ECONSTOR
Make Your Publications Visible.

A Service of

ZBW
Leibniz-Informationszentrum Wirtschaft
Leibniz Information Centre for Economics

Seithe, Mirko

Working Paper

Introducing the Bonn Experiment System (BoXS)

Bonn Econ Discussion Papers, No. 01/2012

Provided in Cooperation with:
Bonn Graduate School of Economics (BGSE), University of Bonn

Suggested Citation: Seithe, Mirko (2012) : Introducing the Bonn Experiment System (BoXS), Bonn Econ Discussion Papers, No. 01/2012, University of Bonn, Bonn Graduate School of Economics (BGSE), Bonn

This Version is available at:
https://hdl.handle.net/10419/74638

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.

WWW.ECONSTOR.EU

Mitglied der Leibniz-Gemeinschaft

Bonn Econon Discussion Papers

Discussion Paper 01/2012

Introducing the Bonn Experiment System (BoXS)

by

Mirko Seithe

February 2012

Bonn Graduate School of Economics

Bonn Graduate School of Economics
Department of Economics
University of Bonn
Kaiserstrasse 1
D-53113 Bonn

Financial support by the
Deutsche Forschungsgemeinschaft (DFG)
through the
Bonn Graduate School of Economics (BGSE)
is gratefully acknowledged.

Deutsche Post World Net is a sponsor of the BGSE.

Introducing the Bonn Experiment System (BoXS)

Mirko Seithe*

February 5, 2012

Abstract

Computerised experiments play a vital part in the modern economic and social sciences. As the technology advances, more complex and sophisticated experiments become feasible. Fast internet connections are widely available today and mobile devices have become capable of running complex graphical applications. The Bonn Experiment System (BoXS) was designed to provide a platform for designing and conducting computerised experiments which is both easy to approach and use, as well as flexible in its possible applications. It does not require installation and uses a lean client which runs on Windows, MacOS and Linux and only requires a web browser and a Java Runtime Environment to execute.

This paper intends to highlight the main features and limitations of the BoXS and provide some guidance for experimenters getting started with it. It also discusses some of the design decisions and provides in-depth information on technical aspects of the system. The appendix includes both the documentation and full code examples.

JEL-Classification: C88, C99

Keywords: Experiment System; Software; Experiment Conduction; Java

*I would like to thank the participants of the Bonn Experiment Workshop and the users of the Bonn Experiment System for their helpful comments and suggestions, as well as the DFG for financial support. *Contact details*: Mirko Seithe, Schulstr. 15a, D-53757 Sankt Augustin, Germany, mseithe@uni-bonn.de, http://boxs.uni-bonn.de

Contents

1 Introduction **6**

1.1 Related Work 6

1.2 Introducing the Bonn Experiment System 8

1.3 Outline 10

2 Using the BoXS **11**

2.1 Quick Start Tutorial 11

2.2 Starting an Experiment 14

2.3 The Experimenter View 15

2.4 Internet Experiments 17

2.5 Laboratory Experiments 17

2.6 Using an Offline Server 18

2.7 Autorun Experiments 19

2.8 Troubleshooting 19

2.9 Documentation 20

3 The BoXS Programming Language **21**

3.1 Code Based and Graphical Approaches 21

3.2 Program Execution 22

3.2.1 Lexing and Parsing 22

3.2.2 Internal Implementation 23

3.2.3 Error Handling 24

3.3 Implemented Functionality 25

3.4 Basic Calculus 26

3.5 Variables 27

3.5.1 Internal Data Representation 27

3.5.2 Local, Group and Global Variables 27

3.5.3 Arrays and Matrices 28

3.5.4 Automatically Generated Variables 29

3.6 Displaying Information and User Input 30
3.6.1 Displaying Text and Graphics 30
3.6.2 Videos 32
3.6.3 Subject Input 33
3.6.4 Waiting and Assertions 33
3.7 Matching 34

4 Design and Implementation 35

4.1 Programming Language 35
4.2 Network Architecture 37
4.3 Communication Protocol 38
4.4 The Server 39
4.4.1 Server Robustness 39
4.4.2 Connection Robustness 40
4.4.3 Security 41
4.4.4 Notes on the Implementation 42
4.5 The Client 43
4.5.1 Implementation as Java Applet 43
4.5.2 Internal Implementation 46

5 Limitations and Future Development 46

5.1 Feature Selection 47
5.2 Future Development 47
5.3 Limitations 48

6 Conclusion 48

A List of all Functions in the BoXS Programming Language 50

A.1 Basic Operations and Calculations 50
A.1.1 Basic Calculus 50
A.1.2 More Calculus and Trigonometric Functions 50
A.1.3 Boolean Algebra 50

A.1.4 Random Number Generation 51
A.2 Program Flow Control 51
A.2.1 if(expression) { ... } 51
A.2.2 while(expression) { ... } 52
A.2.3 for(initialization; condition; iteration) { ... } 52
A.3 Displaying Text and Graphics 53
A.3.1 display([message]) 53
A.4 Waiting 54
A.4.1 wait([message],[messageafterclick]) 54
A.4.2 waitForPlayers([message],[messageafterclick]) 55
A.4.3 waitTime(time) 55
A.4.4 waitForExperimenter() 55
A.5 User Input 56
A.5.1 inputString(variablename) 56
A.5.2 inputNumber(variablename) 56
A.5.3 choice(varname,values) 57
A.5.4 checkbox(varname,description) 57
A.5.5 assert(expression) 58
A.5.6 style(text) 59
A.5.7 manualLayout() 60
A.5.8 Non-compulsory Input 61
A.5.9 Default values 61
A.6 Matching 61
A.6.1 matchAll(roles) 61
A.6.2 matchPerfectStranger(roles) 62
A.6.3 matchStranger(roles) 63
A.6.4 matchManual(username,group,role) 63
A.6.5 matchDone() 64

B Example BoXS Programs **65**

B.1 Questionnaire 65
B.2 Public Good Game 65
B.3 Chat Client 66
B.4 Dutch Auction 67
B.5 Localization 68
B.6 Real Effort Task 68

1 Introduction

Behavioural experiments have become a vital part of economic research in the preceding decade as they allow researchers to study actual human behaviour beyond the predictions of theoretical models. Most major economics departments now run dedicated laboratories which centralise the recruitment of experiment subjects and simplify the experiment conduction.

Nowadays most experiments are conducted using computers instead of pen-and-pencil methods, which brings both theoretical and practical advantages for experimenters: 1) Computers allow for experiments involving complex real-time interaction between subjects, e.g. in market or auction related experiments, which would be extremely tedious to conduct otherwise. 2) Using computers to interact with the subjects reduces possible experimenter effects and makes reproducing an experiment easier. 3) An experiment which was programmed once can be easily documented and shared amongst researchers. 4) The data generated by an experiment can be automatically collected and exported to spreadsheet and statistics programs.

In the course of the last decade, computer technology has vastly improved, affecting both the abilities of modern computers as well as their possible applications. Fast and stable internet connections are widely spread among both institutional and private users and computers have become able to display high quality audio and video files. At the same time the acceptance of computers has increased with most users as has their sophistication in using them.

1.1 Related Work

In the very beginning of computerised experiments no experiment software existed which would help experimenters design and run their experiments. This required every experimenter to implement her experiment from scratch using complex programming languages like C++, which in turn required the experimenter to either acquire significant programming skills or delegate the imple-

mentation to a professional programmer. While many experiment designs are easy to explain, they may be very hard to implement. Especially the programming of network communication and the graphical user interface can be very complicated and tedious and often outweighs the advantages of using computers in the first place.

The first major improvement on this situation came in the form of RatImage, developed by **?**, which is a library of common functions required for most experiments, for example user interface design. While RatImage still required the experimenter to program his experiment in low-level programming languages, many tasks could be vastly simplified by using its predefined routines. Unfortunately, RatImage, which was designed for the outdated MS-DOS operating system, seems to be neither supported nor available any more.

The next major improvement was z-Tree, which was introduced by **?**[1] and has been steadily supported and improved ever since. Based on the citation count it is probably the most relevant experiment software to date, especially for economists. The main feature of z-Tree is that it allows the experimenter to design many experiments without writing any program code. It provides an extensive graphical user interface which makes all the important functions accessible and allows the user to design experiments by arranging basic components like text fields and buttons on a tree-like structure. By providing this simplified approach z-Tree allows experimenters with no prior programming experience to implement and run an experiment, while at the same time providing a feature set extensive enough to allow for the implementation of most experiment types. Z-Tree is designed for the Microsoft Windows operating system and provides both server (zTree) and a client (zLeaf) applications which communicate using the TCP-IP protocol. The most recent version of z-Tree implements graphics, both for presentation and interaction, the support for external hardware and chat functionality.

Regate, designed by **?**, is another experiment software system for Win-

[1] See also **?**.

dows which enables experimenters to program and conduct computerised experiments[2]. It provides an elaborate and complex user interface which experimenters can use to program and supervise their experiments. Programs consist of several script statements which are inserted in a tree structure. Debugging and testing are simplified in Regate by a) enabling the experimenter to play several subjects on the same computer and screen at the same time and b) providing the possibility to simulate subjects' behaviour by having the software make random choices in a specified range. Regate includes an online documentation and provides several sample programs.

Finally, **?** provides a good overview on how internet experiments can be implemented. He explains both how to use existing experiment software like z-Tree and RatImage in an internet environment as well as the more basic programming approach based on HTML and PHP.

All the mentioned platforms have weaknesses. First, they are designed for laboratory experiments only and are often not designed for mobile or internet experiments. They also heavily rely on Windows as their only supported platform and cannot be used on devices like mobile phones. Second, they are often not very user friendly, not very easy to learn and often lack a comprehensive and up-to-date documentation. Finally, many of the mentioned platforms are no longer supported and often cannot be used on recent computers.

1.2 Introducing the Bonn Experiment System

This chapter introduces the Bonn Experiment System (BoXS), which provides a platform for experimenters which is both flexible and easy to use.

The flexibility of the Bonn Experiment System arises from two facts. First, the system is based on the Java platform, which allows it to be used on a wide variety of platforms, including both different device types like netbooks and mobile devices as well as different operating systems like Windows, Linux and

[2]Since no published paper on Regate is available yet, this paragraph is based on the presentation and the manual available at the official homepage: `http://www.gate.cnrs.fr/~zeiliger/regate/regate.htm`

```
1  display("<h1>Questionnaire</h1><hr>")
2
3  display("<br>Please enter your name:")
4  inputString(name)
5
6  display("<br>Please enter your field of study:")
7  inputString(study)
8
9  display("<br>Please enter your age:")
10 inputNumber(age)
11 assert(age>10 && age<100 && age==round(age))
12
13 display("<br>Please select your gender:")
14 choice(gender,"female","male")
15
16 wait()
```

(a) Program code.

(b) Resulting screen.

Figure 1: A simple questionnaire in the Bonn Experiment System.

MacOS. Second, while it is still possible to download and use the BoXS in an offline environment, it can use the internet as a medium to connect the computers participating in an experiment, which enables any computer worldwide to participate in an experiment without requiring the experimenters to set up their own network structure. This allows for a variety of experiment environments:

- Laboratory experiments, both using an official server (which is easier to use) or an offline server (which allows for experiments without an internet connection).
- Internet experiments in which subjects participate using their private computers at home.
- International experiments where subjects from different countries participate using computers connected over the internet.
- Mobile experiments using netbooks, laptops or Java-compatible mobile phones connected over wireless internet.
- Cross-platform experiments involving Windows, Linux and MacOS.

The Bonn Experiment System also introduces useful features like the simple measurement of response times and the tracking of a input history for each variable, which may be interesting for researchers interested in choice revision behaviour or the individual decision process.

Besides being flexible, the BoXS is also very robust. When a subject's computer or even the experimenter's computer crashes, the experiment continues and the affected subjects/experimenter can simply reconnect and resume the experiment at the point before the crash while all previous data is preserved.

The BoXS is easy to use for several reasons. The programming language implemented in the BoXS is designed to be compact, easy to learn and intuitive to use and resembles popular programming languages like Java. The BoXS also features extensive documentation including an online manual, example programs, a tutorial, a site answering frequent questions, a discussion group where questions can be posted and, coming soon, video tutorials for the most common questions. The user feedback from experimenters writing their first experiments using the BoXS has so far been very positive. Furthermore, the BoXS does not require any installation on a computer. This makes setting up even complex experiment environments easy as inviting someone to participate in an experiment only requires sending a link. Testing and debugging is also easy as the BoXS allows the easy simulation of a large number of subjects.

1.3 Outline

Section 2 of this chapter is intended for experimenters who have not used the BoXS before and want to learn about its features. It starts with a brief tutorial and provides information on how to use the BoXS in different environments.

Section 3 describes the BoXS Programming Language (BoXSPL), which is introduced by the BoXS and is intended to provide a simple way for non-programmers to design experiments. The section describes how programs are executed and how the most important commands work.

Section 4 provides a more in-depth technical description of the underlying network architecture and communication, as well as on how the server and client software is realised. It is primarily intended for readers with a computer science background who are interested in how the Bonn Experiment System works.

The last two sections discuss the current state and the possible future de-

velopment of the BoXS. Finally, a full documentation of the BoXSPL as well as several example programs are provided in the appendix.

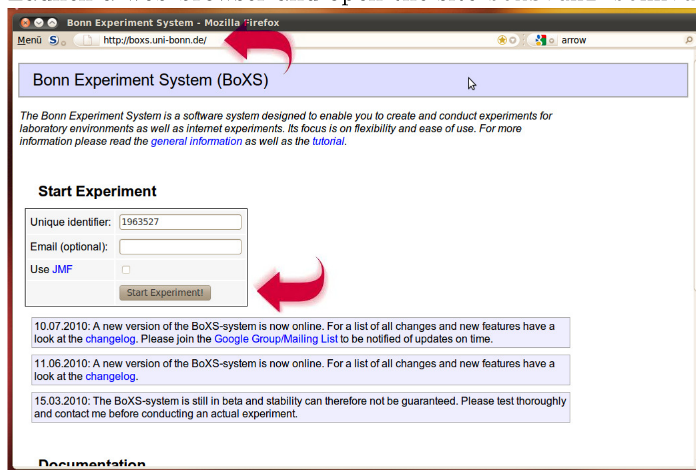
2 Using the BoXS

This section provides practical tips for experimenters considering to use the Bonn Experiment System (BoXS). It begins by providing a brief tutorial which demonstrates how to write a simple experiment, proceeds with a description of the user interface and explains how to use the BoXS in laboratory and internet experiments.

2.1 Quick Start Tutorial

This tutorial explains how to write the quintessential "Hello World"-program in less than 5 minutes. For this tutorial to work an internet connection, an internet browser and the Java plug-in for the browser are required.

1. Launch a web browser and open the site `boxs.uni-bonn.de`.



2. Click on "Start Experiment!".

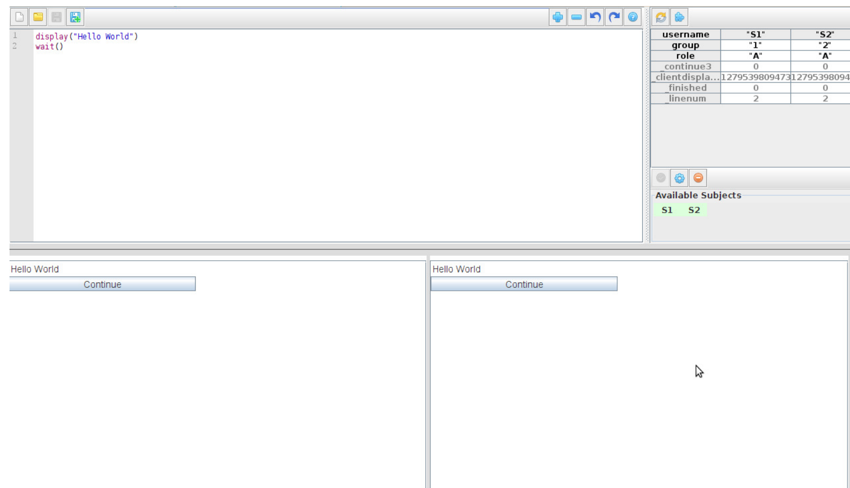
3. Click on "1 Experimenter, 2 Subjects".

4. When asked for a password, just click on "Ok".

5. The top half of the screen contains the experimenter view. The bottom part contains two subject views for testing purposes. Click on the large white area in the top and enter the following program:

```
display("Hello World")
wait()
```

6. When done, click on the green start-icon on the right.



The Hello World program is successfully compiled by the server and executed on the two simulated subject clients in the bottom. When you click on "Continue" in the subject views the experiment ends and you can write and start a new experiment. Feel free to experiment by editing and expanding the example program.

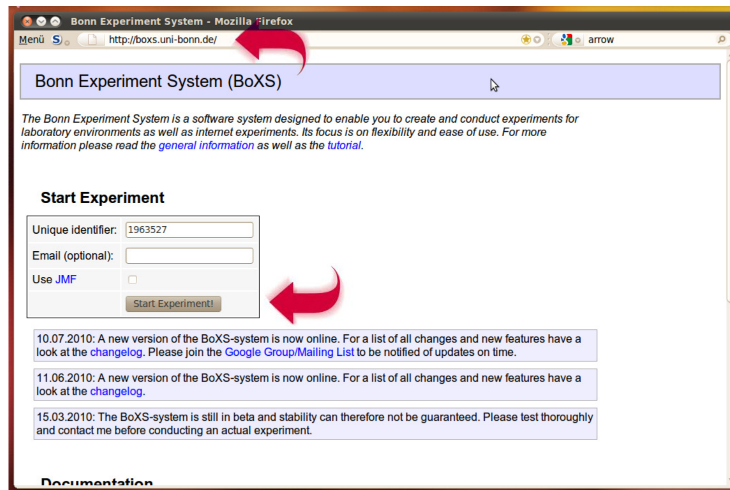


Figure 2: The starting page.

2.2 Starting an Experiment

Figure 2 shows the BoXS web site which is typically used to the start an experiment. At the beginning, the experimenter has to specify a realm id and, if required, her email address. In the BoXS, each experiment is uniquely identified by its realm id which ensures that your subjects do not get mixed up with other experiments. By default the realm id is a generated random number which is sufficient for most cases. Alternatively, it can be set to the experimenter's name, institution or her experiment's name. Specifying an email address enables the BoXS to automatically send results to the experimenter's email account. Note that this is completely optional as data can be exported without using this mail option.

Upon clicking on 'Start Experiment!', the 'Available Setups'-page shown in figure 3 is displayed. This page offers a large number of possible display set-ups for the experiment, which each include an experimenter view and/or one or more subject views. The quick start tutorial uses one experimenter view and two subject views on the same page, which is useful for testing purposes. Other available set-ups include pure experimenter or subject views, which are useful for actual experiments, as well as pages with up to 16 subjects each, which are

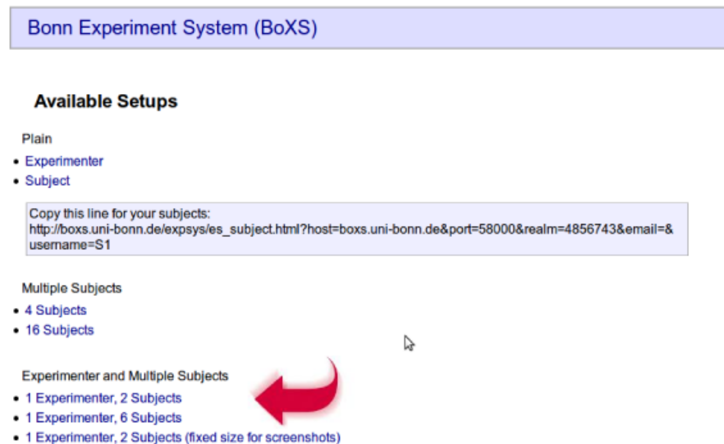


Figure 3: The 'Available Setups' page.

intended for testing and debugging. If more than 16 subject views are required, the respective page can be opened multiple times. Every set-up opened from this page is automatically associated with the created experiment and shows up on the corresponding experimenter's screen.

Note that while an arbitrary number of subjects can be active at the same time, only one experimenter can. When another experimenter client is started, all previously connected experimenter clients for the respective realm are disconnected automatically.

2.3 The Experimenter View

Figure 4 shows the main experimenter view which allows her to write programs as well as start, supervise and cancel experiments.

When the experimenter view is first displayed, the experimenter is asked for a password. By default each new realm is created without a password. In order to specify a password it can be entered at this point and experimenters are required to enter it whenever they reconnect to this realm in the future. Setting a password is *strongly recommended* as ill-meaning and well-informed subjects could specify it otherwise and prevent the legitimate experimenter from

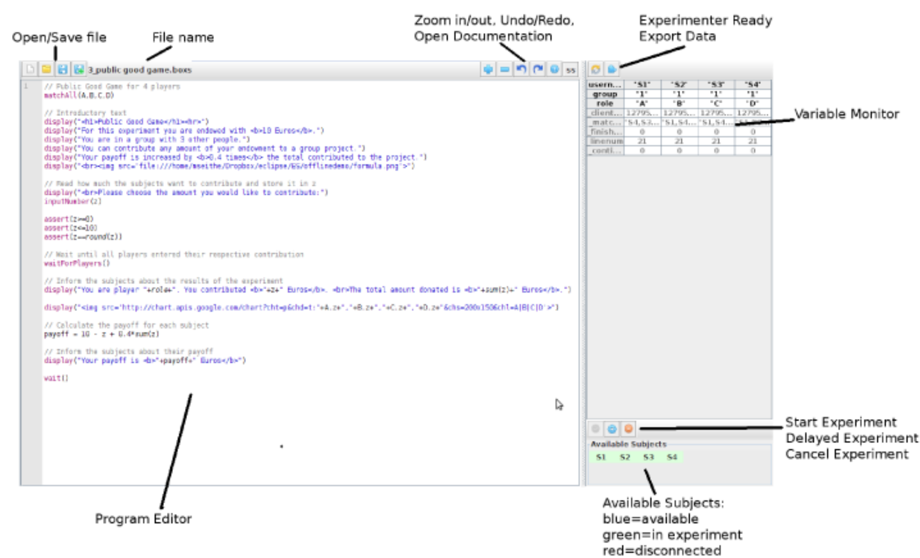


Figure 4: The experimenter view.

accessing her experiment.

The main component of the experimenter view is the program editor in the left part into which experimenters can enter their programs. The program editor allows copy and paste as well as undo and redo. It also provides automatic syntax highlighting, which enhances the readability of the program code. The buttons on top of the program editor allow the experimenter to load and save programs and to change the font size of the editor.

On the bottom right of the client view a list of all the subjects who are connected to the experiment is displayed. This list is updated whenever new subjects join the experiment or the status of a subject changes. Subjects who are available for an experiment are displayed in green, subjects in an experiment are displayed as blue and subjects who were disconnected during an experiment and suspended are shown in red. On top of the subject list buttons to start a normal experiment, to start a delayed experiment and to cancel an experiment are provided.

The variable list in the top right area displays all data generated by the currently running experiment in a table with each subject in a separate column.

The data displayed in this table can be exported to a comma-separated values file (CSV) by clicking on the export button on top of it.

The separation between the program editor and the variable view can be dragged by the experimenter to suit her layout preferences.

2.4 Internet Experiments

In the tutorial both the experimenter and the subject clients are executed on the same computer. Conducting a real experiment with other people is relatively straightforward. In order to conduct an internet experiment, one can copy the subject link displayed on the 'Available Setups'-page and send it to the desired subjects. For example:

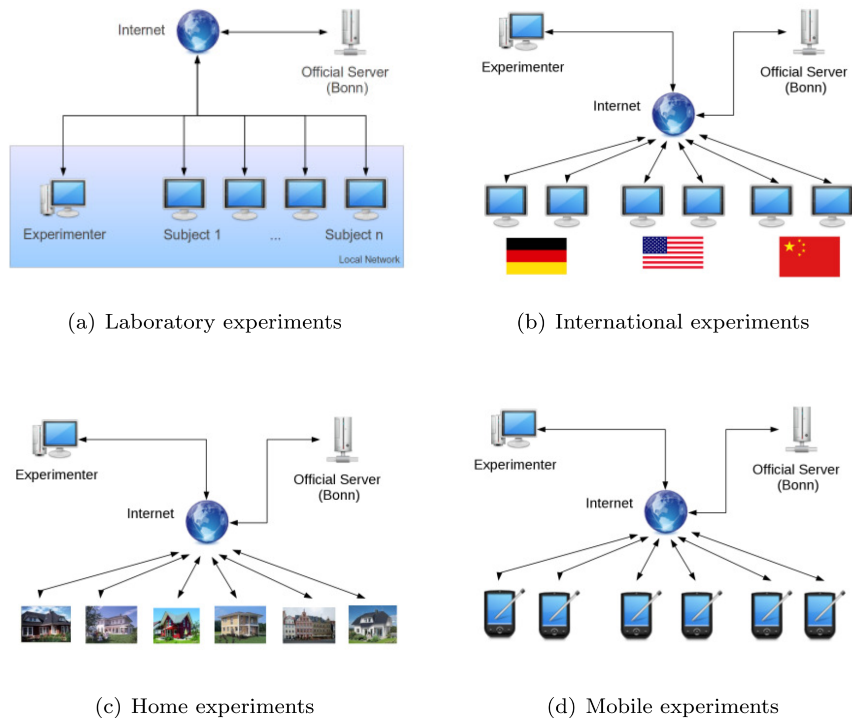
`http://boxs.uni-bonn.de/expsys/es_subject.html?host=boxs.uni-bonn.de&port=58000&realm=1963527&email=&username=new`

The link contains the realm id and the server data required for participating in the experiment. If another person opens this link in her web browser, she shows up in the subject list with the user name specified in the link. The experiment can then be started by pressing the start-button in the experimenter's view as described in the tutorial. Information on starting the experiment automatically is provided in section 2.7.

2.5 Laboratory Experiments

Laboratory experiments using the BoXS work very similar to internet experiments. In the beginning the subject link copied from the 'Available Setups'-page has to be opened on each computer in the laboratory. The user name should be changed to reflect each computers' cubicle/room number in order to correctly identify the subjects and their computers later on. As copying this link can be quite tedious, it is generally a good idea to bookmark the link on every computer so that it can be reused for future experiments.

Like in the internet experiment example the subjects show up in the experimenter's available subjects list and the experiment can be started by clicking



(a) Laboratory experiments

(b) International experiments

(c) Home experiments

(d) Mobile experiments

Figure 5: Possible applications for the Bonn Experiment System.

on the start-button.

2.6 Using an Offline Server

Usually the official BoXS server is recommended for all experiments as it is the most convenient way. There are some cases, however, in which the set-up and use of a local BoXS server can be advantageous. Experiments for which no internet connection is available, for instance due to technical restrictions or restrictive laboratory policies, are a good example. Another kind of situation are high-frequency experiments where extremely fast reaction times and very low latencies are required.

The package required for running a local BoXS server can be downloaded from the general information section of the homepage which also includes some tips on how to set it up. In a nutshell, the experimenter needs to execute the downloaded BoXS server on one of the computers. Then the official server's

name (`boxs.uni-bonn.de`) on the participation links has to be substituted by the IP address of the computer running the server. Afterwards, everything should work like when using the official internet server with the exception of the email functionality.

2.7 Autorun Experiments

Some experiments, especially internet experiments, require to be run while the experimenter is not available. Consider the case in which the experimenter wants participants to fill out an online questionnaire during a certain time period. Doing this with the methods discussed previously would require the experimenter to sit in front of her computer and manually start an experiment whenever a participant connects to the BoXS.

In order to simplify this process so-called autorun experiments have been implemented. An autorun experiment is created by writing a program as usual and clicking the blue 'autorun'-button when done. The experiment is now stored on the server and the experimenter can turn off her computer without affecting it.

Whenever a subject with the appropriate realm id logs onto the server, the stored experiment is automatically executed. The data of the experiment, including all previous observations, is sent to the experimenter by email after each completed observation. Note that a valid email address *has to be specified* at the beginning of the experiment in this case.

2.8 Troubleshooting

The following two problems are encountered frequently when using the BoXS and can be solved easily:

- If nothing is displayed after clicking on a link on the 'Available Subjects'-page, the Java plug-in is probably not properly installed. The Java plug-in is available for free and most web browsers notify the user in case it is miss-

ing and aid her in its installation. Otherwise it can be installed manually by visiting the Java homepage at `www.java.com` and downloading and installing the Java Standard Edition Runtime Environment (JRE).

- If a message claiming that clients cannot connect to the server is displayed despite a working internet connection, the experimenter's institute's firewall is probably at fault. In order to resolve this, the corresponding IT department should be kindly requested to open the ports 58000 and 58001, which are used by the BoXS, for TCP connections.

2.9 Documentation

Several ways are available to learn more about how to use the Bonn Experiment System:

- The appendix of this chapter as well as the largely equivalent online documentation provide an elaborate documentation for each command available in the BoXSPL:
  `http://boxs.uni-bonn.de/documentation/index.html`
- The documented example programs, which are printed in the appendix of this chapter and can be downloaded on the web site provide examples for how the BoXS can be used and how common experiment types can be realised:
  `http://boxs.uni-bonn.de/examples/index.html`
- The frequently asked questions section on the web site contains a big list of answered questions and is a good place to start when problems and questions are encountered:
  `http://boxs.uni-bonn.de/general/index.html`
- A public mailing list exists where all users can ask questions and are invited to contribute to the general discussion:
  `http://groups.google.com/group/bonn-experiment-system`

- Video tutorials demonstrating the basic features of the BoXS are available on the homepage and demonstrate how to do the most common tasks using the BoXS.

3 The BoXS Programming Language

In this section I describe the thought process behind the design decisions met concerning the BoXS Programming Language (BoXSPL). The goal of the BoXSPL is to create a language which is easy to learn for novice users while still allowing the implementation of most experiment types. This section intends to provide an overview of the BoXSPL. For more information on the commands and concepts described in this section please refer to the appendix or the official homepage where more elaborate documentation is available.

3.1 Code Based and Graphical Approaches

While most professional programming languages like C++ and Java are purely text based programming languages, languages designed for novice programmers like z-Tree or Regate provide strong graphical user interfaces for designing a program. The advantage of graphical approaches is that they may be easier to learn and less intimidating for novice users as standard experiment types like questionnaires can often be created without even writing a single line of code. In more complex experiments, however, the experimenter is usually required to write program code at some point either way.

Text based languages provide advantages for advanced users as it is usually faster to type a desired command using the keyboard than to create it using a graphical interface. Sophisticated users may furthermore take advantage of features like copying and pasting and are free to choose any text editor they like. Another advantage of text based languages is that their programs can be easily shared and archived, as they are compatible across versions and platforms, or published, as they can be easily printed.

With the BoXSPL I introduce a text based programming language. In order to ease the learning curve for novice users I provide a rich documentation, several sample programs and an editor with syntax highlighting. I also provide a tutorials and videos to reduce the time and effort required to get new users started with the BoXS and create a first experiment.

3.2 Program Execution

This section describes how the BoXS server processes a program written by an experimenter and how it is executed.

3.2.1 Lexing and Parsing

In computer science, a lexical analyser (lexer) is an algorithm which reads a given text string and translates it into a set of tokens, for example string tokens, numbers and operators[3]. These tokens are then handed over to a syntactic analyser (parser), which analyses and structures the tokens and, as a final step, arranges and translates them to a format which can be executed[4].

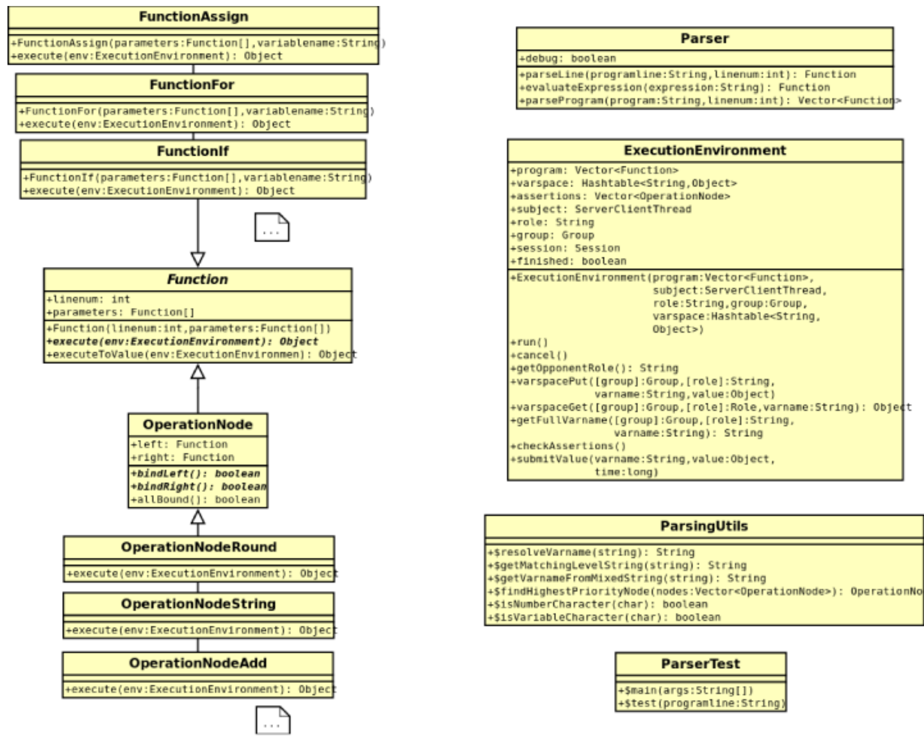
When the BoXS project was initiated as a small prototype, a hand-made simple lexing/parsing-algorithm was implemented. As the complexity of the language increased and more test cases were created, the stability, quality and performance of the lexing/parsing process has been steadily enhanced and improved. An alternative to hand-made lexers are so-called lexer- and parser-generators for Java, for example JLex and CUP[5], which are freely available. These generators process a given language specification, which can be enhanced and changed later on, and create lexer- and parser-code which can be included in any program.

The main advantage of using a such a professional lexer/parser generator is the high reliability and robustness of the resulting algorithm. Furthermore

[3]See Wikipedia, `http://en.wikipedia.org/w/index.php?title=Lexical_analysis&oldid=366935008`

[4]See Wikipedia, `http://en.wikipedia.org/w/index.php?title=Parsing&oldid=373059757`

[5]See `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.



Note: Some attributes, operations and classes are omitted from the diagram in order to improve readability.

Figure 6: Class diagram of the lexing/parsing classes.

it simplifies the future documentation and enhancement of the underlying language specification. In the long run the transition to a lexing/parser generator seems advantageous. However, as the required changes would likely incur a lot of initial instability as the lexing/parsing process is vital for the BoXS, the migration process has a relatively low priority at the moment.

3.2.2 Internal Implementation

Lexing, parsing and execution are done in several steps. An overview of the classes involved in the process is shown in figure 6. First, the complete program is partitioned based on the matching and flow control commands (if, for and while) contained in it. In the next step, the resulting code is scanned line by line and translated into corresponding Function-objects. These objects are then filled with the data required for their proper execution at runtime.

For instance, the line `var=round(15/4+6)` is translated into the following tree-like structure:

```
FunctionAssign ("var", OperationNodeRound (OperationNodeAdd
  (OperationNodeDivide (OperationNodeDouble:15.0,
   OperationNodeDouble:4.0),OperationNodeDouble:6.0)))
```

The FunctionAssign-object, which is on top of the hierarchy, assigns a value to the variable named `var`. In order to calculate the correct value for this it executes the OperationNodeRound-object which in turn executes and evaluates objects further down in the hierarchy.

As a result of the lexing/parsing process, a program which is entered as a text string is converted into a vector of Function-objects, which can each reference one or more related Function-objects. In the first versions of the BoXS this conversion, which is arguably the most computationally intensive and complex process in the BoXS, was done while the experiment was running. In order to improve performance this process is now done before the experiment is executed, which vastly improves the execution performance in more complex experiments. At runtime, the BoXS calls the `execute`-methods of all Function-objects, which are implemented as very fast and efficient operations.

In order to ensure the proper functioning of the BoXS lexer/parser a suit of critical test terms and expressions has been collected which is executed and tested before a change is incorporated into the official BoXS server. Every time an internal error in the BoXS is found, a corresponding expression is added to this test collection in order to ensure that this error is not accidentally reintroduced in a future version.

3.2.3 Error Handling

Unfortunately not all programs written by experimenters are flawless. There are two categories of errors which can occur when executing a user-written program. The first category contains so-called compile-time errors which prevent

the program from being lexed and parsed correctly, for instance misspelled commands, missing brackets or other types of syntax errors. The second category consists of so-called runtime errors which occur and can only be detected while the program is executed, for example the referencing of undefined variables or an invalid mathematical operation.

Both compile-time and runtime errors which occur when running a program in the BoXS are reported to the experimenter and displayed in a separate window, including the line which caused the error. Error messages serve the purpose of informing the experimenter about mistakes in her program and making her aware of possible implications.

When compile-time errors are encountered, the BoXS only shows the error message and does not start the experiment. When runtime-errors occur, the philosophy of the BoXS is to keep the experiment running whenever possible and only halt the execution for subjects who are directly affected by the error.

Furthermore two specific types of possible errors do not raise error messages: 1) Referencing an undefined variable does not result in an error but returns the numerical value 0. The rationale for this is that it makes programs significantly shorter by eliminating the need to initialise every variable (for example `counter=0`). 2) Some questionable mathematical expressions, for instance `var=1/0`, does not result in an error message but in the pseudo-value `Infty` (infinity), which may produce odd results when used for further calculations.

3.3 Implemented Functionality

One important process in creating a programming language is to find the right compromise between its accessibility and its generality. While a simple language with only a few commands might be very appealing to novice users, a lack of functionality would narrow down its possible applications.

Before a description of the functions implemented in the BoXSPL is provided, consider the questionnaire example program shown in figure 7 for an impression of how a typical BoXS-program looks. A typical program includes

```
display("Please enter your age:")
inputNumber(age)
assert(age>=10 && age<=100)
display("Please enter your gender:")
choice(gender,"male","female")
wait()
```

Figure 7: Example questionnaire.

Displaying on the subjects' screens display, video
Requesting input from the subjects inputString, inputNumber, choice, assert
Basic algebra + - / * % < > >= <= == != !
Mathematical functions log exp round sin cos tan
Random number generation randomUniform, randomUniformInteger, randomGauss
Controlling the program flow if, while, for
Waiting for the subjects wait, waitForPlayers, waitForExperimenter, waitTime
Matching subjects into groups matchManual, matchAll, matchStranger, matchPerfectStranger

Figure 8: List of all functions implemented in the BoXS.

`display`-commands to display instructions and questions, includes some input commands like `inputNumber` and `choice` and ends with a `wait`-command.

Figure 8 shows the set of functions which are implemented in the first version of the BoXS Programming Language (BoXSPL) grouped by function. The functions allow for most experiment types and questionnaires. Each function is designed to have a clear purpose and be easy to understand.

3.4 Basic Calculus

On the most basic level the BoXSPL includes the most common mathematical functions as well as string concatenation. It can evaluate arithmetic expressions, calculate with integer and real numbers at double precision and understands the use of brackets. Furthermore the BoXS can generate uniformly and normally

distributed pseudo-random numbers based on the linear congruential generator implemented in Java[6]. It also provides program flow control commands in the form of an `if`-command for conditional execution as well as a `for`- and a `while`-command for repeated execution.

3.5 Variables

The BoXS provides a very flexible data structure which allows for variables with different scopes, i.e. local, group-specific and global types, as well as arrays and matrices of arbitrary dimensions.

3.5.1 Internal Data Representation

The BoXS uses a so-called HashMap-object to store all data generated by each experiment as it provides a very flexible way of data storage. A map in computer science is a general data structure which can store an arbitrary number of key-value pairs. The HashMap-class, as provided by the Java programming language, provides a very efficient implementation of such a map by generating hash codes for each key in order to reduce the time required to access stored data.

The keys used in this map are the variable names, which are stored as a string, and the corresponding values are arbitrary objects. In the current version these objects are either strings or double precision numbers. In future versions this might be used to store more complex objects like lists or images.

3.5.2 Local, Group and Global Variables

In order to ensure that all data is stored unambiguously, a variable name needs to be transformed and resolved internally before a variable is stored. The variable name `payoff`, for example, would be problematic as it would be unclear to which subject the payoff belongs. In order to avoid this, each variable name

[6]See `http://download.oracle.com/docs/cd/E17476_01/javase/1.4.2/docs/api/java/util/Random.html`.

(Suppose there is one group (1) with two subjects (S1 and S2) in roles A and B.)

Assignment for ...	Program Line	Internal Representation
... current subject	`var=5`	S1.var=5
... subject A in current group	`A.var=5`	S1.var=5
... subject B in current group	`B.var=5`	S2.var=5
... subject B in current group 1	`1.B.var=5`	S2.var=5
... all subjects in current group	`*.var=5`	S1.var=5, S2.var=5
... all subjects in group 1	`1.*.var=5`	S1.var=5, S2.var=5
... all subjects in all groups	`*.*.var=5`	S1.var=5, S2.var=5

Table 1: Local, group and global variable examples.

(Suppose there is one group (1) with two subjects (S1 and S2) in roles A and B.)

	Program Line	Internal Representation
specific index	var[3]=5	S1.var[3]=5
calculated index	var[1+2]=5	S1.var[3]=5
string index	var["A"]=5	S1.var[A]=5
string index	hello["german"]="Willkommen..."	S1.hello[german]=...
variable index	var[experimentround]=5	S1.var[3]=5
variable index	var[role]=5	S1.var[A]=5
3-dimensional	var[1][2][3]=5	S1.var[1][2][3]=5

Table 2: Array and matrix examples.

is internally prefixed by the respective subject's username , which is always unique[7] for each experiment[8].

If no specific prefix is specified by the experimenter, a variable is treated as a local variable which means that it only applies to the current subject. Therefore the line `payoff=5` only sets the current subject's payoff to 5. In order to change another players variables or to do group-specific or global[9] changes a prefix has to be used. This allows the experimenter to create pseudo-global variables and share variables among subjects. Table 1 shows some examples how this can be done in the BoXSPL.

3.5.3 Arrays and Matrices

The BoXSPL allows arrays and matrices of arbitrary dimension to be stored. Table 2 shows some examples of what can be done with this.

The reason why the BoXS is so flexible with respect to arrays is that they are

[7]The server always ensures that the usernames of the subjects are unique. If several subjects login with the same username, they are renamed internally by adding the suffix `_<number>`.

[8]In the first versions of the BoXS, variable names stored in the format of `group.role.varname`. This turned out to be problematic, however, as the re-matching of subjects would lead to all variables getting mixed up.

[9]Note that this so-called global level is specific to the current realm. Cross-realm communication is not possible for obvious security reasons.

not stored as arrays internally. They are stored in the very same HashMap where all data is stored. The array indices are evaluated at runtime and appended to the variable name. In effect, a one-dimensional array with a length of 5 is stored like 5 separate simple variables.

Arrays do not need to be defined ahead of time and their dimensions can be arbitrarily changed at runtime. Furthermore both number and string indices are allowed, which is very useful in some situations.

3.5.4 Automatically Generated Variables

The BoXS automatically creates several variables during the execution of an experiment. While some of these variables are only required for the internal execution process, some variables may be interesting for experimenters. Most automatically generated variables are prefixed with "_" in order to avoid confusion.

- `_linenum`: The number of the line in the program which is currently being executed (usually a wait-command).
- `_finished`: 1 if the experiment has finished for this subject, 0 otherwise.
- `_continue<linenumber>`: 1 if the subject has clicked successfully on the wait-button specified in the given line, 0 otherwise.
- `_clientdisplaytime<linenumber>`: The exact time[10] at which a stage was displayed on the respective subject's screen. This may be useful for experimenters in order to synchronise the BoXS to other devices based on the time. The line number specifies the wait-command which triggered the stage in question.
- `_inputhistory_<varname>`: This variable stores every input made by the subjects. This allows the experimenter to learn about the decision process

[10] In milliseconds since January 1, 1970, 00:00:00 GMT, see: `http://download.oracle.com/docs/cd/E17476_01/javase/1.4.2/docs/api/java/util/Date.html`.

and possible choice revisions, as well as the response times. The data is stored as a comma-separated string where each action is formatted as `<time>.<input>` and where `time` is the number of milliseconds since the current stage was displayed on the subject's screen and `input` is the value entered by the subject at that time. This feature can be disabled using the `disableInputHistory()`-command if the data is not required.

3.6 Displaying Information and User Input

An experiment software needs to enable the experimenter to both present instructions or questions on the subjects' screens as well as receive their input. In the BoXSPL several commands are available to achieve this.

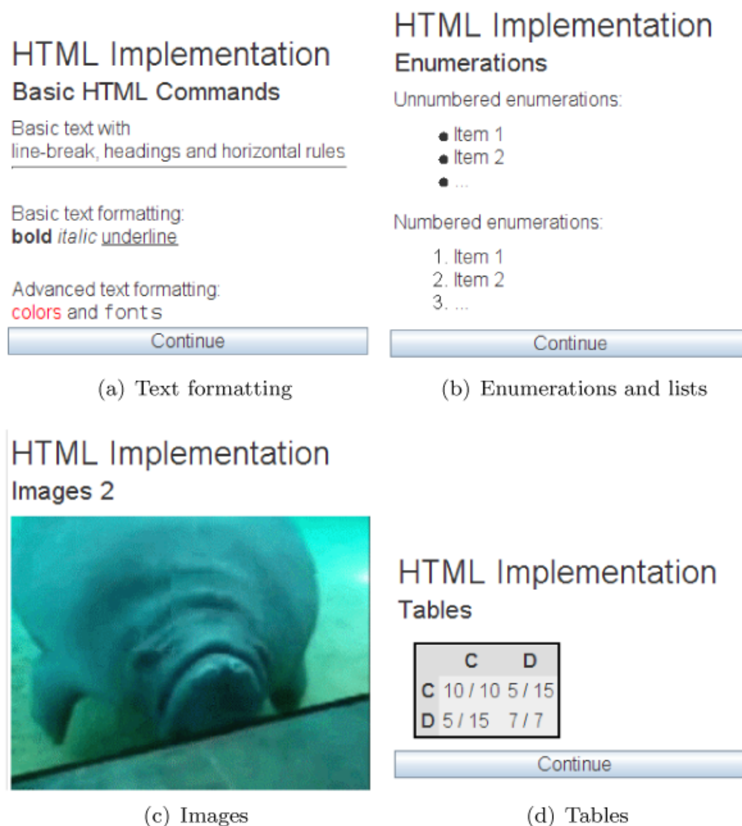
Each command is processed on the server by evaluating variables and solving calculations and distributed to the clients where it triggers the creation of a corresponding graphical components like a text boxes or a buttons. After creating all components for a screen they are, by default, vertically aligned and displayed. In general the BoXS client tries to recycle components and realise each stage with as few changes as required in order to increase the performance and reduce possible flicker effects in experiments where the information to be displayed changes frequently, for example in market experiments.

In case the components do not fit the screen's height, a vertical scrollbar is displayed, which allows the subjects to view components which do not fit on the screen. Horizontal scrollbars are not shown.

3.6.1 Displaying Text and Graphics

For displaying instructions, graphics and other types of data the BoXSPL provides the `display`-command. The `display`-command, as well as several other commands, supports the Hyper-Text Mark-up Language (HTML) and provides a great amount of flexibility.

The HTML format, which is also used for website programming, is both popular, flexible and relatively easy to learn. Besides simple text, HTML allows



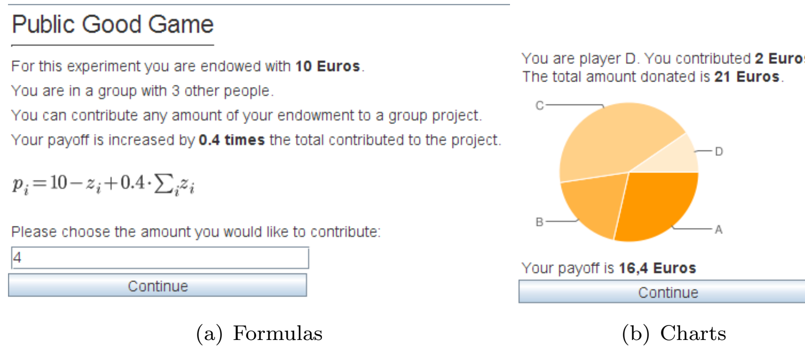
(a) Text formatting (b) Enumerations and lists

(c) Images (d) Tables

Figure 9: Examples using the `display`-command.

for text formatting, lists, enumerations, tables and images. The image formats which can be used include the standard formats JPEG and PNG as well as animated GIFs which can be used for displaying moving images on the subject's screen. Figure 9 shows some examples of what can be achieved by using HTML formatting.

By default, a modern style using sans-serif fonts and a compact layout is used in the BoXS, which is likely sufficient for most experimenters. If required, more advanced experimenters can use advanced Cascading Style Sheet (CSS) formatting to further customize the BoXS's appearance. CSS code can either be used within a `display`-command, which changes the appearance for this exact element, or globally by using the `style`-command, which is not discussed in this document.



(a) Formulas (b) Charts

Figure 10: More examples using the `display`-command.

Since the BoXS is usually run over the internet, several kinds of services which are available on the internet can be used within an experiment to gain access to additional functionality. In the example program shown in figure 10 this is used to include a mathematical formula, which is generated from TEX-code using a Google service, as well as a chart based on data generated in an experiment and visualised using the Google Chart API[11].

3.6.2 Videos

The BoXSPL also includes an experimental `video`-command which can be used to include video and audio files into an experiment. In order to use it the experimenter needs to provide a video or audio file in a format that is compatible to the Java Media Framework (JMF)[12]. Using the JMF has the advantage of true platform-compatibility but unfortunately also introduces some restrictions.

First, using the `video`-command requires the subject clients to be able to access the Java Media Framework, which has to be specified when starting the clients and can slow down the starting process significantly. Second, the video and audio codecs supported by the JMF are not very satisfying as they only support relatively dated and inefficient compression algorithms, which leads to poor video quality, a big transfer size or both. Third, the default Java

[11] See `http://code.google.com/intl/de-DE/apis/chart/`.
[12] See `http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/formats.html`.

security settings are very strict and forbid applets to access local videos which requires the the experimenter to edit the Java security settings on each subjects' computer[13].

For the above reasons the `video`-command is to be considered experimental at this stage and its use is generally discouraged. Moving pictures without sound can be easily achieved using animated GIFs in the `display`-command.

Unfortunately the basic problems regarding the JMF is not likely to be solved in the near future. However, as the HTML format is currently being expanded to include video, it is likely that future versions of Java may deliver a less complicated way to use videos.

3.6.3 Subject Input

The current version the BoXSPL provides four commands for requesting subject input, that is the `inputString`- and `inputNumber`-commands for string and numerical input, as well as the `choice`- and a `checkbox`-commands for selections. These commands create appropriate graphical components on the subject's screen and send every input made by a subject to the server where it is processed. A list of all available input commands as well as documentation on their usage is provided in the appendix.

3.6.4 Waiting and Assertions

When executing an experiment the BoXS continues until it encounters a wait-command. When a `wait`-command is encountered all previous statements in the program are executed and displayed to the subject, as well as a 'continue'-button. The experiment execution is then halted until the subject enters the required information of the respective stage and clicks on the 'continue'-button. The `assert`-command can be used to specify additional restrictions, for example a maximum value for an input variable.

Besides the `wait`-command the BoXSPL includes a `waitTime`-command,

[13]For instructions on how to do this see the online documentation of the `video`-command.

which waits for a specific time, a `waitForExperimenter`-command, which waits until the experimenter clicks on a button and a `waitForPlayers`-command, which waits until all subjects of a subject group have clicked on their respective 'continue'-buttons.

3.7 Matching

Matching is the process by which an experiment system assigns subjects to groups and gives them roles which are unique for each group. In an economic trust game, for example, the subjects would be partitioned into groups of two where each group designates one subject as the 'investor' and the other subject as the 'trustee'.

The most basic matching type which is provided by the BoXS is the manual matching (`matchManual(username,group,role)`), which allows the experimenter to precisely specify a group and a role for each subject. While this approach allows for the most customisation, it becomes increasingly messy and impractical as the number of subjects in an experiment increases.

The second and most common matching type is the alphabetical matching (`matchAll(roles)`), which sorts the subjects based on their user names and assigns them in alphabetic order. The experimenter only needs to specify the names of the roles and the BoXS automatically creates as many groups as possible. Note that the subjects are always assigned to the same groups if they are rematched using the same command.

Some experiment designs require a so-called stranger matching which ensures that subjects are matched to random subjects in subsequent parts of the experiment. The so-called perfect stranger matching furthermore requires that a subject is never matched into a group which contains a previous 'group-mate'. The BoXS provides the `matchStranger(roles)`- and the `matchPerfectStranger(roles)`-commands to execute these types of matching.

A perfect stranger matching requires a surprisingly high amount of calculation in order to determine the matching order which guarantees the most

possible matches. Due to this computational complexity the BoXS uses matching tables, which drastically reduces the time required for the matching but restricts the matching to combinations which have been pre-calculated. A list of all pre-calculated perfect stranger matches is provided in section A.6.2 in the appendix of this chapter.

The matching specified by any of the above matching commands is preserved until the `matchClear()`-command is called. Afterwards a new matching can be started.

If no matching is specified by the experimenter, the BoXS by default assigns all subjects into groups with one player each.

4 Design and Implementation

This section describes the technical aspects of the Bonn Experiment System (BoXS). In the first part of this section I discuss basic decisions made in designing the Bonn Experiment System, that is the choices of the programming language, the network architecture and the communication protocol. The following sections describe how both the server and the client of the BoXS have been designed and how they work internally.

4.1 Programming Language

The task of a programming language is to allow human programmers to write computer programs which can then be translated (compiled) to a native format which is executable by computers. Today several hundred programming languages exist, each designed to satisfy certain needs, for example high performance, platform independence or the support of complex scientific functions[14].

The most popular and mature programming languages which are used for application programming at the time of writing this chapter are C++, Visual

[14]See Wikipedia entry for "Programming Language":
`http://en.wikipedia.org/w/index.php?title=Programming_language&oldid=371608777`

Basic and Java.

C++, first designed in 1979 by Bjarne Stroustrup, is probably the most widely used programming language for applications and video games today. Program code written in C++ can be compiled for most platforms and usually performs very well. However, the program code has to be specifically compiled into native code and distributed for every target platform. For example, a program compiled for Microsoft Windows can not be executed under Linux or MacOS, or even on some other Windows versions.

Visual Basic is developed by Microsoft and is designed to be easier to use than C++. At the same time it is the most restrictive programming language as programs developed using it are restricted to the Microsoft Windows operating system and can not be used on other platforms.

Java, which was first published by Sun in 1995 is based on the "Write Once, Run Anywhere"-philosophy, which allows the programmer to write and compile a program once and execute it on every platform. In order to make this work Java programs are not compiled to native code but to an intermediate byte code. This byte code is then executed by the so-called Java Virtual Machine, which is available for almost all platforms. Today Java is very popular, especially for internet applications, and the Java Virtual Machine, which is required for executing Java applications, is pre-installed on most computers and can be installed for free otherwise.

One common misconception about Java is that it is slower than other languages because of the additional translation process required during the program execution. While this criticism was justified for early Java versions, the modern Java Virtual Machines have become much faster and perform just-in-time-compilation, which means that the parts of the program which are most important for its performance are automatically compiled into native code at runtime[15].

[15] See Wikipedia entry for 'Java':
`http://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=372353698`

The BoXS uses Java as the programming language for both the server and the clients in order to ensure full cross-platform compatibility, even within the same experiment. This has the advantage that it allows the BoXS to be used in internet environments. It also allows laboratories using the BoXS to freely choose the operating system for its computers, for example allowing the use of open operating systems like Linux, which may be used to reduce the costs required to set-up and administrate the laboratory computers.

4.2 Network Architecture

There are two approaches to design a network architecture. The client-server approach designates one central server computer to which all so-called clients connect. If clients want to share information in this architecture, they have to send it to the server, which then processes and/or distributes it to the appropriate receivers. The peer-to-peer approach tries to minimise the role of the server and emphasises direct connections between different clients. It has become very popular for file sharing as it provides a high bandwidth and reduces the need for powerful and costly servers. In general, the peer-to-peer approach does allow a higher bandwidth as well as a slightly lower latency.

The BoXS uses a client-server architecture for reasons similar to the ones described in **?**. First of all, the need for a server in an experiment system is hard to eliminate as subject registration, the matching of the subjects, the distribution of the experiment programs and the collection of the resulting data are intrinsically central processes and are best implemented using a server. While it would be possible to add peer-to-peer elements to the network architecture, the slight advantages in speed would probably not justify the resulting increases in complexity and effort. High bandwidth is not an important requirement for most experiments and the latency is usually low enough in the client-server approach to be hardly noticeable both in local networks and over the internet.

Basically the network structure resembles that of z-Tree with one exception. While the z-Tree program (as opposed to the z-Leaf) includes the server as well

as the experimenters' user interface, the two roles are separated in the BoXS. The BoXS server, which is described below, can therefore be executed either on the experimenters' computer or on a separate computer, for example on the official server.

4.3 Communication Protocol

The internet and most local networks support two major communication protocols. The Transmission Control Protocol (TCP) and its extension TCP/IP are widely used for most internet applications like web browsing and sending emails. It provides a high degree of reliability and guarantees the arrival of the transmitted data packages between sender and receiver in the right order. The disadvantage of TCP/IP is that it incurs a significant latency, especially if packages become corrupted or delayed[16]. The User Datagram Protocol (UDP), as opposed to the TCP, does not guarantee the correct order or even the correct arrival of data packages. Instead it provides a fast transmission speed and a low latency. The UDP is widely used for real-time applications like live audio or video streams and online games. The reasoning for this is that for a game or a voice transmission a missing package may not be perceived as bad as a constant lag which would result in a delayed playback[17].

Java supports both TCP and UDP sockets and connections. As communication based on the UDP protocol does not guarantee the correct arrival of sent packages, the programmer using it has to provide additional algorithms to account for cases in which packages were transmitted erroneously. Packages would have to be checked on arrival, unordered packages would have to be sorted and missing packages would have to be requested and sent again. As the correct and robust implementation of these functions is both tedious and non-trivial, using the TCP protocol was the obvious choice for the BoXS. Besides convenience,

[16]See Wikipedia entry for 'Transmission Control Protocol (TCP)':
`http://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=371744160`

[17]See Wikipedia entry for 'User Datagram Protocol (UDP)':
`http://en.wikipedia.org/w/index.php?title=User_Datagram_Protocol&oldid=372018803`

it is questionable if programming a secure data connection based on UDP can improve upon the corresponding mechanisms which are already implemented in the TCP.

The Bonn Experiment System uses two connections between the server and each client. While one connection for each client would be sufficient for both directions in principle, experience in developing the BoXS has shown that both performance, stability and latency of the connections can be improved by using separate connections for both directions as they allow for asynchronous data transmission.

4.4 The Server

The main task of the server is to keep track of all its connected clients, to ensure the correct transmission of data within the system and recover connections in case of connection issues. Additionally the server has to parse and execute experiment programs, correctly match and assign subjects to the correct experiment sessions and provide a way for experimenters to control and manage their experiments.

One key feature of the BoXS is that it provides official servers which can be accessed over the internet and eliminate the need for experimenters to run and administrate their own server. In order to make this possible and attractive the server has to meet particularly strong requirements concerning robustness and security.

4.4.1 Server Robustness

In order to ensure the highest robustness possible, I decided to implement the BoXS as a highly multi-threaded architecture. A thread in programming is a part of a process which can be executed separately[18]. Programs can create several threads which are then 'forked' and executed independent from each

[18] See Wikipedia entry for 'Thread (computer science)':
`http://en.wikipedia.org/w/index.php?title=Thread_%28computer_science%29&oldid=371822693`

other and at the same time.

One major advantage of using threads is that the crash of one thread does not necessarily affect the other threads. Furthermore multi-threaded programs take advantage of modern computer processors, which possess multiple processor cores and have the potential to run much faster as a result. The downside of multi-threading is that it requires a lot of sophisticated programming techniques to ensure that the threads are synchronised correctly and do not disturb each other or incur non-deterministic behaviour.

In the case of the BoXS server the main process's only task is to wait for and accept incoming connection attempts from clients. After a client connects successfully, all subsequent communication is handled by a communication thread which is immediately forked and started. Additional threads are forked for each experiment and each subjects' role in an experiment. Therefore the malfunctioning of one thread can effect neither the vital functions of the server nor the execution of other experiments.

In order to ensure that the resources of the server are shared evenly across the different experiments, several mechanisms are in place to detect and interrupt programs which get trapped in an infinite loop and consume too much processing power as a result.

So far the server program has proven to be very reliable. It should be noted, however, that no severe stress tests have been done to date. If many experiments with high levels of interaction were to run at the same time, the speed of the server might decrease and the available memory might get depleted. For such experiments the use of a separate server is recommended.

4.4.2 Connection Robustness

One worrying thought might be that subjects or even the experimenter temporarily lose their internet connection during an experiment. While the mechanisms described in the previous section already ensure that this does not affect the remaining subjects, the thought of the subjects' data being lost is not pleas-

ing.

The BoXS offers the possibility to reconnect both subjects and experimenters who lost their connection and resume the experiment at almost the exact same position where they left. In order to do this, the BoXS suspends and stores each client session which gets interrupted during an experiment for up to 24 hours. When a client tries to connect to the BoXS and provides realm and subject ids which match those of a suspended session, the session gets reassigned to the client and resumed. The server then ensures that the reconnected subjects' clients are updated by sending them the most recent experiment state.

4.4.3 Security

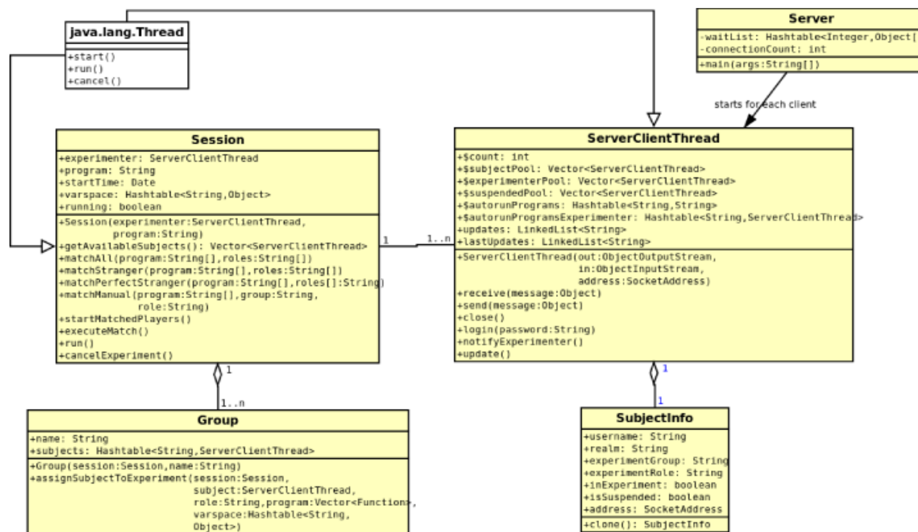
Both experiment designs and experiment results contain a lot confidential data both from the experimenter and from the subjects. This is especially important for the BoXS as it a) uses the internet as its medium, b) several experimenters work on the same server at the same time and c) the subjects use the same software client as the experimenter.

The first mechanism implemented to ensure that this data cannot be accessed by unauthorized persons is the so-called realm id, which is a string or a number specified by the experimenter at the beginning of the experiment. This exact realm id has to be entered on every other computer which is intended to enter the experiment. This ensures that the subjects and experimenters as well as the corresponding data of different experiments do not get mixed up.

The second mechanism is an experimenter password which can be specified by the experimenter and ensures that only the experimenter creating the realm or someone entrusted with the password can access the subjects' clients and their data.

Limitations

While these two mechanisms provide sufficient security for most environments, experimenters should be aware that no extensive security checks were done on



Note: Some attributes, operations and classes are omitted from the diagram in order to improve readability.

Figure 11: Class diagram of the main server classes.

the BoXS.

In the current version of the BoXS data is transmitted between the server and the clients without being encrypted, which might allow a man-in-the-middle attack by a sufficiently sophisticated and motivated hacker.

Additional security features might be implemented in future versions of the BoXS.

4.4.4 Notes on the Implementation

Figure 11 shows a Unified Modelling Language (UML) diagram of the most important server classes. The main process is the main()-function of the Server-class, which is called when the server is started and opens the TCP ports 58000 and 58001 in order to wait for incoming connections. Whenever a client connects to both of these ports, a ServerClientThread is created, forked from the main process and started.

The ServerClientThread-object provides all functions required for the server to communicate with a specific client and manages the login process as well as the information about the connected client, which is stored in a SubjectInfo-

object. While being executed, the ServerClientThread-object waits for and processes data sent by the client as it arrives, for example input generated by the subject or an experimenter's program.

Whenever an experiment is started by an experimenter, a Session-object is created which contains both the experiment program as a string as well as a variable space in which all data generated by the experiment is stored. The Session-object also contains the matching methods which create Group-objects and fill them with available subjects based on a specified matching rule.

4.5 The Client

The client is the software which runs on both the experimenters' and the subjects' computers and is designed to provide an easily usable interface for both experimenters and subjects. For experimenters it must provide the means to write, execute and supervise an experiment, as well as the possibility to receive and store data generated by an experiment. For subjects it must graphically display the current stage of the experiment, as specified by its experimenter's program, as well as receive and transmit the user input generated by the subject to the server, where it is processed. Note that both the experimenter and the subject clients are reliant on their connection to the server to fulfil their task.

4.5.1 Implementation as Java Applet

As one aim of this project is to make the system as universal and flexible as possible, the client software is implemented using the Java Programming Language, more specifically as a Java Applet. As previously described, the Java Programming Language allows the generation of program code which can be run on every operating system and every platform. Figure 12 shows the same client running in different environments and on different operating systems.

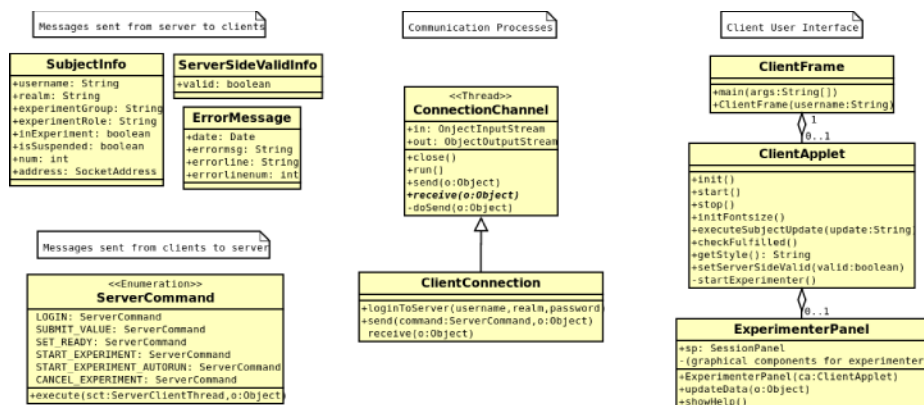
An applet is a program which can be executed within a web browser without prior installation. Java is the most common choice for programming applets and is supported by most internet browsers and used by many web sites to provide

(a) Desktop PC, Ubuntu Linux

(b) Netbook, Windows

(c) MacBook, MacOS

Figure 12: The BoXS client.



Note: Some attributes, operations and classes are omitted from the diagram in order to improve readability.

Figure 13: Class diagram of the main client classes.

advanced functionality ranging from small tools like stock and news tickers to large applications like text and spreadsheet applications or even games.

In the case of the BoXS, implementing the clients as Java applets allows them to be run on every operating system and every type of device as long as they have an internet connection and an internet browser supporting Java[19], both of which are available on most computers.

Several applets can be executed at the same time, not only on the same computer but even within the same web page. This allows multiple instances to be run side-by-side, which is useful for testing in the BoXS as several subject applets can be easily simulated at once.

In the case of the BoXS, both the experimenters and the subjects use the same client applet. Whether the applet belongs to an experimenter or a subject is evaluated at runtime based on the user name and on whether the password provided by the user is correct. The size of the applet is approximately 80 KB which is sufficiently small to be loaded without delay on most computers.

4.5.2 Internal Implementation

Figure 13 displays an UML diagram of the client classes. The main class for the clients is the ClientApplet-class, an object of which is created for every BoXS client applet that is started. Alternatively it is possible to start the BoXS as an independent program without a surrounding internet browser by executing the ClientFrame's main()-function. In this case a window is created which contains a ClientApplet-object and behaves like an applet otherwise.

The ClientApplet-object manages both the internal behaviour of the client as well as its graphical representation. When started it tries to connect to the BoXS server and either displays an experimenter's graphical user interface, as specified by the ExperimenterPanel-class, or a subject's user interface, depending on the login data. The ClientApplet also creates a ClientConnection-object which is the client analogue to the ServerClientThread and handles the connection between the client and the server.

The diagram also shows the objects used for the server-client communication in both directions. The SubjectInfo-object contains information about a subject's identity and its current status. The ServerSideValid-object is sent by the server to signal whether the input made by a subject is valid. An ErrorMessage-object contains information on the line and the description of an error which was detected while parsing or executing a program. The ServerCommand is the only object sent from a client to the server and is used to start or cancel an experiment as well as to submit values entered by a subject.

5 Limitations and Future Development

I have received a lot of feedback and was able to implement most of the proposed features and improve many aspects of the Bonn Experiment System based on it. While I highly appreciate this feedback, I am unfortunately not able to

[19] At the time of writing this chapter, most web browsers designed for desktop PCs and notebooks already support Java. While this is not necessarily true for mobile devices yet, it is very likely that full Java support arrives within the next few years as both the computational power of these devices as well as their operating systems are quickly advancing.

implement all proposals made.

5.1 Feature Selection

The obvious reason is that every additional feature and every change requires a significant amount of time and effort to implement, document and test. Especially the latter should not be underestimated as even a slight change can have widespread effects which are often hard to anticipate. In order to decide whether a certain feature is implemented, I try to estimate the likely number of affected users (Is every experimenter affected or only a very small subset?), the severity of the lack of the feature (Does it make some experiment designs infeasible or is it merely an inconvenience?) as well as the expected implementation effort (Is it done in two hours or two weeks?).

Another reason why I am hesitant to implement some proposals is that every new feature adds to the overall complexity of the system. One of the major advantages of the Bonn Experiment System is that it allows a large number of experiments while requiring the experimenter only to learn a small set of commands. The more buttons, commands and tweaking possibilities a system has, the harder and more intimidating it might become.

5.2 Future Development

Obviously it is hard to predict how the development of the BoXS advances in the future. I am dedicated to get the current version error free and intend to continue developing it in the future. If I will no longer be able to support and maintain the BoXS any further, I will try to find a way to ensure possible further development, either by publishing the software as open source or by handing it over to another researcher or programmer willing to take care of its future development.

5.3 Limitations

There are several features which were already requested and have made it to the wish list for future versions.

- Functions: Currently it is not possible for experimenters to create user-defined functions and procedures. While this is not very relevant for short experiments, the lack of user-defined functions can lead to unnecessarily long and messy programs in some cases, for example if the randomisation of stages is required.

- External devices: At the moment the BoXS offers no possibility to connect external devices like for example medical devices and input devices like joysticks.

- Delay: A slight lag exists between the execution of the program on a server and the point in time when it is displayed on the subjects' screens. While this lag is usually sufficiently small for non-time-critical experiments, it may be of importance for some experiments.

6 Conclusion

The Bonn Experiment System provides a novel and attractive way of designing and conducting laboratory and internet experiments. The possibilities to easily set-up and run experiments over the internet, to include web based content as well as advanced features like the measurement of response times allow for many new and exciting experiment designs and environments.

The possibility to execute the BoXS client applets without prior installation eases the set-up in lab environments and allows for experiments which use existing infrastructure outside labs, including the subjects' computers. True cross-platform compatibility provides freedom of choice and possible support for mobile devices in the future.

The programming language BoXSPL is easy to use and easy to learn. The small number of commands as well as the available online documentations, tutorials and sample programs make the learning process easy for novice users and provide rich possibilities for advanced users. Several experiments have been conducted using the BoXS and the feedback received from the experimenters was very positive.

One exciting and unexpected example for how the BoXS expands the space of experiment possibilities was provided by a kind professor who wrote me about how he used the BoXS in his lecture to teach about experimental methods by programming ad-hoc experiments together with his students who could participate using their laptops.

A List of all Functions in the BoXS Programming Language

A.1 Basic Operations and Calculations

A.1.1 Basic Calculus

The BoXS compiler correctly evaluates +,-,/,* and the modulus (%). It also correctly derives the priority from brackets as required.

```
1 display("<h2>Calculating</h2>")
2 display("12 + 8 = "+(12+8))
3 display("12 + 8 * 3 = "+(12+8*3))
4 display("18 / 6 = "+(18/6))
5 display("18 / (6-2) = "+(18/(6-2)))
6 display("5 * 3 + 2 = "+(5*3+2))
7 display("5 * (3 + 2) = "+(5*(3+2)))
8 display("15 % 4 = "+(15%4))
9 wait()
```

```
Calculating
12 + 8 = 20
12 + 8 * 3 = 36
18 / 6 = 3
18 / (6-2) = 4.5
5 * 3 + 2 = 17
5 * (3 + 2) = 25
15 % 4 = 3
Continue
```

A.1.2 More Calculus and Trigonometric Functions

The BoXS can calculate the natural logarithm (log), as well as the exponential function (exp), sine (sin), cosine (cos) and tangent (tan). It also provides the functions round, round1 and round2 to round a number to 0, 1 or 2 decimals.

```
1 display("<h2>Mathematics</h2>")
2 display("log(30)="+log(30))
3 display("exp(10)="+exp(10))
4 display("sin(5)="+sin(5))
5 display("cos(PI/4)="+cos(PI/4))
6 display("tan(1)="+tan(1))
7 display("round(log(30))="+round(log(30)))
8 wait()
```

```
Mathematics
log(30)=3.4011973816621555
exp(10)=22026.465794806718
sin(5)=-0.9589242746631385
cos(PI/4)=0.7071067811865476
tan(1)=1.5574077246549023
round(log(30))=3
Continue
```

A.1.3 Boolean Algebra

The BoXS can check for equality (==), inequality (!=) and compare (<,>,<=,>=). It knows the logic operations AND (&&) and OR (||).

```
1  display("<h2>Logic operations</h2>")
2  a=15
3  b=3
4  c=5
5  display("Equal:<br>(a == b * c) = "+(a==b*c))
6  display("Not equal:<br>(a != b) = "+(a!=b))
7  display("Less than/equal:<br>(a &lt;= b) = "+(a<=b))
8  display("And:<br>(1 && 0) = "+(1&&0))
9  display("Or:<br>(1 || 0) = "+(1||0))
10 wait()
```

```
Logic operations
Equal:
(a == b * c) = 1
Not equal:
(a != b) = 1
Less than/equal:
(a <= b) = 0
And:
(1 && 0) = 0
Or:
(1 || 0) = 1
[Continue]
```

Notes

- The BoXS internally uses the number 0 as false while the number 1 is treated as true.

A.1.4 Random Number Generation

Currently uniformly and normally distributed random numbers are supported.

```
1  display("<h2>Uniformly distributed in [0,1)</h2>")
2  display(randomUniform())
3  display(randomUniform())
4  display("<br><h2>Uniform integers in [5,10]</h2>")
5  display(randomUniformInteger(5,10))
6  display(randomUniformInteger(5,10))
7  display("<br><h2>Gauss distributed</h2>")
8  display(randomGauss())
9  display(randomGauss())
10 wait()
```

```
Uniformly distributed in [0,1)
0.6372758448375837
0.13205509632194734

Uniform integers in [5,10]
6
8

Gauss distributed
-0.4541906306469427
1.1714924549772257
[Continue]
```

Notes

- The random numbers are different for each subject. If they are supposed to be the same, they can be assigned to global variables.
- The numbers are generated using the internal Java random number generator, which is based on a linear congruential generator and produces pseudo-random numbers.

A.2 Program Flow Control

A.2.1 if(expression) { ... }

Tests if the expression is fulfilled (i.e. not equal to zero) and executes the code in brackets only if the expression is met.

Parameters

expression	The expression which must be fulfilled.

Notes

- Note that each curly bracket needs to be in a single line of code.

A.2.2 while(expression) { ... }

The while-command executes a part of your program repeatedly for as long as a given expression is fulfilled (i.e. not equal to zero). Compared to the for-command it is slightly more versatile.

Parameters

expression	The expression which must be fulfilled for the loop to be continued.

```
1  display("<h1>Square numbers</h1>")
2
3  i=1
4  while(i<=10)
5  {
6     display(i+" * "+i+" = "+(i*i))
7     i=i+1
8  }
9
10 wait()
```

Square numbers

1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81

Continue

```
1  i=1
2  s="<table>"
3  while(i<=10)
4  {
5     j=1
6     s=s+"<tr>"
7     while(j<=10)
8     {
9        s=s+"<td align=center width=40>"+(i*j)+"</td>"
10       j=j+1
11    }
12    s=s+"</tr>"
13    i=i+1
14 }
15 display(s)
16 wait()
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Continue

Notes

- Note that each curly bracket needs to be in a single line of code.
- In order to avoid infinite loops, execution is aborted when too many repetitions occur. An error message is given in this case.

A.2.3 for(initialization; condition; iteration) { ... }

The for-command executes a part of your program repeatedly for as long as a given expression is fulfilled. Compared to the while-command it is usually more compact and easier to use.

Parameters

initialization	Initialization code which is executed before the loop. Usually this is used to initialize a counting variable.

condition — The expression which must be fulfilled for the loop to be continued. Usually this is used to check if the counting value exceeds the number of desired repetitions.

iteration — Code which is executed after each repetition. Usually this is used to increase the counting variable.

```
1  display("<h1>Square numbers</h1>")
2
3  for(i=1; i<=9; i=i+1)
4  {
5     display(i+" * "+i+" = "+(i*i))
6  }
7
8  wait()
```

Square numbers

1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81

Continue

```
1  s="<table>"
2  for(i=1; i<=9; i=i+1)
3  {
4     s=s+"<tr>"
5     for(j=1; j<=9; j=j+1)
6     {
7        s=s+"<td align=center width=40>"+(i*j)+"</td>"
8     }
9     s=s+"</tr>"
10 }
11 display(s)
12 wait()
```

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Continue

Notes

- Note that each curly bracket needs to be in a single line of code.
- In order to avoid infinite loops, execution is aborted when too many repetitions occur. An error message is given in this case.

A.3 Displaying Text and Graphics

A.3.1 display([message])

The display-command shows a message on the subject's screen. Due to its support of HTML commands it is very versatile and can be used to display most types of information.

Parameters

message — The message to be displayed. The message can be both a constant string or a variable which is evaluated on the server.

```
1  payoff=5+3*0.25
2  display("Your payoff is "+payoff+" Euros")
3  wait()
```

Your payoff is 5.75 Euros

Continue

```
1 display("<h2>Formatting text</h2>")
2 display("Make text <b>bold</b>")
3 display(" or <i>italic</i>")
4 display(" or <u>underlined</u>")
5 display("Use <font color=red>colors</font>")
6 display(" and <font face=Courier>fonts</font>")
7 wait()
```

Formatting text

Make text **bold**
or *italic*
or underlined
Use colors
and fonts

Continue

```
1 display("<h2>Using tables</h2>")
2 table  = "<tr><th></th><th>C</th><th>D</th></tr>"
3 table = table + "<tr><th>C</th><td>10,10</td><td>5,15</tc
4 table = table + "<tr><th>D</th><td>5,15</td><td>7,7</td><
5 display("<table>"+table+"</table>");
6 wait()
```

Using tables

	C	D
C	10,10	5,15
D	5,15	7,7

Continue

Notes

- This command supports full HTML-syntax for formatting the output if required. You can also use CSS formatting.
- You can include local and remote images (including animated GIFs). You can also use internet services like e.g. Google Charts to include additional functionality to your programs.
- Text is scaled to fit the resolution of the client screen. Images are not scaled, however.

A.4 Waiting

A.4.1 wait([message],[messageafterclick])

The wait-command creates a button on each subject's display. The experiment does not continue for the subject until the button is pressed. If the current subject screen requires the subject to do something, for example enter a number, the button is disabled until she does so.

Parameters

message	The message to be displayed on the button. If no parameter is given, a standard message is displayed (Continue).
messageafterclick	The message to be displayed after the subject clicked on the button. If no parameter is given, a standard message is displayed (Please wait for the experiment to continue).

```
1  waitForPlayers("my message")
```

my message

Notes

- The button is only enabled (i.e. 'clickable') when all assertions are fulfilled and all required input elements are filled out.
- The wait-command is subject specific. When a subject clicks on the wait button, the experiment continues for this subject even if the other subjects have not finished yet.

A.4.2 waitForPlayers([message],[messageafterclick])

The waitForPlayers-command is similar to the wait-Command. The difference is that the experiment only continues after this command when all subjects of the a subject's group have pressed the button. This command can be used to synchronise groups of subjects before experiment parts which require each subject to have reached a certain point in the program.

Parameters

See wait.

A.4.3 waitTime(time)

The waitTime-command halts the execution of the program for the specified time.

Parameters

time	The time the program waits in milliseconds (!).

Notes

- No button or message is displayed for the subjects which would indicate that the experiment is waiting for a certain time. You might want to point this out in the instructions in order to avoid confusion.

A.4.4 waitForExperimenter()

Sometimes the experimenter needs the experiment to wait until he has done something, for example until she has explained the next stage to the subjects. The waitForExperimenter-command halts the execution of the experiment for all subjects until the experimenter presses the 'Ready'-button on the experimenter's client.

Notes

- No button or message is displayed for the subjects which would indicate that the experiment is waiting for the experimenter. You might want to point this out in the instructions in order to avoid confusion.

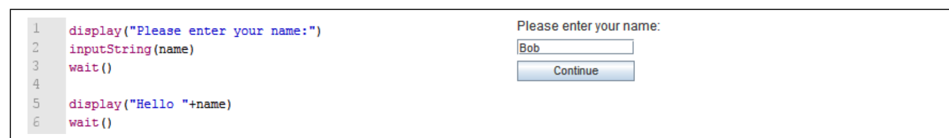
A.5 User Input

A.5.1 inputString(variablename)

Displays a text-field in which the subject can enter a value. This value is stored in a variable with the specified name. The user can type in all characters, including numbers and foreign characters.

Parameters

variablename — The name of the variable in which the result is stored.


```
1 display("Please enter your name:")
2 inputString(name)
3 wait()
4
5 display("Hello "+name)
6 wait()
```


Notes

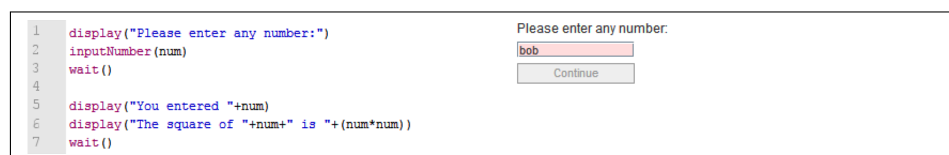
- The default text is an empty string. If you want to specify a default, assign a value to the variable before the inputString-command.
- If you want this command to be non-compulsive, which means that the subject can continue without entering something, you can use the inputStringNC-command. The syntax is equivalent.

A.5.2 inputNumber(variablename)

Displays a text-field in which the subject can enter a numeric value. This value is stored in a variable with the specified name. The BoXS enforces the entered text to be a natural or real number.

Parameters

variablename — The name of the variable in which the result is stored.


```
1 display("Please enter any number:")
2 inputNumber(num)
3 wait()
4
5 display("You entered "+num)
6 display("The square of "+num+" is "+(num*num))
7 wait()
```


Notes

- The default text is an empty string. If you want to specify a default, assign a value to the variable before the inputNumber-command.
- If the text entered is not a number the text-field is highlighted and the subject cannot continue until a correct value is entered.
- Both real and integer numbers can be entered.

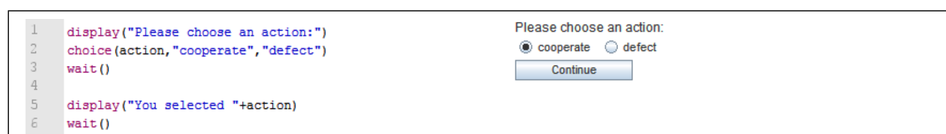
- If you want this command to be non-compulsive, which means that the subject can continue without entering something, you can use the inputNumberNC-command. The syntax is equivalent.

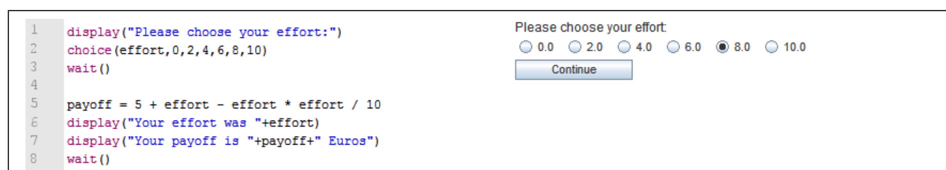
A.5.3 choice(varname,values)

The choice-command displays a group of radio buttons. The user can only select one option at a time. When the user selects a value, it is stored in a variable with the specified name.

Parameters

variablename	The name of the variable in which the result is stored.
values	A comma-separated list of strings or numbers.





Notes

- By default no option is selected. If you want to specify a default, assign a value to the variable before the choice-command.
- In some cases it is nice to have the experiment system automatically randomize the order of the choice options. The choiceRandomize(...)-function fulfils this role and allows for randomization without additional programming.
- If you want this command to be non-compulsive, which means that the subject can continue without entering something, you can use the choiceNC-command or the choiceRandomizeNC-command. The syntax is equivalent.

A.5.4 checkbox(varname,description)

The check-command displays a single checkbox along with the specified description. The user can select the checkbox or leave it unchecked. When the user selects the checkbox, the value 1 is stored in a variable with the specified name.

Parameters

variablename	The name of the variable in which the result is stored.

description	The description shown alongside the checkbox.

Notes

- By default no option is selected. If you want to specify a default, assign a value to the variable before the choice-command.
- A checkbox is always non-compulsive.

A.5.5 assert(expression)

The assert-command restricts the subject's possibilities when she is faced with input fields. Using the assert-command any amount of assertions on the variables can be added.

Parameters

expression	The assertion. The assertion can reference other variables.

```
1  display("Please enter a number greater than 5")
2  assert(val>5)
3
4  display("Also it must be less than 9")
5  assert(val<9)
6
7  display("And it must not be 8")
8  assert(val!=8)
9
10 inputNumber(val)
11 wait()
```

Please enter a number greater than 5
Also it must be less than 9
And it must not be 8
6
Continue

```
1  display("Please enter any number:")
2  inputNumber(a)
3  wait()
4
5  display("You entered "+a+":")
6  display("Now enter a larger number:")
7  inputNumber(b)
8  assert(b>a)
9  wait()
```

You entered 4:
Now enter a larger number:
3
Continue

Notes

- If an assertion is violated by user input, the user cannot continue until she chooses a valid input.
- You can impose multiple assertions on one variable. In this case user input is only allowed if it satisfies all assertions.
- The assert-command requires a wait/waitForPlayers- or button-command to have an effect.

- The assert-command does not give specific feedback to the subject on why she cannot continue when an assertion is violated. Therefore the assumptions should be made clear and communicated to the subject in the experiment instructions and/or the experiment program itself.

A.5.6 style(text)

While the default style is sufficiently attractive for most experiments, experimenters might run into situations where they need more control over how things are formatted. The style-command allows to specify a style in the cascading style sheet (CSS) format which is automatically applied to all following commands. This style can specify every format aspect ranging from font name and size up to color, transparency effects, borders etc.. It is particularly useful for the display commands but also effects buttons etc.

Parameters

text	The desired style which can be specified in the CSS format.

Example

```
1  mystyle="body{ padding: 0px; font-size: 16px; font-family: 'Ubuntu','Verdana','Arial',serif }"
2  mystyle=mystyle+"h1{font-size: 130%; margin-top: 0px; margin-bottom: 3px; font-weight: normal; }"
3  mystyle=mystyle+"h2{font-size: 115%; margin-top: 0px; margin-bottom: 3px; font-weight: normal; }"
4  mystyle=mystyle+"table{background-color: #eeeeee; border:solid; border-width:1px; border-color: #000000; margin:5px; margin-left:10px;}"
5  mystyle=mystyle+"td,th{padding:5px;text-align: center;}"
6  mystyle=mystyle+"th{background-color: #dddddd; }"
7
8  style(mystyle)
9
10 display("<h2>Einleitung</h2>")
11
12 display("Herzlich Willkommen zu diesem Internetexpriment. ")
13 ...
```

Einleitung

Herzlich Willkommen zu diesem Internetexpriment.

Sie nehmen nun an einem wirtschaftswissenschaftlichen Experiment teil. Für das Starten des Experimentes sowie das Beantworten der folgenden Fragen erhalten Sie 0 Euro. Zusätzlich können Sie im Hauptteil dieses Experimentes weiteres Geld verdienen. Sie können das Geld, welches Sie in diesem Experiment verdienen, ab der nächsten Woche werktags zwischen 10 und 12 Uhr am Lehrstuhl für empirische Wirtschaftsforschung abholen:

Lehrstuhl für empirische Wirtschaftsforschung Lennéstr. 43 53113 Bonn Ihre Teilnehmernummer ist: **655239**

Bitte notieren Sie sich diese Adresse und ihre Teilnehmernummer bevor Sie fortfahren.

Weiter

Notes

- A lot of information on how to create cascading style sheets is available on the internet.

A.5.7 manualLayout()

By default all components are arranged vertically by the BoXS (automatic layout). When this is not sufficient, the manualLayout-command can be used to specify the exact position of each component. While this is sort of cumbersome, it allows for a great amount of freedom in designing the visual appearance of an experiment.

In order to do this, the first line of a screen needs to be manualLayout() in order to disable the automatic layout. Afterwards, display and every input command take 4 additional parameters which specify the horizontal and vertical position as well as the width and the height of the respective component.

Example

```
manualLayout()
display("<div width=370><h2>Risk Preferences<hr>",10,0,400,50)

display("<b>Fixed Pay",20,50,70,30)
display("<b>Lottery",120,50,100,30)
display("<b>Your Choice",240,50,100,30)

display("200",20,80,100,30)
display("0 with 50%<br>500 with 50%",120,80,100,30)
choice(c1,"fixed","lottery",240,80,150,30)

display("250",20,120,100,30)
display("0 with 50%<br>500 with 50%",120,120,100,30)
choice(c2,"fixed","lottery",240,120,150,30)

display("300",20,160,100,30)
display("0 with 50%<br>500 with 50%",120,160,100,30)
choice(c3,"fixed","lottery",240,160,150,30)


wait("Continue","Please wait",300,220,100,20)
```

Risk Preferences

Fixed Pay	Lottery	Your Choice
200	0 with 50% 500 with 50%	○ fixed ○ lottery
250	0 with 50% 500 with 50%	○ fixed ○ lottery
300	0 with 50% 500 with 50%	○ fixed ○ lottery

Continue

Notes

- The four additional parameters must be supplied. Otherwise the component in question is not displayed.
- The origin of the coordinate system (0,0) is the top left edge of the client

applet.

- The manualLayout command is only effective for one screen. If the next screen should follow a manual layout as well the command needs to be repeated on that screen. Otherwise the BoXS defaults to automatic layout.

A.5.8 Non-compulsory Input

In some cases it is required to have input components which are non-compulsory, i.e. the subject should be allowed to proceed even if she did not fill out a component. An example for this would be a text field for an email address or a comment, which is optional. In order to allow for this, the BoXS includes non-compulsory versions of all input commands. These non-compulsory commands have the same syntax as the usual commands and end with the letters 'NC':

- inputStringNC(...)
- inputNumberNC(...)
- choiceNC(...)
- choiceRandomizeNC(...)

A.5.9 Default values

It is possible to specify default values for all input components. In order to so this one can simply assign a value to the variable in question before the corresponding input component is created. For example:

```
name="Bob"
inputString(name)
wait()
```

A.6 Matching

Matching is the process by which the subjects are assigned to groups and roles. You can use automatic matching, including (perfect) stranger matching, as well as manual matching.

Notes

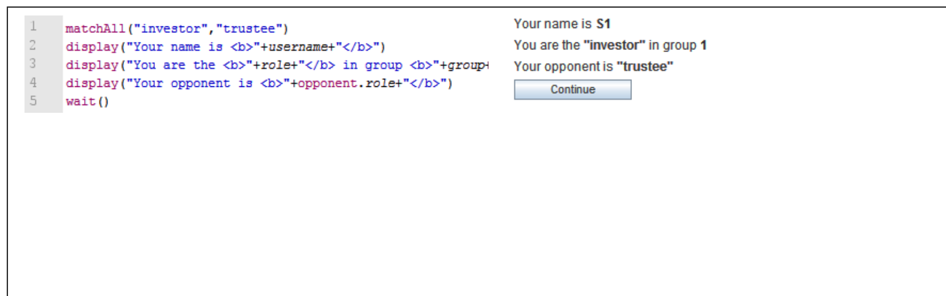
- Implicit matching: You do not need to specify matching information. If you specify no matching information, every subject is automatically allocated to a separate group and receives the role "A", which is sufficient for experiments which require no interaction among the subjects.

A.6.1 matchAll(roles)

The matchAll-command distributes the subjects in alphabetical order of their username to the desired groups/roles.

Parameters

roles A comma-separated list of all roles.



Notes

- The role names can be letters (A,B,C) or arbitrary strings (investor, trustee).

A.6.2 matchPerfectStranger(roles)

The matchPerfectStranger-command distributes the subjects to the desired groups/roles in a way that ensures that no subject are matched to the same subject again.

Parameters

roles A comma-separated list of all roles.

```
1  matchPerfectStranger(A,B,C,D)
2  // Experiment Part 1
3
4  matchPerfectStranger(A,B,C,D)
5  // Experiment Part 2
6
7  matchPerfectStranger(A,B,C,D)
8  // Experiment Part 3
9
10 matchPerfectStranger(A,B,C,D)
11 // Experiment Part 4
```

Notes

- Perfect stranger matching is an extremely computationally expensive operation. Therefore the system uses pre-calculated tables. Use the following table to figure out how many matches you can achieve given subject- and rolecount.

	Number of Roles														
Number of Subjects	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	5	1	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	7	-	1	-	-	-	-	-	-	-	-	-	-	-
9	-	-	4	-	-	-	-	-	-	-	-	-	-	-	-
10	-	9	-	-	1	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	11	3	1	-	1	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	13	-	-	-	-	1	-	-	-	-	-	-	-	-
15	-	-	4	-	1	-	-	-	-	-	-	-	-	-	-
16	-	15	-	5	-	-	-	1	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	17	5	-	-	1	-	-	1	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	-	19	-	4	1	-	-	-	-	1	-	-	-	-	-
21	-	-	7	-	-	-	1	-	-	-	-	-	-	-	-
22	-	21	-	-	-	-	-	-	-	-	1	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
24	-	?	7	5	-	1	-	1	-	-	-	1	-	-	-
25	-	-	-	-	3	-	-	-	-	-	-	-	-	-	-
26	-	?	-	-	-	-	-	-	-	-	-	-	1	-	-
27	-	-	8	-	-	-	-	-	1	-	-	-	-	-	-
28	-	?	-	5	-	-	1	-	-	-	-	-	-	1	-
29	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
30	-	?	-	-	-	-	-	-	-	-	-	-	-	-	-

- You can only do perfect stranger matching until all possible matches are use. If you try doing more matches, an error is thrown.
- Use the command matchHistoryClear to reset the perfect stranger matching algorithm.
- The role names can be letters (A,B,C) or arbitrary strings (investor, trustee).

A.6.3 matchStranger(roles)

The matchStranger-command distributes the subjects randomly to the desired groups/roles.

Parameters

roles A comma-separated list of all roles.

Notes

- The role names can be letters (A,B,C) or arbitrary strings (investor, trustee).

A.6.4 matchManual(username,group,role)

By using the matchManual-command you can manually specify how the subjects are to be matched. Each command matches exactly one subject. The first argument is the username of the subject which shall be assigned, the second and third argument are group and role.

Parameters

username	The username of the subject which shall be matched.
group	The desired group.
role	The desired role.

```
1  matchManual(S1,1,investor)
2  matchManual(S2,1,trustee)
3
4  display("username = "+username)
5  display("group = "+group)
6  display("role = "+role)
7
8  wait()
9
```

username = S1
group = 1
role = investor
Continue

Notes

- The role names can be letters (A,B,C) or arbitrary strings (investor, trustee).
- Important: In the most recent version of the BoXS username, group and role can be variables which vastly expands the possibilities of the matchManual-command. This implies that the corresponding values *must be contained in quotation marks*. For example:

  ```
  matchManual("S1","1","A")
  ```

A.6.5 matchDone()

If you want to change the matching at some point in the experiment, use the matchDone()-command. The experiment halts until all subjects have reached this point.

B Example BoXS Programs

B.1 Questionnaire

This example demonstrates how to implement a simple questionnaire.

```
1  display("<h1>Questionnaire</h1><hr>")
2
3  display("<br>Please enter your name:")
4  inputString(name)
5
6  display("<br>Please enter your field of study:")
7  inputString(study)
8
9  display("<br>Please enter your age:")
10 inputNumber(age)
11 assert(age>10 && age<100 && age==round(age))
12
13 display("<br>Please select your gender:")
14 choice(gender,"female","male")
15
16 wait()
```

B.2 Public Good Game

The public good game, as implemented in this example, is a game in which 4 players can contribute a part of their initial endowment to a group project which benefits everyone. Has a special feature it uses the Google Chart API in order to graphically display the distribution of the contributions.

```
1  // Public Good Game for 4 players
2  matchAll(A,B,C,D)
3
4  // Introductory text
5  display("<h1>Public Good Game</h1><hr>")
6  display("For this experiment you are endowed with <b>10 Euros</b>.")
7  display("You are in a group with 3 other people.")
8  display("You can contribute any amount of your endowment to a group project.")
9  display("Your payoff is increased by <b>0.4 times</b> the total contributed to the project.")
10 display("<br><img src='http://chart.apis.google.com/chart?cht=tx&chl=p_i=10%20-%20z_i%20%2B%20%200.4%20%20\cdot%20\sum_i{z_i}'>")
11
12 // Read how much the subjects want to contribute and store it in z
13 display("<br>Please choose the amount you would like to contribute:")
14 inputNumber(z)
15
16 assert(z>=0)
17 assert(z<=10)
18 assert(z==round(z))
19
20 // Wait until all players entered their respective contribution
21 waitForPlayers()
22
23 // Inform the subjects about the results of the experiment
24 display("You are player "+role+". You contributed <b>"+z+" Euros</b>. <br>The total amount donated is <b>"+sum(z)+" Euros</b>.")
25
26 display("<img src='http://chart.apis.google.com/chart?cht=p&chd=t:"+A.z+","+B.z+","+C.z+","+D.z+"&chs=200x150&chl=A|B|C|D'>")
27
28 // Calculate the payoff for each subject
29 payoff = 10 - z + 0.4*sum(z)
30
31 // Inform the subjects about their payoff
32 display("Your payoff is <b>"+payoff+" Euros</b>")
33
34 wait()
```

B.3 Chat Client

The chat client implemented in this example allows two subjects to send messages to each other.

```
1  // Colorful chat for two players, A and B
2  matchAll(A,B)
3  A.color="#33bb33"
4  B.color="#5555bb"
5
6  *.chat=""
7  quit=0
8  // Repeat until the user presses the leave-button
9  while(quit==0)
10 {
11    // Display previous chat and a textbox where the user can type
12    display("<h1>Chat window</h1>")
13    display(chat)
14    display("<br><h2>Enter your message:</h2>")
15    inputString(message)
16
17    // Buttons to send the typed message or leave the chat
18    send=0
19    button(send,"Send message")
20    button(quit,"Leave chat")
21
22    // Refresh every second
23    waitTime(1000)
24
25    // If the user pressed the send-button, append his text to the chat
26    if(send)
27    {
28       *.chat=chat+role+":  <font color="+color+">"+message+"</font><br>"
29    }
30 }
```

B.4 Dutch Auction

In a Dutch auction two subjects watch the price of a good decrease over time and can buy it at the current price by clicking on the corresponding button.

```
1  matchAll(A,B)
2
3  // Welcome screen
4  display("<h1>Dutch auction</h1>")
5  display("Click when you are ready:")
6  waitForPlayers("Ready!")
7
8  // Initialize variables
9  *.winner=0
10 price=30
11
12
13 // Repeat this until the price is 0 or a winner was found
14 while(price > 0 && winner==0)
15 {
16   // Display the current price and offer a button to bid
17   clicked=0
18   display("Current price: "+price)
19   button(clicked,"bid")
20
21   // Wait one second
22   waitTime(1000)
23
24   // If the player clicked, he/she will be the winner
25   if (clicked)
26   {
27     *.winner=role
28     *.winnerprice=price
29   }
30   price=price-1
31 }
32
33 // Display outcome
34 display("Auction finished.")
35
36 if (winner==0)
37 {
38   display("No one bought the good.")
39 }
40 if (winner!=0)
41 {
42   display("Player "+winner+" bought the good for "+winnerprice+".")
43 }
44
45 waitForPlayers()
```

B.5 Localization

Sometimes an experiment has to work in different languages at the same time. This example shows an easy way to implement such a feature.

```
// Language selection screen
display("Please select your language:")
choice(language,"Deutsch","English")
wait()

// Begin of actual experiment
welcomemessage["Deutsch"] = "Willkommen zu unserem Experiment"
welcomemessage["English"] = "Welcome to our Experiment"
display(welcomemessage[language])

question01["Deutsch"] = "Bitte geben Sie Ihr Alter ein:"
question01["English"] = "Please enter your age:"
display(question01[language])

inputNumber(age)
assert(age>10 && age<100 && age==round(age))

continue["Deutsch"] = "Weiter..."
continue["English"] = "Continue..."
continue2["Deutsch"] = "Bitte warten sie bis das Experiment weiter geht"
continue2["English"] = "Please wait for the experiment to continue"
wait(continue[language],continue2[language])
```

B.6 Real Effort Task

In this tedious real effort task subjects have to count the number of ones in a table of digits.

```
correct=0
total=0

for (repetition=0; repetition<5; repetition=repetition+1)
{
  // generate 01-table
  table=""
  count=0

  for (row=0; row<8; row=row+1)
  {
    for (col=0; col<20; col=col+1)
    {
      c=randomUniformInteger(0,1)
      count=count+c
      table=table+c+" "
    }
    table=table+"<br>"
  }

  // Display table and choices
  display("Correct: <b>"+correct+"</b> Wrong: <b>"+(total-correct))
  display("<br><pre>"+table)
  pad=randomUniformInteger(-2,2)
  display("How many ones are there?")
  choice(userc,count+pad-2,count+pad-1,count+pad,count+pad+1,count+pad+2)
  wait()

  // Was user choice right?
  total=total+1
  if(userc==count)
  {
    correct=correct+1
  }
}

```