Neil Satra

Sketching Charts

Computer Science Tripos, Part II

Pembroke College

May 12, 2014

Proforma

Name:	Neil Satra
College:	Pembroke College
Project Title:	Sketching Charts
Examination:	Computer Science Tripos, Part II, 2014
Word Count:	TODO
Project Originator:	Alan Blackwell (afb21), Neil Satra (ns532)
Supervisor:	Alistair Stead (ags46)

Original Aims

This project aims to explore if users are able to create information visualisations faster, or experiment with it more, if given the tools to directly manipulate their charts. Specifically, it involved:

1. Building an application that lets users create graphical visualisations of their data by simply sketching their desired output, like they would on paper.

2. Evaluating the learnability of the interface, and how it compares to existing tools for creating charts, through a user study.

Work Completed

I have completed all the core work items by successfully building a Chart component in C# for Windows applications. This component lets users make a rough sketch of the chart they want to create with a stylus on their tablet, and then uses sketch recognition to create an actual chart based on their data. It performs the sketch recognition by running data mining algorithms on computed features of the digital ink. It also imports data by parsing a user-specified spreadsheet file. I also designed and developed an interface that exposes this component in a user friendly way, attempting to minimize the learning curve by applying Human Computer Interface principles to match the users' mental model through liveness and direct manipulation. I ran a user study assessing how quickly users learnt how to use it (both native and non-native English speakers), and comparing its complexity to that of the existing charting application Microsoft Excel.

As an extension, I implemented the ability to beautify the hand-drawn sketches in a natural-looking manner to match edits made to the chart. I also implemented the ability to erase parts of the hand-drawn sketch, and have those same changes applied to the chart, to further solidify the metaphor to pen and paper.

Special Difficulties

- Acquiring and fixing the source code of a component used for sketch recognition from the team that wrote it.
- Automating the testing of the highly visual parts of the project.

Declaration of Originality

I, Neil Satra of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Motivation

This project is an exploration of Human Computer Interface concepts governing the interactions of users with tools that let them explore and visualise data.

The design of most charting tools is driven by the choice of interface: the mouse and keyboard. Thus, they usually allow graph generation through one of two means:

1. Configuring a chart template through a number of wizards and dialogue boxes.

2. Manually writing code to generate chart graphics.

Due to these design constraints, these solutions take users a relatively long time to learn how to use.

Users need to familiarise themselves with how the choices in the wizard, or the commands in the language, translate to graphics.

A better solution is to use a metaphor to a system users have already learnt to use - drawing using pen and paper. Such a system would benefit from matching the users' mental model.

Additionally, there is a long lag between the users expressing their intention in current systems, and seeing the results of their changes after they close the configuration dialogue or compile and re-run the code. A better solution would exhibit 'liveness' by immediately accommodating users' changes.

1.2 Background and Related Work

Sketching inputs have been studied since the 1960s (Sutherland, 1964) as more natural interfaces to computers, especially for graphics-related tasks, compared to indirections like the mouse and keyboard. This has largely been motivated by the widely recognised importance of interaction to Information Visualisation (InfoVis) (Lee et al., 2012).

Additionally, the metaphor of sketching on paper can encourage exploratory work due to the ease of creating changes and visually expressing what sort of change one is trying to make, by minimising the gap between a person's intent and the execution of the intent, or as Norman and Draper (1986) call it, the 'Gulf of Execution'.

Meanwhile, there has been increasing adoption of touch-enabled phones and multi-touch slates amongst the general public, demonstrating people's affection for what have been referred to as Natural User Interfaces (Lee et al., 2012).

Much more related work is discussed later, in the context of design decisions for this project.

1.3 Project Description

This paper describes a different interface, which allows the user to sketch a subset of a chart on their computer touch screen like as would on paper. The hypotheses are that:

- H1 This interface is more 'learnable' over time
- H2 It encourages exploratory data visualisation creation by making modification easier

when compared to other charting applications. Both these hypotheses were investigated through a user study.

The end result is a proof-of-concept charting application that works as below:

1. The user imports data from a Microsoft Excel file.
2. They sketch a rough indication of a chart.

3. They drag the data onto elements of the chart to actually bind the data to the chart.
4. The tool then creates a 'formal' chart.
5. The tool transforms the user's original sketch to more closely match the formal chart, making the mapping between sketch and formal chart elements evident to the user.
6. Any changes on either the sketch or formal chart is fed through to the other view. For example, erasing the a sketched bar removes a data series from the formal bar.

Chapter 2

Preparation

This project involved vast exploratory design work, in preparation of the actual implementation.

2.1 Requirements

This project could have gone in numerous different design directions from the start. Since the functionality and its benefits over existing systems depended heavily on the design chosen, it was hard to separate what benefits the program would achieve from what features the program would include and how it would expose them to the user.

However, in order to focus our exploration and make sure that the focus was on achieving some real deliverables for end users, a requirement analysis was necessary. The following functional goals were listed based on the project proposal, which focus on what basic tasks the system must help the user achieve, while allowing leeway in how the program exposed and implemented these features.

The system must allow the user to:

Core 1: Use any data they have in reasonably arranged formats in common file types.

Core 2: Specify the type of chart they want by drawing a likeness of it on screen using a stylus.

Core 3: Bind the data to the chart using an interface that makes it clear how the data is affecting the visualisation.

Core 4: Specify visual, size and positioning properties of the chart through the sketches.

Core 5: Manipulate the visual appearance of the created chart.

In addition, time permitting, the system may:

Extension 1: Transform user-drawn sketches to show the visual link to the formal chart elements.

Extension 2: Feed back manipulations applied to the formal layer back to the user drawn sketches, in order to keep the visuals of the formal and sketch layers in synchronisation.

Extension 3: Allow users to undo actions by erasing sketches, and remove the corresponding formal elements without throwing errors.

Extension 4: Allow users to manipulate any property of chart elements, not just one, so that the domain of visualisations they can create is infinite. For example, allow them to bind not just the height of bars in a bar chart to data, but also their width and colour.

Extension 5: Analyse the data and infer properties that may allow it to automatically suggest properties of the chart, such as which field belongs on which axis, or whether a data series should be log scale or linear scale.

Extension 6: The user must be able to export the chart as a Microsoft Chart object that can be embedded as a dynamic object in Microsoft Office files, not just as a raster image.

The core of this project focusses on making more usable software, rather than providing additional functionality, compared to existing systems. Hence, some usability goals were also specified:

Usability 1: Users must be able to create charts at least as quickly as they can using current systems.

Usability 2: Users must be able to build a mental model of the software's behaviour within 2 uses of it. They should thus be able to accurately predict the consequences of any action taken within the software.

Usability 3: Changes to the information visualisation must occur through directly manipulating the visual representation of the chart, rather than through disconnected User Interface widgets.

Usability 4: The user must be able to easily try out changes to the visualisation, see what that would look like, and undo them if needed.

2.2 Design Goals

In order to meet the usability requirements, some design principles need to be followed. The main benefit of a sketch-based interface is that it is more likely to match the user's mental model, or their expectation of what every UI widget will do, since it draws upon familiar metaphors, and offers liveness and direct manipulation of the visualisation. This section explores the theories of direct manipulation and liveness, and how they might be applied to this project.

Direct Manipulation

> Direct manipulation systems offer the satisfying experience of operating on visible objects. The computer becomes transparent, and users can concentrate on their tasks. [It] permits novice users access to powerful facilities without the burden of learning to use a complex syntax and lengthy list of commands. Direct manipulation involves three interrelated techniques:
>
> 1. Provide a physically direct way of moving a cursor or manipulating the objects of interest.
> 2. Present a concrete visual representation of the objects of interest and immediately change the view to reflect operations.
> 3. Avoid using a command language and depend on operations applied to the cognitive model which is shown on the display.
>
> (Shneiderman, 1983)

The mouse and keyboard already provided more direct manipulation than the previous command line interfaces; stylus-based and touch interactions are the next step. A stylus provides a more direct way of moving the cursor and manipulating objects of interest than a mouse since it interacts directly with objects on screen. Besides the inherent benefits of stylus interaction, this application provides a concrete visual representation of the objects of interest, since all configuration settings for the chart are represented as visual parts of the chart rather than options in a configuration window far removed from the chart itself. Additionally, changes are made directly to the chart, resulting in instantaneous feedback, rather than being made to a settings window which then updates the chart when closed.

Live Programming

Blackwell (2002) suggests that classical definitions of programming need to be re-thought based on the evolving use of computers. Specifically, it is suggested that:

1. All computer users must now be regarded as programmers.
2. All software applications now include programming languages.
3. These tools only differ in their usability for the purpose of programming.

Ko et al. (2011) clarify that this is end-user programming - "Most programs today are not written by professional software developers, but by people with expertise in other domains working towards goals for which they need computational support... Although these end-user programmers may not have the same goals as professional developers, they do face many of the same software design challenges..."

Since one of the alternatives to this project is to actually use programming frameworks like D3.js to generate charts, the task of instructing the computer on how to turn data into graphics is rightly considered programming. One of the properties that improves the usability of such a programming tool is liveness.

> "Liveness in programming environments generally refers to the ability to modify a running program. Liveness is one form of a more general class of behaviors by a programming environment that provide information to programmers about what they are constructing."

(Tanimoto, 2013)

This application allows you to directly draw or erase lines in the chart, for example, rather than making changes to textual code and then re-running the code to verify that it does create the change you intended. Thus, it provides a live programming environment, whose benefits have been previously documented (Tanimoto, 2013).

2.3 Work Items

The following broad work items were identified as necessary to achieve the objectives above:

1. Get the requisite approvals for the human study from the Ethics Review Committee.

2. Assess the various methods to build a classifier for ink recognition, including using the RATA Framework.

3. Run an initial user study to see how people naturally draw graphs, and also use it to collect training examples for the classifier.

4. Build a UI that accepts strokes, runs them through the classifier, and shows the user feedback to indicate successful recognition.

5. Build the UI widget that lets users import their spreadsheets in Microsoft Excel (xlsx) or Comma Separated Value (csv) files. It must then expose the various fields detected.

6. Build the charting component to convert the recognised sketches and the ink into a finished visualisation.

7. Run a pilot study followed by a user study to evaluate the system.

An iterative development process, similar to the Spiral Model and Scrum, was adopted for this project. This allowed early experimentation with and user testing of the various components and different interface designs. After each iteration of development, design choices arose. Various options were evaluated, one was built, resulting in a new iteration, at which point the process repeated.

To manage technical debt and follow best practices, we adopted a form of Test-Driven Development (TDD). In its pure form, TDD relies on a fully planned out functional specification and automated testing. However, since the design and functionality of this project were evolving over time, small specifications were made for each iteration, and changed as necessary after a sprint (to borrow Scrum terminology). Also, since the input needed to sketched by hand, and the output was visual and changes in literal pixel values between trials, it was non-trivial to simulate and verify this interaction automatically. These parts were verified through repeated manual testing each time the code was changed or refactored. Parts of the code that were effectively in a functional programming style (deterministic calculations with no side effects) could, however, be checked with unit tests. In addition, to pick up design flaws early, the prototypes were constantly used for hallway testing (friends and family were asked to use the application and speak out what they were thinking as they did).

Thanks to the constant refactoring of the code to minimise coupling, the project could be open sourced at the end and plugged in as a standard GUI component in other Windows applications in the end, making sure that this software can actually be used.

2.4 Development Environment

For this project, the hardware available was a Microsoft Surface Pro (1st gen), which features the active digitizer screen required for precise inking. Since this machine runs Windows by default, we chose to develop the system using the .NET framework, which has built-in support for Tablet PCs and Ink handling.

As a precaution, insurance was taken out on the machine to ensure quick replacement in case of damage or loss. Additionally, version control was used extensively in the project, to ensure no work was lost. The code for the RATA ink stroke recogniser, described below, was uploaded to a Git repository in collaboration with RATA's authors. The code for this project was then written in a fork of that repository, to allow updates to RATA to be pulled in. The dissertation itself was written in LaTeX, so that the text files could be versioned in another Git repository. Both repositories were backed up off-site on repository host Bitbucket. The dissertation was also backed up online using file synchronisation software Microsoft OneDrive.

2.5 Building the classifier

Core to the system is an ink recognition component that identifies the chart element (e.g. bar, axis) that the user has sketched. This must work above a certain accuracy threshold or the system will prove frustrating to users (Frankish et al., 1995). However, since the project scope included other components too, the time available to build this classifier was limited. Additionally, given the complexity of building a classifier, and our limited experience with building them, it would be difficult to get the same accuracy as those provided in a mature library. Hence, we decided to build a classifier using existing tools rather than implementing one from scratch.

2.5.1 Recognition Method

A number of different approaches have been taken in building systems that automatically interpret hand-drawn sketches. These approaches vary in the recognition accuracy they offer, and the robustness to generalise across multiple domains. Some of the approaches that have been attempted:

- Focus on **defining shapes structurally**. A base vocabulary of primitives like lines, ellipses and arcs is built by describing the properties of such shapes. Shilman et al. (2002) used a hand-coded grammar to describe shapes in a domain as a composition of such primitives. Alvarado and Davis (2004) used dynamically constructed Bayesian networks to scale this process to multiple domains. Hammond and Davis (2006) developed a language to manually describe how diagrams in a domain are drawn, displayed and edited.

- Look at the **visual appearance** of shapes and symbols. Kara (2004) and Ouyang and Davis (2009) used image-based similarity metrics to perform template matching. Shilman et al. (2004) broke up the ink into connected subgraphs of nearby strokes, which were then compared to known symbols. Oltmans (2007) proposed a visual parts-based method that utilise a library of shape contexts to describe and identify symbols in a domain.

- **Compute features** of the ink. Patel et al. (2007) select these features and sets their thresholds statistically. Yu and Cai (2003) use heuristics for the same purpose. Chang et al. (2010); Rubine (1991); Willems et al. (2009) all use machine learning to automatically find relationships between features and choose an appropriate feature set accordingly.

Chang et al. (2010) showed that the first two approaches forego some accuracy, since they rely on the final pixel values of the sketch and so do not fully exploit the rich temporal data stored with digital ink. Within the last approach, they showed that using machine learning allows recognisers to be generated for new domains with less effort than statistical or heuristic methods. Since the domain for this charting application could grow over time, we wanted a recogniser that could be adapted easily over time. Using data mining, this can be done using training data rather than programming effort.

Having decided on using a feature-based approach that selects the feature set by data mining a training dataset, there were a number of alternatives available to us. While most projects, like (Rubine, 1991) and (Willems et al., 2009), rely on one or two data mining algorithms, (Chang et al., 2010)'s RATA.SSR combines the results from four well-performing algorithms in WEKA (Hall et al., 2009) tuned to their best configurations, to provide a more accurate recogniser. RATA.SSR thus outperformed all the other recognisers tested (PaleoSketch (Paulson and Hammond, 2008), CALI (Fonseca et al., 2002), $1 recogniser (Wobbrock et al., 2007)) on domains other than the one they were explicitly demonstrated on.

RATA.SSR also has the advantage of outputting an API that can be plugged into our system after we build the recogniser. Additionally, one of the authors, Beryl Plimmer, previously worked with some members of the Graphics and Interaction Group at the Cambridge Computer Laboratory, and so could be reached for support and source code, which proved to be an invaluable resource.

2.5.2 Data collection

After acquiring RATA, I inspected the code and did manual testing, which revealed some blocking bugs. Since we were in contact with the authors of the software, I was able to confirm with them that these were indeed bugs. I implemented fixes for them and contributed them back to the authors, and am working towards getting the code ready for to be published Open Source.

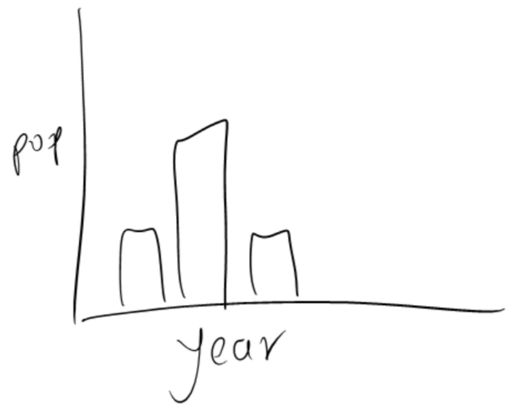
With a working version of RATA, an initial study was run to collect training data. We asked 10 participants (20-26 year old Cambridge students, studying a large variety of subjects) to draw a chart. I spoke out the following prompt:

> Imagine you are a government official trying to use a bar chart to visualise how the population has grown over time. Can you sketch out what this bar chart might look like? Just treat this screen like paper.

They were then presented a simple UI with a large white canvas and the following task description written in a panel:

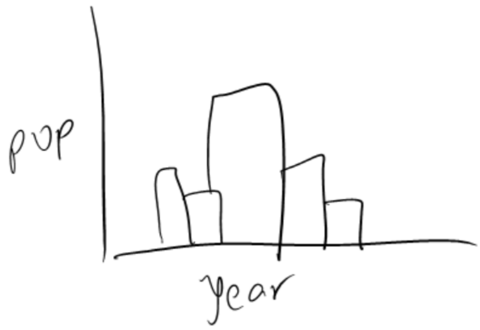
> Draw 2 axes. Label the x axis 'Year' and the y 'Population'. Draw 3 bars of different heights. Each shape (axis, bar) should be drawn in one stroke.

They were asked to draw the same chart 2 times, in order to get 20 training samples in total, and to observe how much variation there is between multiple sketches by the same user. On the second drawing, they were encouraged to draw a less conventional chart, to make the system as robust in the face of variations as possible.



(a) Regular chart

(b) Using lines instead of bars



(c) Grouped bars

(d) Using a mouse instead of the stylus

Figure 2.1: Some of the more unusual chart sketches collected

Three elements were then defined: Axis, Bar and Text (an extra element, 'L Axis' was added later based on feedback from a pilot user study). We went through each figure and labelled the various elements.

Figure 2.2: Elements of the sketch labelled as Axis (cyan), Bar (brown) or Text (dark blue)

Once all the elements in all the charts were labelled, we could begin training. RATA includes a dataset generator tool that allowed easy extraction of various features of the strokes, such as 'distance from first to last point', 'absolute curve of largest segment' and 'pressure variation'. Data for 121 such attributes, about 270 ink strokes was compiled into a .csv file for use in training.

2.5.3 Training

The labelled data was then sent to a 'Vote' classifier in Weka, which combines the probability distributions derived from multiple classifiers (Kuncheva, 2004). Specifically, the types of classifiers combined were Logit Boost, Bayes Net, LMT (Logistic Model Trees) and Random Forest. In order to assess how well each of these individual classifiers were performing, an experiment was set up using Weka Experimenter. The data collected in the initial study was shuffled, and then 66% was chosen randomly as training data, the rest as testing. Then a paired T Test gave the following results:

Tester: Paired Corrected T Tester
Analysing: Percent correct
Confidence: 0.05 (two tailed)

(1)	meta.LogitBoost '-P 100 -F 0 -R 1 -L -1.7976931348623157E308 -H 1.0 -S 1 -I 10 -W trees.DecisionStump' 8627452775249625582
(2)	bayes.BayesNet '-D -Q bayes.net.search.local.TAN – -S BAYES -E bayes.net.estimate.SimpleEstimator – -A 0.5' 746037443258775954
(3)	trees.LMT '-P -I 50 -M 200 -W 0.0 -A' -1113212459618104943
(4)	trees.RandomForest '-I 100 -K 0 -S 1' -2260823972777004705

Table 2.1: Classifier Algorithms Used

Dataset	(1)	(2)	(3)	(4)
'Initial study'	97.05	98.15	98.80	96.07

○, ● statistically significant improvement or degradation

Table 2.2: Comparison of algorithm accuracy

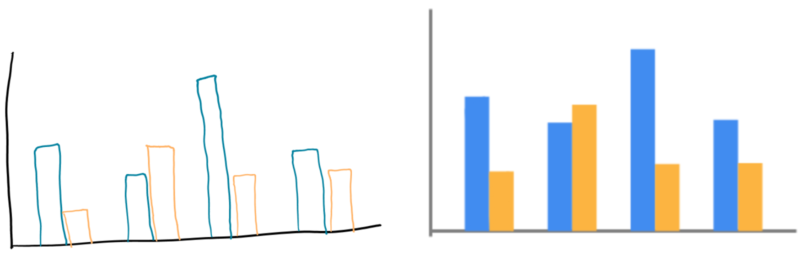
Chapter 3

Implementation

3.1 Design

Now that we had shown that a classifier could be built with relatively small sets of training data that still performed well, we had to design the interface to expose this functionality. This largely involved assessing tradeoffs between choices.

3.1.1 'Sketchy' or Formal



(a) Sketchy look (b) Formal look

User content can be in shown in two styles: sketch or formal. This refers to the usability dimension described by Bresciani et al. (2008) as 'Perceived Finishedness'. Yeung et al. (2008) showed that a doodle-like appearance of low-fidelity prototypes encourages early-stage experimentation and discussion, whereas a formal appearance looks finished and professional. Thus, it had to be decided whether the system should generate the rest of the chart with a 'sketchy' look, or convert the input into a finished chart with a formal look. Blackwell et al. (2008) note how the rapid replacement of pen and paper by computers has made more of our processes formal and precise, potentially discouraging us from exploration and iteration in creative processes that would benefit from them.

Generating the 'sketchy' look can be non-trivial, (Plimmer et al., 2010; Wang et al., 2011) e.g. if the user draws one bar, how should the system generate more bars that look hand-drawn by the same author but not just like stretched versions of the first bar? Additionally, if the users want to present these charts to an audience, they must look polished, and so at some point an export option for a formal look needs to be offered. Thus, we decided to offer a hybrid that shows the user's input in sketch form, with the system's output in formal form. Extensions were implemented to keep these two forms as closely in synchronization as possible, thus making the mapping visually evident.

3.1.2 Sketches or gestures

A related design decision was whether to use sketches or gestures. Sketches share some visual semblance with the chart they're trying to indicate, whereas gestures are just movements chosen from a library of easy-to-distinguish movements, that behave like commands instructing the software to choose a certain chart type. SketchInsight (Walny et al., 2012) uses the latter approach, requiring, for example, just an 'L' shape to indicate a bar chart, or a 'V' shape to indicate a line chart.

Pros

- This would provide higher recognition accuracy, since the domain of shapes to distinguish between is limited, and can be designed to maximise differences between them.

Cons

- A gesture is transient, unlike a sketch. Users wouldn't be able to modify their input, they would have to redo their actions.
- One of the principles of direct manipulation is to avoid using a command language, but a gesture library is exactly that, requiring users to remember which gesture corresponds to which chart type or element.

This would provide higher recognition accuracy, since the domain of shapes to distinguish is much more limited, and can be designed to maximise differences between them.

3.1.3 Modes or modeless

Since the application now has both sketch and formal content, the user must have an easy way to switch between the two views. One approach is to give the user an explicit UI widget to toggle between the two modes. This way, the user explicitly indicates what they want to see, and thus should have a better understanding of what state the system is in. This also allows the system a chance to change the controls available to the user.

The other approach is to avoid modes, requiring lesser cognitive effort from the user since they don't need to keep track of what state the system is in. Norman and Draper (1986) describe this as the Gulf of Evaluation - the difficulty of assessing the state of the system. In this project, this could have been done by showing the sketch view when the user was about to edit the chart. The active digitizer hardware allows the system to detect when the user brings the stylus within range of the screen, just before they actually touch the screen, allowing the chart to be in sketch view by the time the stylus is down. When the stylus goes out of range, the system can switch to the formal chart view. This would solidify the metaphor that edits are done to the sketch, but the final product to be looked at is the formal view.

At first, the modeless version was chosen for its lower cognitive overhead. However, when the extension to allow edits not just to the sketch view, but also the formal view, was undertaken, we had to switch over to a mode-based system to allow the user to interact with the graph in both views with their stylus.

3.1.4 Stock or bespoke charting widget

Since the system is generating a formal version of the chart, a charting component is required to render this visualisation. The .NET framework comes with built-in chart controls that offer basic functionality with relatively low implementation effort. They also allow easy export as dynamic chart objects into Microsoft Office files. However, customising their appearance beyond a certain point is extremely difficult, making it easier to just make one's own charting component from scratch and control all aspects of the rendering. This means having to re-implement a lot of core functionality though, such as scaling shapes correctly, choosing labels that are round figures when possible, and generating colours than work well together for different data series. This also means that the chart can only be exported as a raster image rather than as a chart object.

To enable rapid prototyping, we chose to utilise the standard charting component at first. As our needs to customise the chart grew, we were able to make our own chart class that implemented the same interface as the standard component, and so could be slotted in to replace it.

3.1.5 Finite or infinite domain

Some tools, such as Microsoft Excel, let the user make one of a limited set of charts, such as bar or pie charts, quickly. Others, (which usually involve coding), such as D3.js, let the user make a vast variety of visualisations by creatively combining basic elements like lines, boxes and wedges. However, these require expert knowledge of the tools, and take longer to create basic visualisations.

	Library of charts	**Modular charts**
Description	Draw basic gestures or elements to indicate which chart type is desired	Draw any one of 7 basic components and bind data to attributes of theirs such as width, height, colour or radius
Domain	Finite domain	Infinite domain
Speed	Quick	Slower
Complexity	Simple interface, just drop data on an element to bind data	Complex interface to expose all attributes and manage data binding

Chao et al. (2010) uses pen gestures to indicate 'proto-objects' that can be combined. While an infinite domain system that uses pen sketches would have been intriguing to explore, it would contradict the project's primary usability goal that the system should be faster than users' current systems. Additionally, the 80/20 rule indicated that while a few power users may want to generate custom visualisations, the majority would just want to make simple charts. Thus, the additional functionality didn't justify the additional complexity for the average user.

3.2 Development

Interleaved with the design process above was the development process detailed in this section. The C# code supports a stand-alone Sketch Chart component that can be used in any Windows Forms application that requires charting functionality, as well as a reference implementation of such an application, Sketchography, which lets users import their Microsoft Excel or Comma Separated Values data and generate charts. This satisfies all the core requirements described in section 2.1, as well as 3 of the 6 extensions outlined.

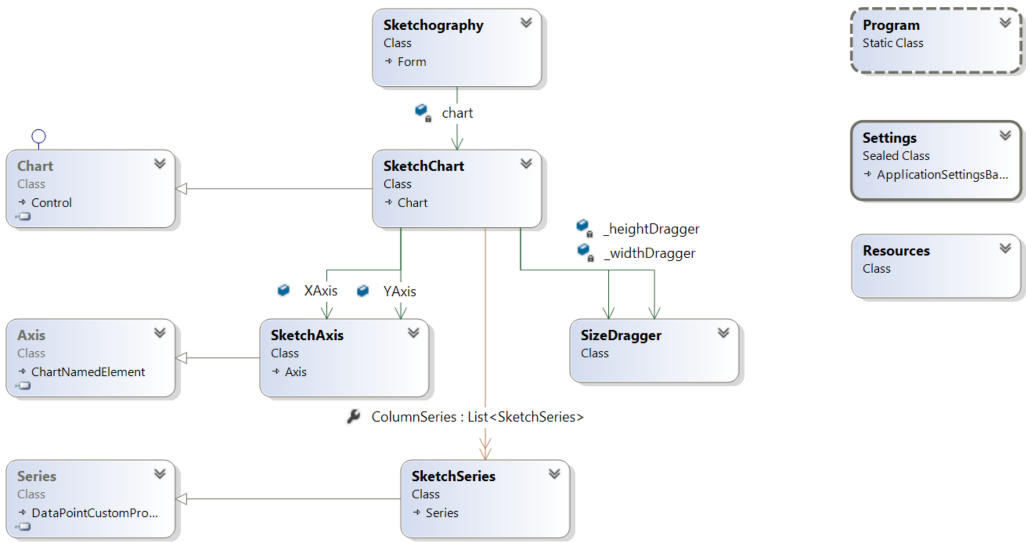


Figure 3.2: Interactions between classes

As seen above, the Sketchography form is the main application run when Program starts. Sketchography contains a SketchChart component, which contains the canvas for users to sketch on, as well as the formal chart generated. SketchChart in turn uses SketchAxis and SketchSeries objects. SketchChart, SketchAxis and SketchSeries inherit from their non-sketch counterparts Chart, Axis and Series (from System.Windows.Forms.DataVisualization.Charting, which is part of the .NET framework), in order to reuse the built-in functionality and members, and complement them with functions and members specific to sketching. SketchChart also references two instances of SizeDragger, which is a custom UI widget to support dragging ends of columns in a column chart to scale them. Settings and Resources are two additional classes used to store properties of the project.

No class is more than 400 lines of code, indicating that functionality has been split up at a reasonably fine grain to not concentrate too much responsibility in any one class. Visual Studio's calculated code metrics show that the project has a maintainability index of 74/100, which gets the highest rating - 'good' maintainability. Should I include the data in this paragraph?

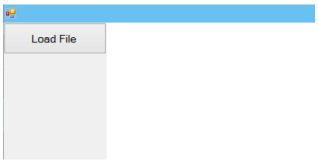
At a high level, the functionality of the program can be divided into 3 responsibilities - data handling, sketch processing and charting (in increasing order of complexity). It is written in an Object Oriented fashion, with separation between the views (Windows Forms) and controllers (C# classes), to allow for easy testing.

3.2.1 Data import and management

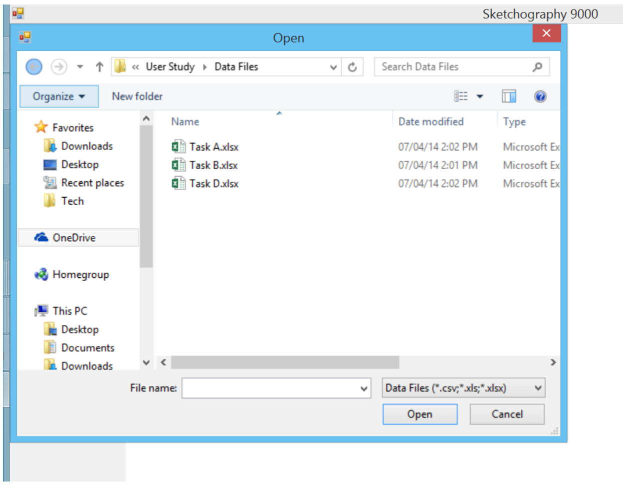
Since the application is targeted at the average user, their data is most likely to be stored in spreadsheet format. Thus, it is important to allow them to import data from .xlsx and .csv files. For the sake of simplicity, the code assumes that the data is well-formed. Specifically, it works on the following assumptions:

1. The data is arranged as records in the rows of the spreadsheet.
2. The first row contains the names of the various fields.
3. No data is missing (if there are m columns and n rows, there are $m \cdot n$ data values.

Under these assumptions, importing tabular data is a common use case, so I studied a number of existing libraries and methods to do this in C#. At first the LinqToExcel package from the NuGet package manager was used to easily import data from the file. However, this package had limited documentation, making it hard to achieve more complex tasks. Additionally, it added an external dependency, and carried a license that allow free reuse of the software as long as the original copyright message was included. The package was helpful for rapid prototyping at first, but after the code was more mature,



(a) Initial state



(b) When "Load File" is clicked

(c) Column headers are shown with data previews in tooltips

we removed this dependency and implemented the file import ourselves using the OLE DB provider.

The interface shows a "Load File" button, which when clicked reveals a standard system filepicker dialog with a filter to show *.csv, *.xls and *.xlsx files. Selecting a file causes the "Load File" button to disappear and be replaced by a list of column headers (the names of the various fields). This helps minimise interface clutter and only expose the most relevant information at all times. However, it comes at the cost of the user needing to hit 'Reset' if they choose the wrong file by mistake. To reassure the user that data has been imported correctly, and for them to check which column header corresponds to which data, hovering over any header reveals the first few entries for that field as a sample.

3.2.2 Sketch Processing Workflow

RATA.SSR provides a sample that calls the classifier API. For this project, that reference was followed, with a lot of extraneous code and unnecessary branches removed, until a basic prototype existed consisting of nothing but a blank canvas which receives user ink, passes it off to a classifier, and outputs the recognition result in a log-like text box. This was then integrated into the application described above, with its data import facility.

When the user draws a stroke on the SketchChart element within the Sketchography window, a number of steps occur:

1. The `SketchChart` is covered by an `InkOverlay` object, which receives the stroke. An `inkOverlay.Stroke` event is fired, which allows our event handler to run custom code.

2. The `InkOverlay.Stroke` event handler calculates additional features of the stroke that are not included in the stroke's properties by default, and stores them in its extended properties.

3. The stroke is then sent to the classifier, which is loaded from a file on program initialization.

4. The `classifierClassify` function returns a string from a set of previously defined strings representing the various recognition results. This string is passed off to the `ConvertToFormal` method, which, as the name suggests, creates the corresponding formal chart elements. The `ConvertToFormal` method is also sent the stroke object itself, so that it can use its properties such as the location to place the formal object accordingly.

After the core of the project was finished, an extension was implemented that allows erasing of strokes. The stylus available has a button simulating an eraser at the back. When this eraser comes into range, a `CursorInRange` event is thrown, wherein we can check whether the stylus is inverted or not.

```
_inkOverlay.EditingMode = e.Cursor.Inverted ?
                    InkOverlayEditingMode.Delete :
                    InkOverlayEditingMode.Ink;
```

Then, when a stroke is received (the same event is thrown whether the stroke is ink or the eraser), if the editing mode is Ink, the steps above are run. If it is Delete, the corresponding chart element is detected and reset. On the next re-paint of the interface, this element will be removed from the canvas. To minimize lag between the action and the feedback, this repainting is forced immediately with an `Invalidate()` call.

Below is a description of how `ConvertToFormal` responds to user actions it receives, via the stroke classifier.

<table>
<tr><th>User Action</th><th>Application Response</th></tr>
<tr><td>Load File</td><td>Bind chart’s data source to the in-memory data table created from the imported file.</td></tr>
<tr><td>Draw an axis</td><td>1. Calculate the bounding box (the smallest rectangle that entirely encloses the stroke) of the sketched axis, and use a heuristic comparing the height to the width to determine whether it is a vertical or horizontal axis.

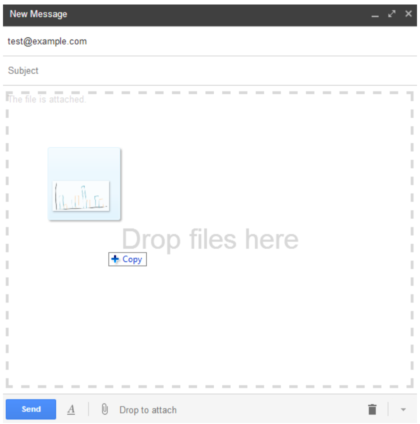
2. The line running through the center of the bounding box is taken as the intended coordinates of the formal axis. This must be done since sketched axes are rarely perfectly straight lines, so using the actual endpoints of the line might result in a non-aligned axis.

3. The endpoint coordinates of the straightened line are used to set the vertical or horizontal position and size of the formal chart. They are also saved in an Axis object, which can then be drawn onto the formal chart.

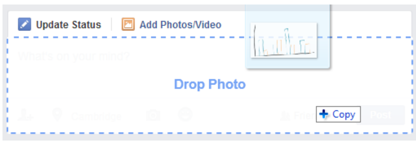
4. If it a horizontal axis, a drop target is below the axis. This is a rectangle of the same length as the axis, and a preconfigured height deemed big enough to make it hard for the user to miss, while minimising the screen space used up.</td></tr>
<tr><td>Draw a bar</td><td>1. A bar is the indicator that the user wants a new data series added on a bar/column chart. Thus, a new data series is added to the chart’s SeriesCollection in the form of a Series object. The name of this series is set as “SeriesX”, where X is incremented for each new series. This will be useful for the drop target system described later.

2. When the new series is added, the Chart object automatically assigns it a new series colour from the colour palette. The application changes the colour of the stroke the user drew to this series colour, to give feedback that it has been recognised as a series.

3. A drop target the same size and position as the bounding box of the bar is created, corresponding to that series.</td></tr>
</table>



(a) Gmail.com compose window, taken in May, 2014



(b) Facebook.com status update widget, taken in May, 2014

In response to strokes that create chart elements, drop targets are generated so that the user may then drag and drop data directly onto the chart element to bind it. This provides direct manipulation, since it doesn't force the user through a menu and dialogue system to configure which data corresponds to which axis. In order to do this, a dictionary is maintained in SketchChart, storing the mapping between Rectangles encoding the location of a drop target, and the string representing the chart element it corresponds to. For the x axis, this would be "X Axis"; for the data series it would be the name of the series.

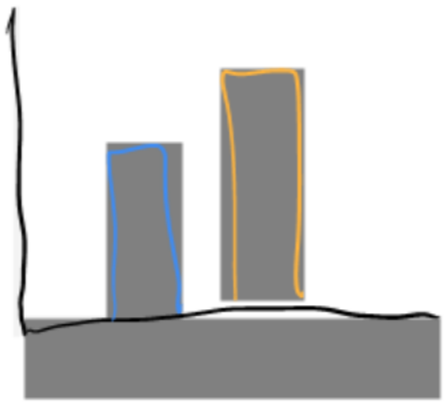


Figure 3.5: Drop Targets for 2 data series and the X axis, revealed when data is being dropped onto the chart

When the user begins dragging a column header from the list on the left, this calls a number of event handlers in Sketchography. These configure the drag and drop action so that the column name saved within the list item is the data sent to the drop location. They also flip a `ShowDropTargets` flag, so that the chart rendering system draws grey rectangles indicating where the drop areas lie. This UI pattern is based on one users may be familiar from a number of drag and drop interfaces they may have previously interacted with, thus making it easier for them to understand what's happening. We also tried adding 'Drop here' text in the targets like in the examples below, but this proved to cluttered in the small drop targets being used. Additionally, hallway testing revealed that once users began dropping, they were already aware that they must drop it in targets, and the change in colour was conspicuous enough to draw their attention to them.

When the data is dropped, this calls another event handler. In Sketchography, the event handler simply verifies that its a valid drag and drop, and then passes on the data to the `DropData` function on the SketchChart. For every rectangle in the dictionary of drop targets, `DropData` checks if the drop event occured within it. When it finds the drop target the user picked, it looks up the corresponding string, and calls `BindDataSeries` on it. `BindDataSeries` adds the binding appropriately. If the data was dropped on a data series rather than the X axis, it also renames that series and the corresponding drop target, so that the user can reassign that bar to some other data if they change their mind. From this point, the charting functionality takes over.

3.2.3 Charting

This part of the functionality, unexpectedly, proved the most challenging and time-consuming. In the first few iterations, a built-in .NET chart component was used, which allowed rapid prototyping. However, converting from the coordinate system used within the charting component to that used in the rest of the window was proving non-trivial and inaccurate. In addition, the component didn't allow very fine-grained positioning of its various elements, or much control over their sizing and other attributes. Hence, we had to implement our own chart control. To make the most of the functionality already implemented in the built-in control, we inherited from it. This also meant our custom control could easily replace the built-in one since it implemented the same interface but with a few extra features.

The painting of the chart graphics is done in a largely procedural fashion unfortunately, as this is how it is done for Windows Forms controls. Most of the work is done in the `PostPaint` event handler, which is called after the built-in chart is done painting (even though the built-in chart control is not shown to the user at all, it still needs to be painted in a hidden background for some of its functionality to work correctly). This painting occurs whenever changes are made that might affect its output. Within the event handler, there are distinct sections that:

- Draw a background to cover up the built-in chart.
- Draw the formal axes and bars.
- Draw a height drag handle if needed.
- Draw a width drag handle if needed.
- Move the user's strokes to fit the bars.
- Draw the drop targets if needed.
- Change the colour of ink strokes to match their series colour if needed.
- Update the legend.

Each of these steps involves a lot of complicated computation, maintaining of state, and conversion between coordinate systems. The particularly interesting parts are described below.

Positioning and scaling bars

To allow maximum flexibility, constants are introduced that control various parameters of the appearance of the bars, such as the ratio of the gap between them, and the fraction of the height of the chart they should cover by default. Then, formulae are used to account for the number of data series, the relative width scale of each series, the relative scale of the gap between data points, and the length of the X axis, to calculate the position each bar should be drawn at.

The application also has to calculate the values to show for labels on the Y axis. This is something that the custom charting control doesn't handle as elegantly as the built-in control, since the values are not usually rounded, whole numbers. This is because the range of values in the data series is simply spread across a set number of labels to determine the individual label values.

Transforming strokes to match formal bars

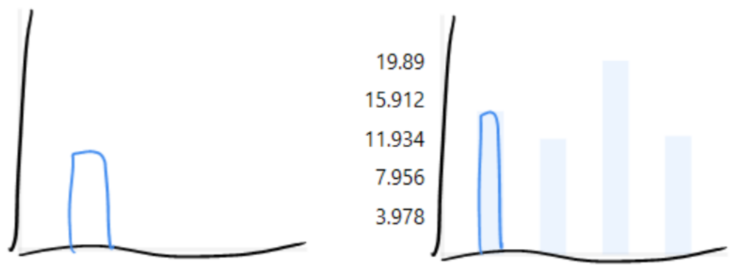


Figure 3.6: Before and after the transform

This was implemented as an extension, to better express the connection between the sketches and the formal bars to the user. It is done by maintaining a mapping between the sketch and the first bar of the series. When the bar is drawn, its coordinates and dimensions are saved. These are then applied as a scale and translation transform to the associated sketch.

Height and width dragging

Another extension was to allow users to make changes to the formal chart, and have those changes feed back to the sketches. We decided to offer the ability to change the width and height of the bars. When in the formal view, if users hover over the first bar in a series, they see a drag handle each for the height and width, in the form of a grey rectangle. We decided on the size of these handles by experimenting with various sizes to see which one formed a target big enough to grasp with a stylus. Since these rectangles are simply graphics drawn onto a picture box rather than explicit controls, they do not offer native drag functionality. Instead, this has to be approximated by toggling a flag when mouse/stylus is down within the rectangle of the drag handle indicating that a drag has started. When the mouse/stylus is lifted, if the flag was on and the cursor position has changed, this is viewed as a drag, and the change in position is taken to be the size change desired. This

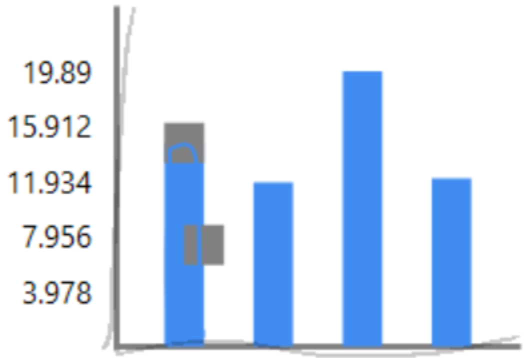


Figure 3.7: Drag handles

scaling is then applied to the variables previously introduced for positioning and scaling bars, so the changes take effect in the next paint cycle, which seems fairly instantaneous to users, if not perfectly so.

To help users discover what these drag handles do, the cursor changes to a sideways drag cursor in the direction corresponding to whether it is a height or width handle, when it hovers over these handles.

At first, the naive implementation of the scaling broke when users tried particularly extreme drags, which they tended to play with even if it wasn't the scale they finally intended to use. The calculations had to be refined over time.

Maintaining the legend

To keep the user aware of the state of the system at all times, it is important to have a legend showing which data is connected to which element of the visualisation. While most charting tools have a separate legend, we decided, in the theme of direct manipulation, that the colour key should be right where the data is - in the list of column headers.

However, the list of column headers is in the Sketchography form, whereas the colours are stored within the SketchChart, which is referenced in the form. We followed a subscriber pattern here, by creating an event in SketchChart that notifies about changes to the legend. SketchChart also exposes a public Legend dictionary, mapping colours to headers. Sketchography subscribes to those notifications, and its event handler regenerates the legend by checking

Figure 3.8: Legend

the list in SketchChart.

Switching between the formal and informal views

We initially did this by offering a simple toggle button that read 'Toggle View', and a text box that showed the current view the user was in 'Sketch View' or 'Chart View'. However, during hallway testing, users repeatedly clicked on the text box, or were just generally confused about which view they were in. This confirmed a previous worry that the disconnect between the action trigger and the reflection of its actions would be confusing, so we implemented radio buttons as shown below. These contain pictures of the two views, so users can spot the difference visually without needing to internalise what the two phrases represent. Another concept of showing visual layers with live previews was not tested because of time and technical constraints.

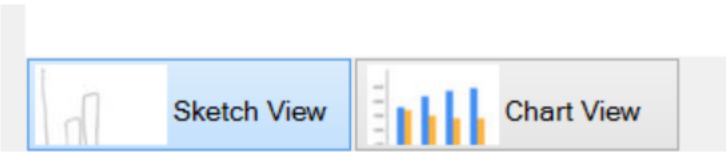


Figure 3.9: Switching between formal and informal views

Behind the scenes, all the chart elements are drawn in both views, with the unselected view dimmed or ghosted out. Hence, when users are sketching, they can see faint feedback of the recognition in the background and thus notice early if there has been a misclassification. This is done by setting two alpha or transparency values - one which is high when the formal view is selected, and one which is high when the in informal mode. Each of the elements then uses one of these alpha values to draw themselves according to which mode they should be prominent in.

Chapter 4

Evaluation

```
> t.test(Task1, Task2, paired = TRUE)

Paired t-test

data:  Task1 and Task2
t = 5.2186, df = 9, p-value = 0.0005502
alternative hypothesis: true difference in means
is not equal to 0
95 percent confidence interval:
 23.34058 59.05942
sample estimates:
mean of the differences
                   41.2
```

The test is valid if the data follows a normal distribution. For each of the tasks, below is a Q-Q plot comparing the actual values (points) to the theoretical normal (line), and the results of the Shapiro Wilk test of normality.

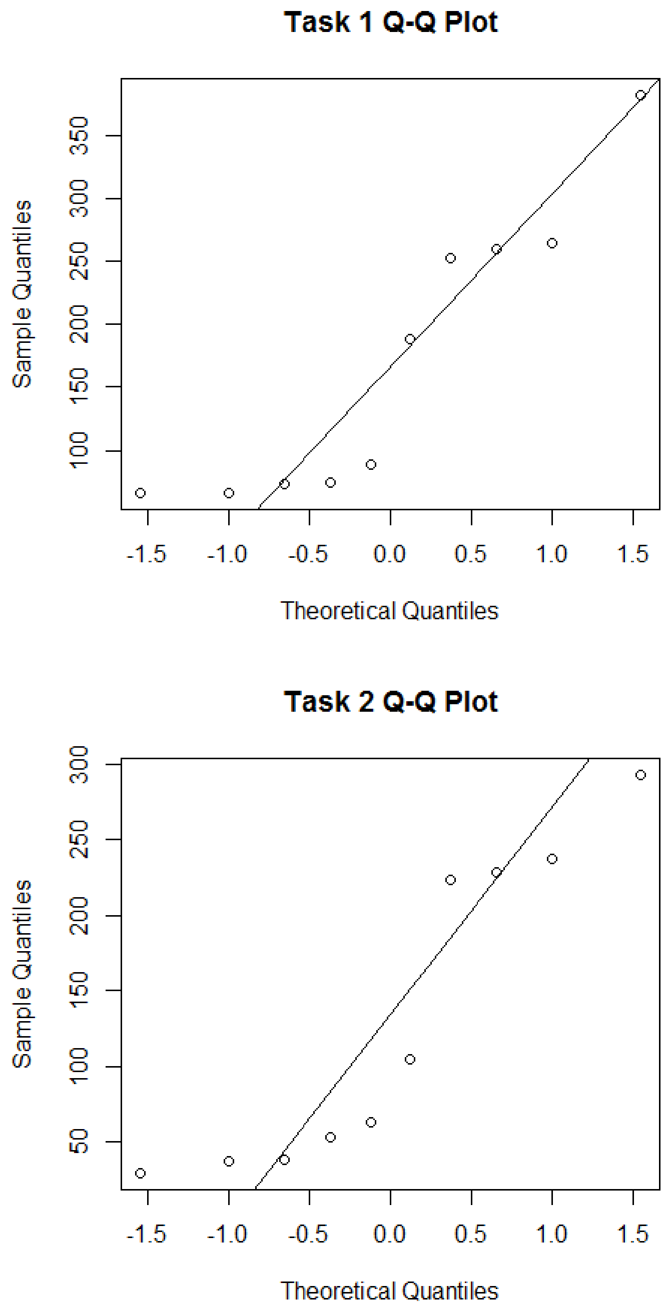


Figure 4.1: Normality Tests for Creation Tasks

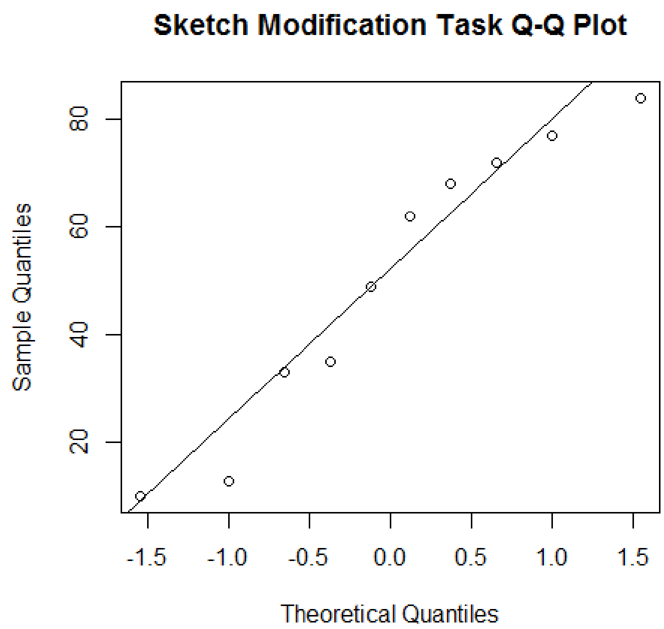
Dataset	W	p-value
Task 1	0.8435	0.04854
Task 2	0.8317	0.03511

Similar evidence that people took longer to change the bar height in Excel than in Sketchography.

```
> t.test(Sketch, Excel, paired = TRUE)

Paired t-test

data:  Sketch and Excel
t = -4.5484, df = 9, p-value = 0.001389
alternative hypothesis: true difference in means
is not equal to 0
95 percent confidence interval:
 -83.55241 -28.04759
sample estimates:
mean of the differences
                  -55.8
```



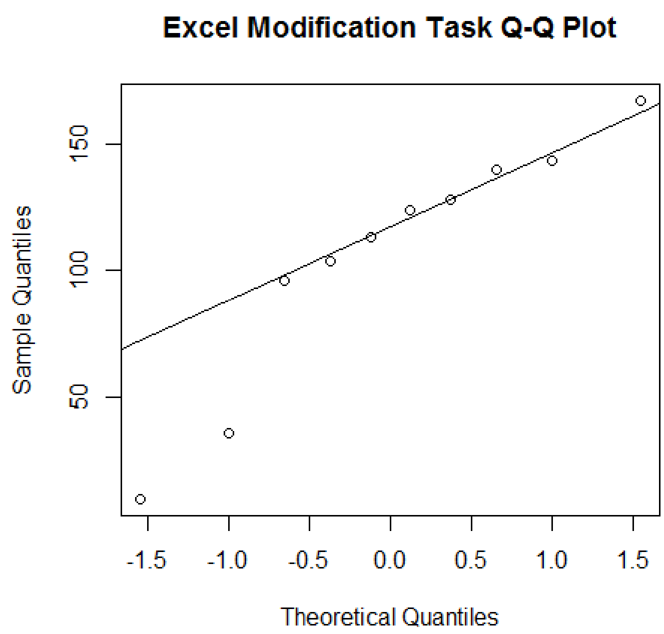


Figure 4.2: Normality Test for Modification Tasks

Dataset	W	p-value
Sketch	0.9426	0.3966
Excel	0.8943	0.1896

Chapter 5

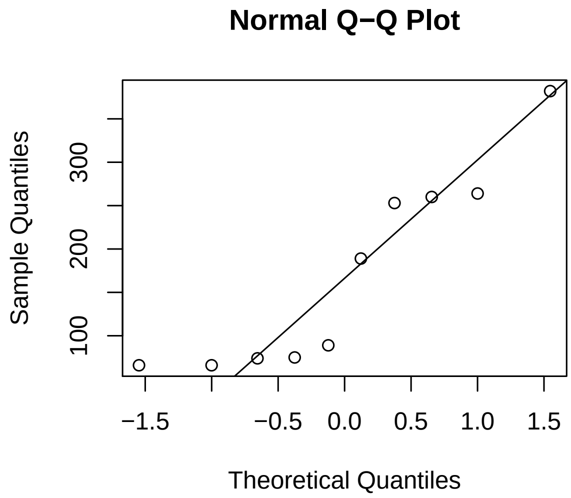
Appendices

Irrefutable evidence that users learnt how to use the software and got faster after one try.

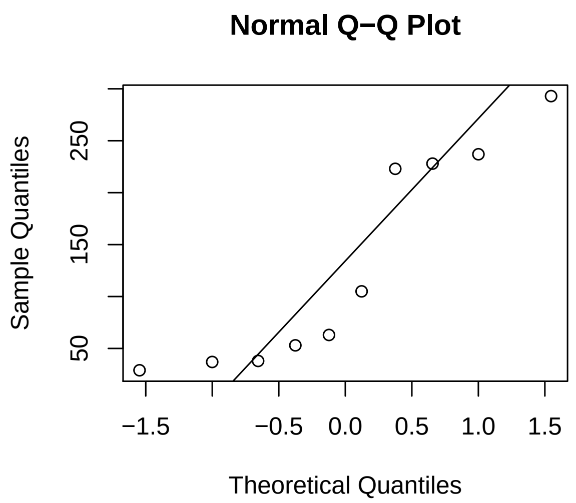
```
t.test(Task1, Task2, paired = TRUE)

##
##  Paired t-test
##
## data:  Task1 and Task2
## t = 5.219, df = 9, p-value = 0.0005502
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  23.34 59.06
## sample estimates:
## mean of the differences
##                    41.2
```

The test is valid if the data follows a normal distribution. For each of the tasks, below is a Q-Q plot comparing the actual values (points) to the theoretical normal (line), and the results of the Shapiro Wilk test of normality.




```
##
##  Shapiro-Wilk normality test
##
## data:  Task1
## W = 0.8435, p-value = 0.04854
```



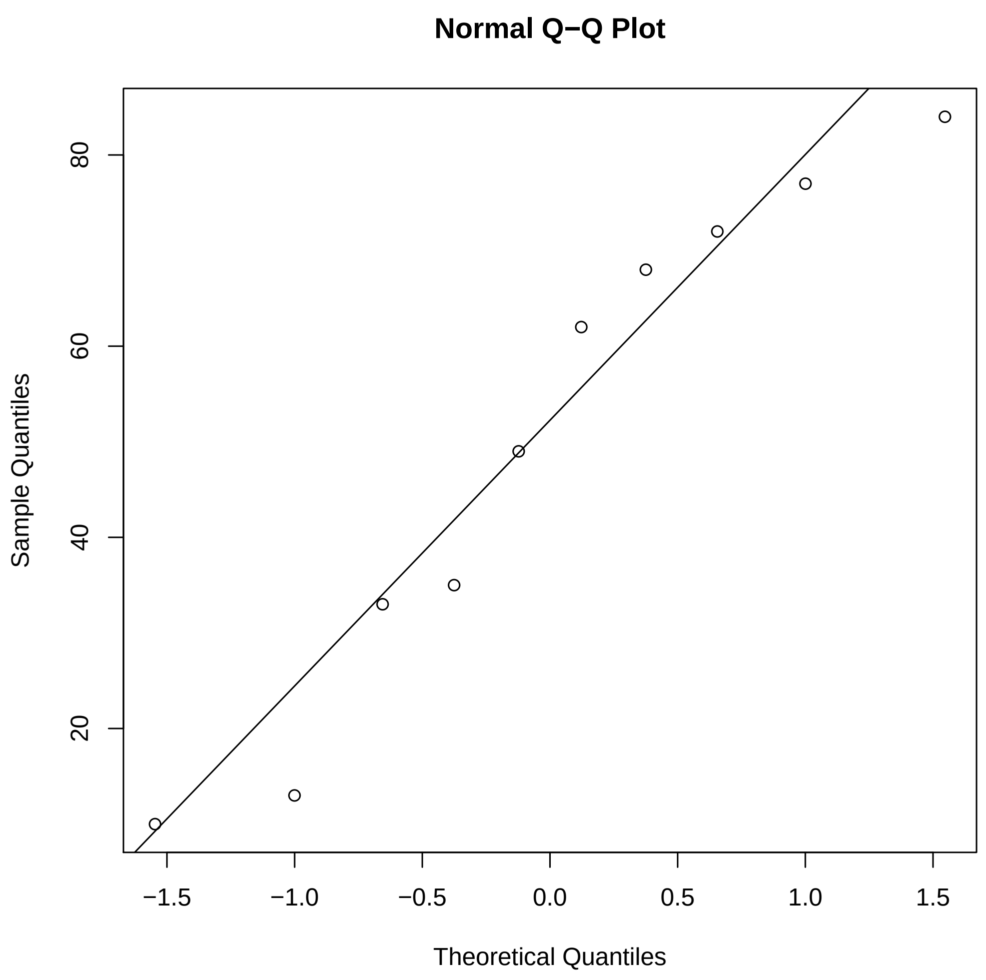

```
##
##  Shapiro-Wilk normality test
##
## data:  Task2
## W = 0.8317, p-value = 0.03511
```

Similar evidence that people took longer to change the bar height in Excel than in Sketchography.

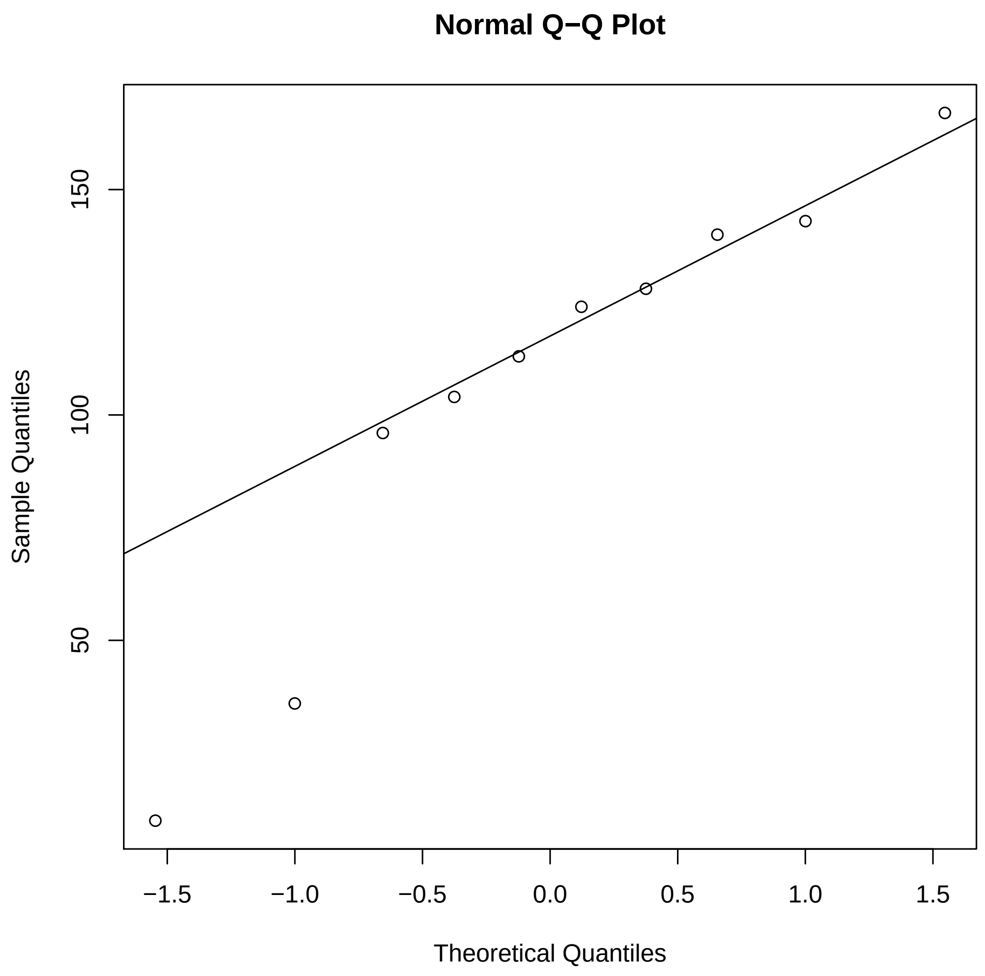
```
t.test(Sketch, Excel, paired = TRUE)

##
```

```
##  Paired t-test
##
## data:  Sketch and Excel
## t = -4.548, df = 9, p-value = 0.001389
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -83.55 -28.05
## sample estimates:
## mean of the differences
##                   -55.8
```



```
##
##  Shapiro-Wilk normality test
##
## data:  Sketch
## W = 0.9246, p-value = 0.3966
```




```
##
##  Shapiro-Wilk normality test
##
## data:  Excel
## W = 0.8943, p-value = 0.1896
```

Bibliography

Alvarado, C. and Davis, R. (2004). SketchREAD: a multi-domain sketch recognition engine. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, page 23–32, New York, NY, USA. ACM.

Blackwell, A. (2002). What is programming. In *14th workshop of the Psychology of Programming Interest Group*, page 204–218.

Blackwell, A. F., Church, L., Plimmer, B., and Gray, D. (2008). Formality in sketches and visual representation: some informal reflections. *Proceedings of the VL/HCC Workshoph, Herrsching am Ammersee, Germany*, page 11–18.

Bresciani, S., Blackwell, A., and Eppler, M. (2008). A collaborative dimensions framework: Understanding the mediating role of conceptual visualizations in collaborative knowledge work. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pages 364–364.

Chang, S. H.-H., Plimmer, B., and Blagojevic, R. (2010). Rata. ssr: Data mining for pertinent stroke recognizers. In *Proceedings of the Seventh Sketch-Based Interfaces and Modeling Symposium*, page 95–102. Eurographics Association.

Chao, W. O., Munzner, T., and van de Panne, M. (2010). Poster: Rapid pen-centric authoring of improvisational visualizations with napkinvis. *Posters Compendium InfoVis*.

Fonseca, M. J., Pimentel, C., and Jorge, J. A. (2002). CALI: an online scribble recognizer for calligraphic interfaces. In *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, page 51–58.

Frankish, C., Hull, R., and Morgan, P. (1995). Recognition accuracy and user acceptance of pen interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 503–510. ACM Press/Addison-Wesley Publishing Co.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

Hammond, T. and Davis, R. (2006). LADDER: a language to describe drawing, display, and editing in sketch recognition. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA. ACM.

Kara, L. B. (2004). An image-based trainable symbol recognizer for sketch-based interfaces. In *in AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural*, page 99–105. AAAI Press.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44.

Kuncheva, L. I. (2004). *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley and Sons, Inc.

Lee, B., Isenberg, P., Riche, N. H., and Carpendale, S. (2012). Beyond mouse and keyboard: Expanding design considerations for information visualization interactions. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2689–2698.

Norman, D. A. and Draper, S. W. (1986). *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Oltmans, M. (2007). *Envisioning Sketch Recognition: A Local Feature Based Approach to Recognizing Informal Sketches*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA. AAI0819449.

Ouyang, T. Y. and Davis, R. (2009). A visual approach to sketched symbol recognition. In *IJCAI*, volume 9, page 1463–1468.

Patel, R., Plimmer, B., Grundy, J., and Ihaka, R. (2007). Ink features for diagram recognition. In *Proceedings of the 4th Eurographics workshop on Sketch-based interfaces and modeling*, page 131–138. ACM.

Paulson, B. and Hammond, T. (2008). PaleoSketch: accurate primitive sketch recognition and beautification. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, page 1–10, New York, NY, USA. ACM.

Plimmer, B., Purchase, H. C., and Yang, H. Y. (2010). SketchNode: intelligent sketching support and formal diagramming. In *Proceedings of the 22nd Conference of the Computer-Human Interaction Special Interest Group of Australia on Computer-Human Interaction*, page 136–143. ACM.

Rubine, D. (1991). Specifying gestures by example. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, page 329–337, New York, NY, USA. ACM.

Shilman, M., Pasula, H., Russell, S., and Newton, R. (2002). Statistical visual language models for ink parsing. In *In AAAI Spring Symposium on Sketch Understanding*, page 126–132. AAAI Press.

Shilman, M., Viola, P., and Chellapilla, K. (2004). Recognition and grouping of handwritten text in diagrams and equations. In *Frontiers in Handwriting Recognition, 2004. IWFHR-9 2004. Ninth International Workshop on*, page 569–574. IEEE.

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69.

Sutherland, I. E. (1964). Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, page 6.329–6.346, New York, NY, USA. ACM.

Tanimoto, S. (2013). A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34.

Walny, J., Lee, B., Johns, P., Riche, N. H., and Carpendale, S. (2012). Understanding pen and touch interaction for data exploration on interactive whiteboards. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2779–2788.

Wang, M., Plimmer, B., Schmieder, P., Stapleton, G., Rodgers, P., and Delaney, A. (2011). SketchSet: creating euler diagrams using pen or mouse. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, page 75–82. IEEE.

Willems, D., Niels, R., van Gerven, M., and Vuurpijl, L. (2009). Iconic and multi-stroke gesture recognition. *Pattern Recogn.*, 42(12):3303–3312.

Wobbrock, J. O., Wilson, A. D., and Li, Y. (2007). Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, page 159–168, New York, NY, USA. ACM.

Yeung, L., Plimmer, B., Lobb, B., and Elliffe, D. (2008). Effect of fidelity in diagram presentation. In *Proceedings of the 22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction-Volume 1*, page 35–44. British Computer Society.

Yu, B. and Cai, S. (2003). A domain-independent system for sketch recognition. In *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '03, page 141–146, New York, NY, USA. ACM.