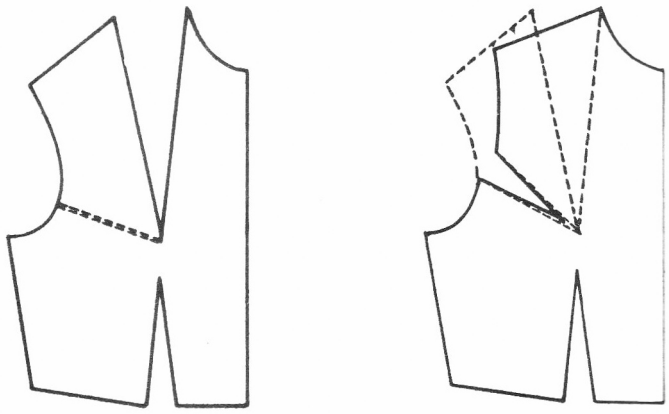
6.2 Appendix B - Introduction to sewing pattern design



Human body mapped to two dimensional plane, picture taken from [16].

The standard pattern design process starts with drawing a base pattern, sometimes called a **sloper**. Nearly all garments can be created from only several slopers or their combination, the most important slopers being blouse, jacket, trousers and skirt (in that order)[16]. For every basic sloper there exist several well-known geometric constructions describing how to create the sloper's outline given body measures of the customer [16][17][18][19][2]. It is to be noted that these constructions are not necessarily exactly equivalent, but since the fabric cutting and sewing process is in itself inaccurate and the fabric is quite flexible, accuracy in a mathematical sense is of no practical importance. Not to mention the fact that even with the best pattern design, custom fitting on the body of the customer is needed anyway for more complicated garments.

When the sloper is constructed, it is transformed to the desired pattern with several types of adjustments, most notably dart transfer, pleating, gathering and cutting. These adjustments may be easily described in terms of geometry. There is although often some room for the designer's experience, because the exact amount of "how much" of the adjustment is applied (e.g. the amount of fabric that overlaps after transferring a dart) is not defined and depends on the intentions of the designer. The pattern adjustments are the most important step in the design process: the standard sloper is transformed to pattern unique to the designed garment.

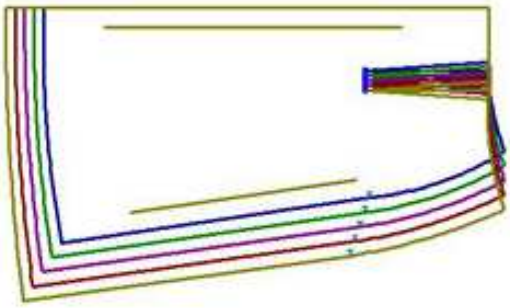


Dart transfer on blouse, picture taken from [2]

After the pattern is adjusted, several markers are added to the pattern. Among the most important being **seam allowance** and the **grain line**.

In haute-couture, where very unusual pattern design is often required, it is not uncommon that the designer creates the pattern directly, without deriving it from a sloper. The designer then makes use of his knowledge of the human body (see picture above), and the design process involves cutting the pattern from a testing fabric and adjusting such testing pattern on the actual body of the customer.

In custom apparel production the final pattern is cut from paper and used as a template to cut the fabric itself. In apparel manufacturing, the pattern is first **graded** - outlines for multiple sizes of the garment are derived from the pattern. This is not done in any exact way, the grading process is based mainly on experience and empiric rules. Outcome of the grading process are grading rules in the form “for size M move this point 2cm to the top” The graded pattern is then digitized (if not digital already) and transferred as input to a machine fabric cutter.



Graded pattern part.
Picture taken from [7].

6.3 Appendix C - Terminology

Czech equivalents of the terms are provided for convenience.

Adjustment Úprava střihu	An operation that transforms a **pattern part.** The basic shape of the garment is preserved.
Armscye Průramek	The armhole, the fabric edge to which the sleeve is sewn. *
Dart Zášivek	Fold sewn into fabric to help provide a three-dimensional shape to a garment, in most cases of a triangular or deltoid shape. If the fold is big enough, the fabric is cut out. *
Grading	Process of deriving patterns for multiple sizes of garment from a pattern for one specific size.
Grain line Směr osnovy	The direction of warp on a woven fabric.
Pattern Střih	Set of outlines of individual pieces of fabric. This pieces are referred to as **parts.**
Pattern part Díl střihu	Outline of one piece of fabric. Parts are the subjects of **adjustments**
Pleat Sklad	A piece of fabric fold on itself and secured in place. Might be of various types.*
Seam allowance Záložky	The area between the edge and the stitching line on pieces of material being stitched together. *
Sloper Základní střih	A basic pattern design from which other patterns are derived through **adjustments.**
Warp Osnova	Lengthwise yarns (fibers) of fabric through which the **weft** is woven.*
Weft Útek	Yarns (fibers) of fabric drawn under and over the **warp.***

* Marked entries cite or paraphrase eponymous entry in English Wikipedia [20]