z-Tree 2.1

**Zurich
Toolbox For
Readymade
Economic
Experiments**

Tutorial

Urs Fischbacher

Table of contents

1 About z-Tree

The z-Tree program was developed at the University of Zurich. It was specially designed to enable the conducting of economic experiments without much prior experience. It consists, on the one hand, of z-Tree, the "Zurich Toolbox for Readymade Experiments", and, on the other hand, of z-Leaf, the program used by the subjects.

In z-Tree, one can define and conduct experiments. One can program a broad range of experiments with z-Tree, including public goods games, structured bargaining experiments, posted-offer-markets or double auctions. The programming of z-Tree requires a certain amount of experience. Thereafter, the effort required for conducting experiments is minimal: An experimenter with a certain amount of experience can program a public goods game in less than an hour and a double auction in less than a day.

On performance: In Zurich, z-Tree is used for almost all experiments that are conducted with computers. 26 PCs with 486er Processors and 16 MB RAM are connected on an Ethernet. The program always works efficiently in this configuration.

The manual of z-Tree consists of two parts, the tutorial and the reference manual. The tutorial can be read sequentially. It starts in chapter 2 with a guided tour in which you learn the basic elements of z-Tree programming. This chapter is concluded with a detailed explanation of hot to set up the environment to test z-Tree on a single computer. In chapter 3 you learn programming experiments. Chapter 4 on questionnaire can be omitted for first reading. Chapter 5 on conducting experiments is essential as soon as you conduct the first experiment. It explains the normal procedure of an experimental session as well as dealing with emergencies as computer crashes. Chapter 6 explains how z-Tree can be installed in a lab.

Z-Tree is very flexible. Nevertheless, it may occur that you wish to realize something that is not covered by the program. On the web site on z-Tree at http://www.iew.unizh.ch/ztree, you find trips and tricks. If you still feel something is missing or if you should find an error in the program, please send an email to ztree@iew.unizh.ch.

I would like to thank the following users of z-Tree for their patience with the program and for their suggestions how to improve the program: Vital Anderhub, Armin Falk, Ernst Fehr, Simon Gächter, Florian Knust, Oliver Kirchkamp, Andreas Laschke, Martin Strobel, Jean Robert Tyran. I would also like to thank Alan Durell, Armin Falk, Christina Fong, Omar Solanki and Beatrice Zanella for helping me to present the program in this manual. Omar Solanki translated the original German manual into English.

1.1 Styles used in this manual

	Example:
Normal text is written in Times Roman	In a non-computerized Experiment…
Programs are written in Courier	`M = 0;`
Entries into fields are written in Courier	`0.1`

Variable text is written in italics
 	
In the example, "if", "else", and the parentheses have to be written exactly as shown. "condition" has to be replaced by the appropriate text.
	`if(` *`condition`* `) {`
	   `statements`
	`}`
	`else {`
	   `statements`
	`}`

Labels in dialogs are written in Arial	Number of subjects
Menus and Menu commands are written in Arial Bold	**Save Client Order**
The menu name can precede the menu command with a '>' as a separator. The '>' character is also used for hierarchical menus.	**File > Save As…** **New Box > Help Box…**
Stage tree elements are written in Arial Small Caps	ACTIVE SCREEN
A key is entered in square brackets.	[F5]
Modifier keys precede the key in <>.	<alt>-[tab]

2 Introduction

2.1 Terms

In a non-computerized experiment there is one or more **experimenter** and a number of **subjects.** The latter communicate with one another through the experimenter. In a computerized experiment this communication takes place through the computer. The computer operated by the experimenter is called the **experimenter PC**. The computers operated by the subjects are called **subject PCs**. The program the experimenter works with is called “z-Tree”; it is the **server program** or in short, the server. The program the subjects work with is called “z-Leaf”; it is the **client program** or in short, the client.

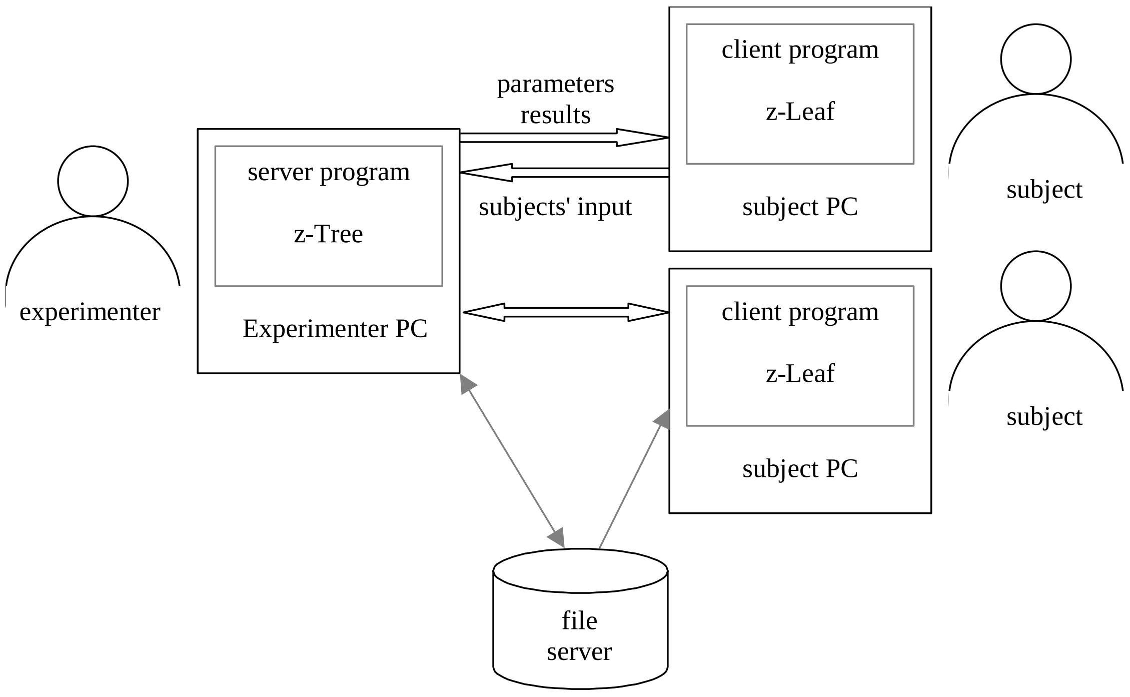


Figure: Client/Server architecture of z-Tree.

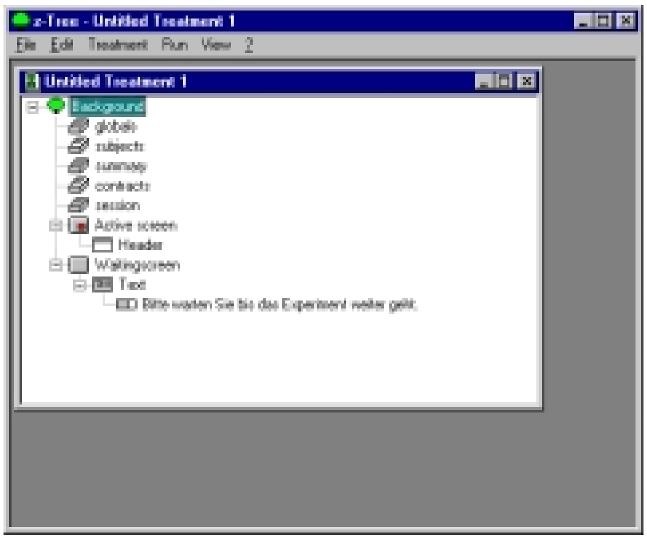
Be careful not to confuse the server program with the file server. The latter is used to save the programs, the data and the results of an experiment. A file server is not absolutely necessary for conducting an experiment. As soon as the clients have established contact with the server, communication between server and clients takes place directly, not via the file server. However, a file server does facilitate start-up. Information that is stored on the file server is readable by all. This allows the clients, for example, to find out on which computer the server was started. This makes it easily possible to use different computers as experimenter PCs.

By **session** we mean the events that occur in the time span between the arrival of the subjects and the moment they have received payment. A set of corresponding sessions constitutes one **experiment**. A **treatment** is a part of a session that is stored in a file. How treatments are defined is explained in section 3, “Definition of Treatments”. Each session consists of one or more treatments followed by a set of questionnaires. Questionnaires can also be freely defined. This will be explained in section 4.

In the chapters 2.2 and 2.3, we make a guided tour through the definition of a treatment. The purpose of these chapters is to get a first impression on how to program with z-Tree.

2.2 A guided tour of a public goods experiment

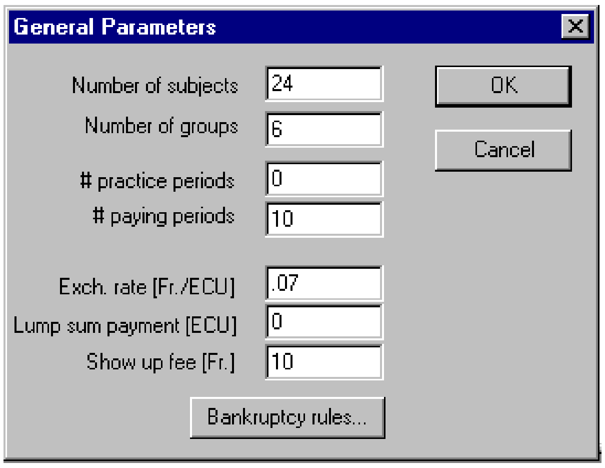
In this chapter we go through the programming of a simple public goods treatment. We will present the programming on an intuitive level and go into detail in the next chapter. In this public goods treatment, the subjects are matched into groups of four subjects. They decide how many out of 20 points they want to contribute to a public good, called project. The subjects' profit is made up of two parts: The first part is the amount they keep: 20 minus their contribution. The second part is the income from the public good: All contributions in a group are summed up, multiplied by 1.6 and distributed among all subjects in the group.



Bitte warten Sie bis das Experiment weiter geht.

Figure : Z-Tree starts with one untitled treatment.

Let us now start programming this treatment. When we start z-Tree, a window containing an untitled, empty treatment is also opened as you can see in the figure above. The treatment is represented by a tree structure called the **stage tree**: A treatment is constructed as a sequence of **stages**. Before we start constructing the stages, we do some preparation. First, we set up some parameters for the treatment. In the **background**, we enter the number of subjects, the number of groups, the number of periods and how points earned are translated into the local currency. We open this dialog either by double clicking the BACKGROUND element in the stage tree or by first selecting this line and then choosing **Info...** in the **Treatment** menu. The following dialog appears.


General Parameters

Number of subjects	24
Number of groups	6
# practice periods	0
# paying periods	10
Exch. rate [Fr./ECU]	.07
Lump sum payment [ECU]	0
Show up fee [Fr.]	10

Figure: In the dialog of the background, some general parameters can be entered.

We set the number of subjects to 24. Because we want to define groups of 4, we get 6 groups. We have 10 repetitions and therefore we set the number of paying periods (# paying periods) to 10. Finally, we set the show up fee to 10. This is the amount of money that is given to the subjects just when they show up. It is given in the local currency unit (SFR in Zurich). In Zurich, in general, we pay a show up fee of 10 Swiss Franks. The exchange rate defines the value of an internal point (experimental currency unit) in the local currency unit. In out Example 100 points are exchanged into 7 SFR.

The parameters specific to the treatment – as the endowment 20 and the efficiency factor 1.6 – are defined in a program. To insert a program, select the stage tree element just above the ACTIVE SCREEN in the stage tree element BACKGROUND. Then choose **New Program...** from the **Treatment** menu. The following dialog appears:

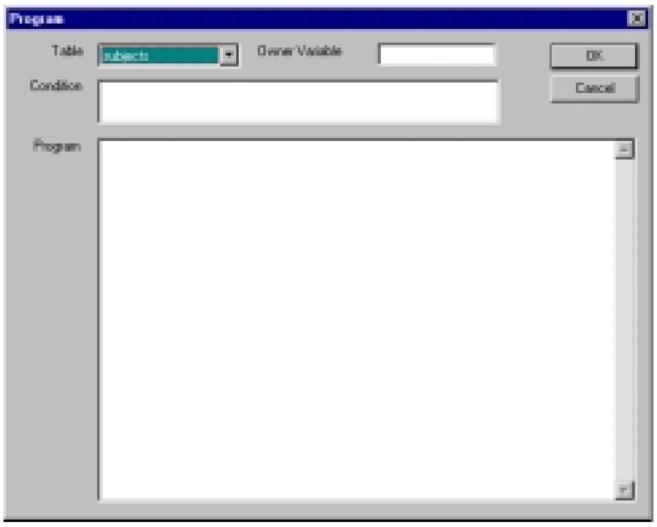


Figure 2.1: Programs are entered in the program dialog.

Enter the following lines into the field Program:

```
EfficiencyFactor = 1.6;
Endowment = 20;
```

It is a good practice to define all parameters at the beginning of the treatment, i.e. in the BACKGROUND. This makes the treatment easier to understand and it allows making changes in the parameters at one single place.

Now, we add the first stage. It is the contribution entry stage where the subjects enter their contribution decisions. To add it, we select the stage tree element BACKGROUND and choose **New Stage...** from the **Treatment** menu. A dialog opens. In this dialog, we can choose some options. In our treatment, we do not have to change the default options. We do only change the name of the stage into `contribution entry` and click at the OK button. We observe that this stage (as any stage) contains two elements (see figure below): The ACTIVE SCREEN and the WAITING SCREEN. The **active screen** represents the ‘important’ screen. On this screen a subject gets information and enters the decisions. The **waiting screen** is shown when the subject has concluded the stage. It is shown until the subject can continue.

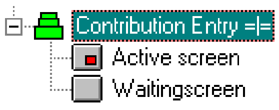
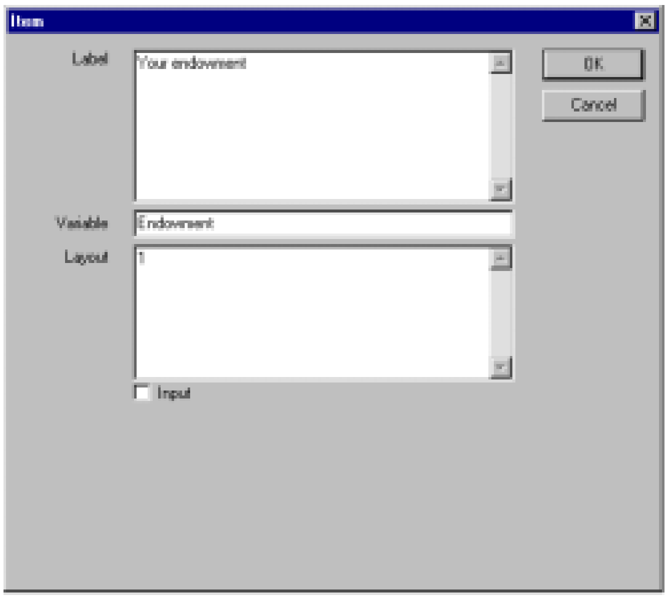


Figure 2.2: A stage with the active screen and the waiting screen as shown in the stage tree.

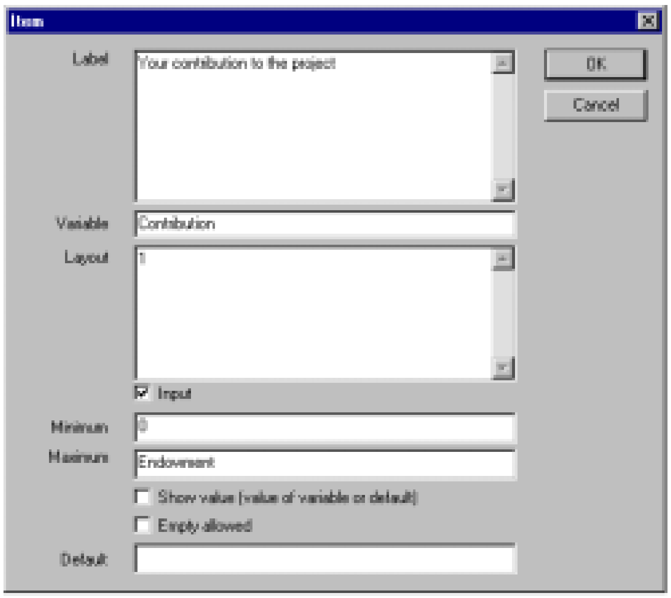
Now, we define the ACTIVE SCREEN of the CONTRIBUTION ENTRY stage. A screen consists of **boxes**: rectangular parts of the screen. There are many different kind of boxes which all can be created from the hierarchical menu **New box** in the **Treatment** menu. The most common box is the **standard box**. To create a box, we select the ACTIVE SCREEN element in the CONTRIBUTION ENTRY stage and choose **New box > New Standard box...**. In the dialog that appears, all options are already set properly. So, we conclude the dialog with OK. Into this box we place **items**. An item is the representation of a variable. First, we show the endowment variable `Endowment`. We select the STANDARD box and choose **New Item...** from the **Treatment** menu. We put `Your endowment` into the field Label, `Endowment` into the field Variable and `1` into the field Layout. The latter means that the value of the variable `Endowment` will be shown as a multiple of 1 (20 and not 20.0) and that the value will be labeled with "Your endowment".



Label: Your endowment
Variable: Endowment
Layout: 1
☐ Input

Figure 2.3: Item dialog. Items are representations of variables. This dialog show an output item.

The second item consists of the contribution entry. We will call this variable `Contribution`. It is an input variable. That means not that the value of `Contribution` will be displayed, but the subject has to enter a value and this value is assigned to the variable. Its label is `Your contribution to the project`. It must be a multiple of `1` (entered in the field Layout) and between `0` and `Endowment` (entered in Minimum and Maximum within the dialog).



Item

Label: Your contribution to the project

Variable: Contribution

Layout: 1

[✓] Input

Minimum: 0

Maximum: Endowment

[ ] Show value (value of variable or default)

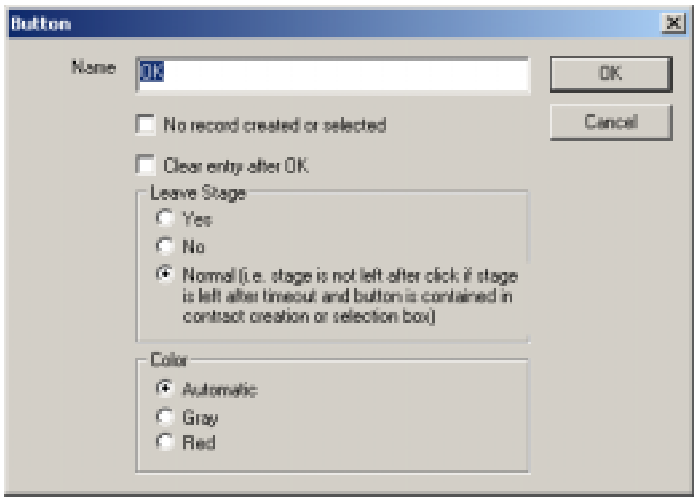
[ ] Empty allowed

Default:

OK | Cancel

Figure 2.4: This item dialog shows an input variable.

This stage will be concluded by pressing a button. This button is inserted after the input item of the variable `Contribution`. We choose **New Button...** from the **Treatment** menu. In the dialog that appears, we can enter the name of the button that is displayed to the subjects. In a stage where input has to be made, the default "OK" is a good choice. The other options in this dialog will be explained later. The default options are a good choice.



Button

Name: OK

[ ] No record created or selected

[ ] Clear entry after OK

Leave Stage
- Yes
- No
- [✓ Normal (i.e. stage is not left after click if stage is left after timeout and button is contained in contract creation or selection box)]

Color
- [✓ Automatic]
- Gray
- Red

OK | Cancel

Figure 2.5: Button dialog.

This completes the first stage.

In our treatment, there is a second stage, the profit display stage: We select the contribution entry stage and choose **New Stage...** from the **Treatment** menu. In the dialog, we name this new stage `profit display`.

In this PROFIT DISPLAY stage, we display the income of the subjects. Before we can do this, we have to calculate it. We insert a program at the beginning of this stage. It is executed when the subjects enter this stage. To insert the program, we select the PROFIT DISPLAY stage and choose **New Program...** from the **Treatment** menu. In the Program field, we enter

```
SumC = sum( same( Group ), Contribution);
N = count(  same( Group ) );
Profit = Endowment - Contribution + EfficiencyFactor * SumC/N;
```

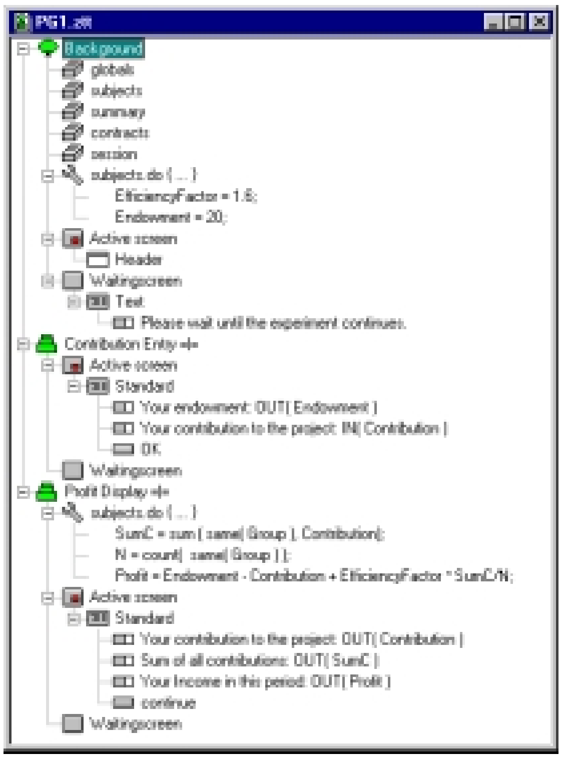
In the first line we calculate the sum of the contributions in the group of the subject. `Group` is a predefined variable. It is 1 for the first group, 2 for the second group, etc. It will be explained in detail in Chapter 3.2.1. In the second line we count the number of subjects in the group. Of course, in this treatment, we could also write `N=4;`. However, the formula above allows us to run this treatment with different group sizes without changing any line of program. In the last line, we calculate the payoff for the subject. The predefined variable `Profit` is used for this. This variable is special because at the end of each period, this variable is summed up. At the end of the session, you can easily generate a file, the so-called payment file that contains for each subject the sum of the profits made during the whole session.

Now, we define the profit display screen: First, we insert a standard box into the ACTIVE SCREEN of the PROFIT DISPLAY stage. Then, we insert three items into this box. We show:

- the own contribution,
- the sum of all contributions and
- the subject’s income for the period.

We insert the items that show the subject’s own contribution and the sum of all contributions as explained above. In the item that displays the subject’s income, we enter `.1` into the field Layout because we want to show the variable `Profit` with one digit precision. The profit display screen also has a button. We name this button "continue" because the subjects do not have to confirm anything. The button just allows the subjects to proceed more quickly than the default timeout.

This was it! Now we are done with the treatment. The figure below shows the stage tree of our public goods treatment. It is now a good time to save the treatment. Choose **Save As...** from the **File** menu. A normal "save as-dialog" appears. You can give the treatment a good name and store it in the directory you like.



```
PG1.ztt
Background
    globals
    subjects
    summary
    contracts
    session
    subjects.do { ... }
            EfficiencyFactor = 1.6;
            Endowment = 20;
    Active screen
        Header
    Waitingscreen
        Text
            Please wait until the experiment continues.
Contribution Entry =|=
    Active screen
        Standard
            Your endowment: OUT( Endowment )
            Your contribution to the project: IN( Contribution )
            OK
    Waitingscreen
Profit Display =|=
    subjects.do { ... }
            SumC = sum( same( Group ), Contribution);
            N = count(  same( Group ) );
            Profit = Endowment - Contribution + EfficiencyFactor * SumC/N;
    Active screen
        Standard
            Your contribution to the project: OUT( Contribution )
            Sum of all contributions: OUT( SumC )
            Your Income in this period: OUT( Profit )
            continue
    Waitingscreen
```

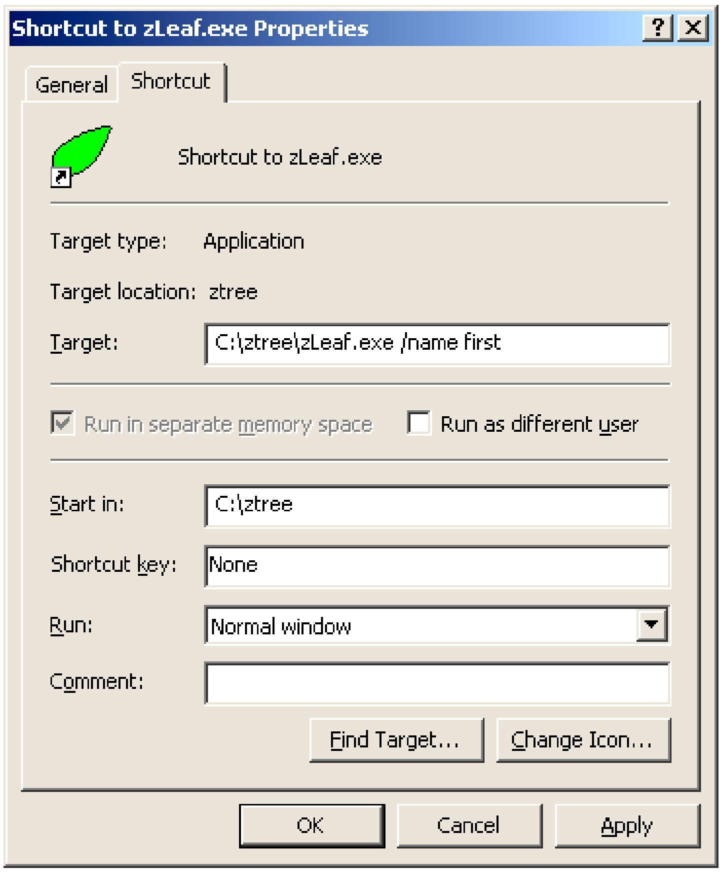
Figure 2.6: Stage tree of the public goods experiment.

2.3 First test of a treatment

In this section, we show how to test a treatment. Testing allows you to find the mistakes in a treatment. Most errors can also be found with a reduced number of subjects. It is, therefore, a good strategy to try a treatment first with a reduced number of subjects because it is much easier. In particular, you only have to make the input for a reduced number of subjects. In general, it is also sufficient to try out only one period. We will now test our treatment with two subjects and one period.

To test a treatment, you have two possibilities. You can work in a lab and start one computer per subject your treatment is programmed for. The easiest way to do this is first to start z-Tree from a directory on a file server and then to start z-Leaf from the same network directory. On each computer where you start z-Leaf, the starting screen of z-Leaf appears.

If you want to test the treatment outside of the lab, you can also run several z-Leaves on a *single* computer. You only have to give the different z-Leaves different names. You can achieve this by creating shortcuts to zleaf.exe (in explorer: Menu **File>Create Shortcut**). For each shortcut, you open the properties dialog, click on the shortcut tab and append the text " `/name` *`Yourleafname`*" to the target field (see Figure 2.7). There must be a space before and after `/name` and there may be no space after the slash (`/`). *`Yourleafname`* can be any name. It must be different for the different shortcuts. You can then start the z-Leaves by double clicking on the shortcuts. You can switch between different programs with the <alt>-[tab] key combination.



Target type: Application

Target location: ztree

Target: C:\ztree\zLeaf.exe /name first

[✓] Run in separate memory space

[ ] Run as different user

Start in: C:\ztree

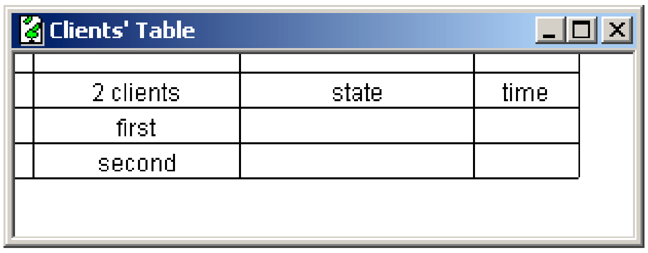
Shortcut key: None

Run: Normal window

Comment:

Figure 2.7: Shortcut dialog in windows. This z-Leaf is called "first".

No matter whether you start z-Leaf on the computer where z-Tree runs or whether you start it on another computer in the lab, you can check how many z-Leaves are actually connected with the z-Tree you are currently running. You do the following: In z-Tree, you choose **Clients' table** in the **Run** menu. When the z-Leaves connect with z-Tree, you see the names of the computers appear in the first column in this window.



2 clients	state	time
first		
second		

Figure 2.8: Clients' Table with two clients connected, a client named "first" and a client named "second".

As soon as enough clients are connected, you can start the treatment. To start the treatment, the treatment window must be in front and then you can choose **Start Treatment** from the **Run** menu. When you have started the treatment, the following screen appears on the computers where z-Leaf runs:

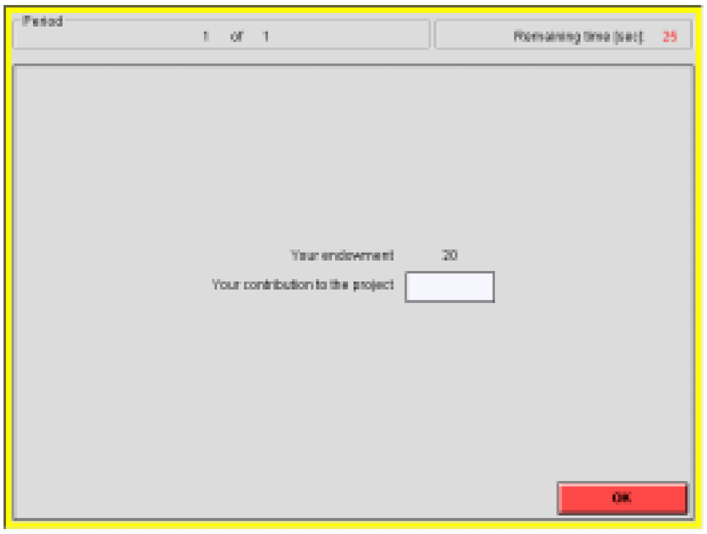
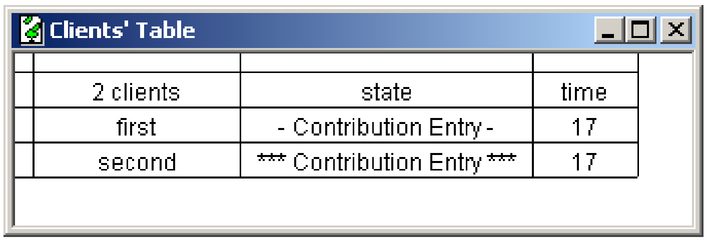


Figure 2.9: Client screen of the public goods treatment.

This is the active screen of the contribution entry stage. It consists of two **boxes**. The **header box** at the top shows the period number as well as the time remaining. Because it is contained in the active screen of the background, it is shown in the active screen of each stage. In the **standard box** we see the items and the button we defined in the stage tree. We can now enter a value between 0 and 20 into the field next to the text "Your contribution to the project" and click the "OK" button. If we enter an illegal value such as 25, -7 or 2.2, a message informs us that such an entry cannot be made.

In z-Tree, in the "Clients' Table", you can observe that the state the subjects are in. For each stage there are two states: The active state, indicated with stars, corresponds to the active screen and the wait state, indicated with a dash, corresponds to the waiting screen. This enables you to check whether there are still subjects who have to make their entries. When subjects make entries, you can observe their decisions in the so-called **subjects Table**. This table can be opened from the **Run** menu.



2 clients	state	time
first	- Contribution Entry -	17
second	*** Contribution Entry ***	17

Figure 2.10: Clients' Table when the subjects "first" has made her entry and the subject" second" has not yet done it.

After all subjects made their entries, their profits are calculated and displayed.

Checking a treatment means that you check whether everything is calculated as you intended and that the screens look fine. If you find an error, you have to correct it (or at least try to correct it) and test the treatment again. How do you correct errors? The easiest way to **correct a program** is to double click the stage tree element of the program. Then you can edit, i.e., modify it. You can **modify** any stage tree element by double clicking it or by selecting it and then choosing the menu command **Treatment>Info...** You can also **move** a stage tree element by selecting it and dragging it to a new place. If the element cannot be moved to this new position, the program will beep. You can **remove** stage tree elements by selecting them and choosing **Edit>Cut**. Finally, you can **copy** and **paste** an element: You select the element to copy and choose **Copy.** Then, you select the place where you want the element to be copied to and choose **Paste**.

We are now at the end of the guided tour. You can now exit the z-Leaves with the key combination <alt>-[F4] and quit z-Tree with the menu command **Quit**. A dialog with the following warning will appear: "The session is not finished with the writing of a payment file. Do you nevertheless want to quit?" You do not have to worry about this message. This message is only relevant when you conduct a session with real subjects. This will be explained in Chapter 3.9.5.

In the next chapter, we will explain in detail how to program treatments.

3 **Definition of Treatments**

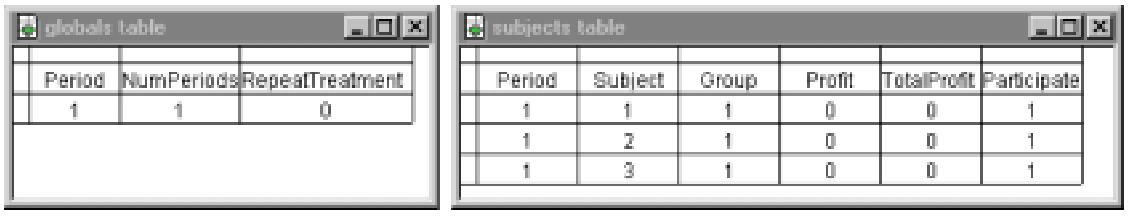
As explained in the introduction, a **treatment** is a part of a session that is stored in a file. In the previous chapter we gave a guided tour through the process of constructing a simple treatment. In this chapter, we explain in detail how to build treatments. In the following sections, we show how increasingly complex experiments are implemented in z-Tree. Each section first contains a theoretical introduction followed by examples that illustrate the concepts presented.

3.1 Simple experiments without interaction

In this chapter, we present the most essential features of z-Tree. We first explain how data is stored in z-Tree and how data is modified by programs. Then, we show how information is displayed to the subjects and how subjects' input is processed. Finally, we give a first overview of how the course of action in a treatment is organized.

3.1.1 Data Structure and Simple Programming

Information on the state of a session is stored in a **database**. This database consists of **tables**. The lines of a table are called **records**, the columns **variables.** Each variable has a name that identifies it. The individual entries in the table are called **cells**. In z-Tree, the cells can only contain numbers, not text. In Figure 3.1, you see a screen shot of two tables, the **`globals` table** and the **`subjects` table,** as they appear in an empty treatment.[1] The first row shows the names of the variables. In this example, the `globals` table contains one record (as always) and the `subjects` table contains 3 records.



globals table

Period	NumPeriods	RepeatTreatment
1	1	0

subjects table

Period	Subject	Group	Profit	TotalProfit	Participate
1	1	1	0	0	1
1	2	1	0	0	1
1	3	1	0	0	1

Figure 3.1: The `globals` table and the `subjects` table are the most important tables in the z-Tree database.

[1] When a treatment is running, you can view the tables. You just have to choose the table in the **Run** menu.

The name of a variable can be any word. To be precise, a variable can be composed of letters, numbers, and the underscore character "_". The first character of a variable must be a letter. Not allowed are diacritical marks, blanks and punctuation marks. Capital letters and small letters are distinguished. Therefore, `hallo`, `Hallo` and `hAllo` are three different variables. You can give long names to variables. This makes programs easier to understand. If you want to give a variable a name made up of several words, you can separate the words with underscores or you begin each word with a capital letter.

Examples of allowed names of variables:

```
A
contribution23
v_2_13
Buyers_offer
BuyersOffer
```

Not allowed as name of variable:

`12a`	starts with a number
`v 2 13`	contains spaces
`v.1.0`	contains dots
`Buyer’sOffer`	contains apostrophe

Some variables are predefined, i.e., they appear in every treatment. Other variables are specific to a treatment – they are defined in **programs**. Programs are placed at the beginning of stages or into the BACKGROUND between the table definitions and the ACTIVE SCREEN. Programs are always defined for a particular table – in other words, they are always executed in a particular table. This table is declared in the program dialog as shown in Figure 3.2. In this dialog, the pop-down menu contains all tables available in the treatment.

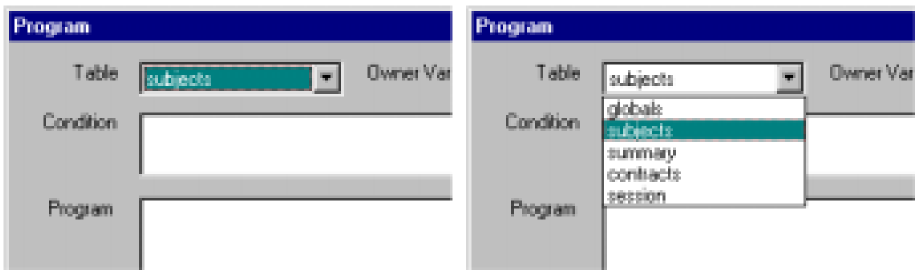


Figure 3.2: In the program dialog it has to be specified in which table the program will run.

The easiest table is the `globals` table because it contains exactly one record. Hence, the variable also determines the cell. So, we can talk about the value of a `globals` variable. For the moment, think of a program in the table.

The most important element in a program is the **assignment** statement. It does a calculation and assigns the result of this calculation to a variable. The syntax of the assignment is the following

```
        Name_of_variable = expression;
```

The expression on the right side of the equation is calculated and assigned to the variable on the left side. Note that each assignment statement is concluded with a semi-colon. If this variable has already a value, this value is overridden. If the variable does not yet exist, then it is created by this statement. All basic kinds of calculation are permitted in an expression. As usual, multiplication and division are calculated before addition and subtraction. In all other cases you calculate from left to right. Thus 2+3*4 makes 14, not 20, and 10-5-2 makes 3, not 7. A series of functions is furthermore available, such as min, max, exp, random, and round. (The reference manual includes a complete list of all operators and functions that an expression can contain.) All variables used in an expression must have been created previously, i.e., they must be defined in the program above or in the stage tree above. It is a good practice to define variables at the beginning of a treatment. You can also use comments to explain the purpose each variable (see section 3.1.2). This makes the program much more easily accessible for other experimenters.

Examples of assignment statements:

```
        p = 20;
        Q = q1 + q2;
        Profit = Endowment - Contribution + EfficiencyFactor * SumC/N;
        Cost = exp( Effort / k );
```

We have seen that the `globals` table contains exactly one record. Other tables may contain more records. If a program is executed in such a table, there is always one record for which the program is executed. This record is called the **current record**. This allows us to omit the index for the record number. Consider, for instance, the predefined `subjects` table. This table contains one record per subject. Programs in the `subjects` table are executed separately for each subject, i.e., separately for each record. So, whenever a program is running, the table and the record is fixed and therefore, within a program, the cell is determined by the variable. Let us make this clear with an example. Assume that a treatment is defined for three subjects. The value of the variable `g` is 5, 12 and 7. Consider now the following program:

```
M = 20;
x = M-g;
```

The program is executed for each row of the `subjects` table. First, it is executed for the first row: The variables `M` and `x` are defined and get their values for the first row:

g
5
12
7

->

g	M	x
5	20	15
12		
7		

Now, the program is executed for the second row. Therefore, `M` and `x` get their values for the second row:

g	M	x
5	20	15
12		
7		

->

g	M	x
5	20	15
12	20	8
7		

Finally, when the program is executed for the third row, `M` and `x` get their values for the third row:

g	M	x
5	20	15
12	20	8
7		

->

g	M	x
5	20	15
12	20	8
7	20	13

In this example, the empty spaces signify undefined values. One should not use them. After a program is executed for *all* records of a table (which is generally the case), there will be no undefined cells left. It important to note that the calculation is conducted record by record – not statement by statement. So, `M` is not known for all subjects when we are calculating `x` for the two first subjects.

3.1.2 Comments

In order to be able to easily understand a program days and weeks after you have worked on it, you insert comments. All text between `/*` and `*/`, as well as between `//` and the end of the line, is a comment and of no consequence for the actual running of the program, i.e., when the program runs, this text is simply omitted. We will use comments in our example to explain the reasons why each statement has a particular form.

Examples:

```
a=1; // initialize
b = sum( /*cos(x*x+) */ a);
// there is an error in the expression in the second line;
// therefore we put questionable parts into a comment to
// localize the error
```

Note: Comments with `/*` and `*/` may not be nested: After /*, the first occurrence of `*/` terminates the comment. The following program is therefore illegal:

```
/* discard the following lines by putting them into a comment
a /* first variable */ =1;
b = 2;
*/
```

You may also use comments to locate errors. To find an incorrect statement, you can turn doubtful passages into comments until no error message appears anymore. After you correct your errors, you can remove the comments.

3.1.3 Simple course of action

A treatment consists of a number of **periods**. This number is fixed when you run a treatment. In every period, a number of **stages** are gone through. Each stage contains two screens that are shown to the subjects who are in that stage. First, the **active screen** is shown. In this screen, subjects can make entries or view information. When the active screen of a stage is shown, we also say that the subject is in the **active state** of that stage. When data has been entered or when time has run out, subjects move to the **waiting state** of that stage and the second screen that belongs to this stage appears, which is called the **waiting screen**. From the waiting state, subjects can enter the next stage. Normally, one progresses on to the next stage when all subjects have reached the waiting state. At the beginning of each stage, calculations are carried out. These calculations are defined in **programs**.

The BACKGROUND element in the stage tree contains the information that it not specific to a particular stage. In the BACKGROUND dialog, you set for instance the number of periods and the number of subjects. Then, the BACKGROUND contains the list of tables that are used in the treatment. After the tables, the BACKGROUND may contain programs. They are executed at the beginning of each period. The elements that are contained in the active screen of the BACKGROUND are inserted into the active screen of each stage. (See also the in Chapter 3.4 on Layout.) Often, a header that shows the current period and the remaining time is shown in every stage. By inserting it in the active screen of the BACKGROUND it is not necessary to include this header in every stage. The elements that are contained in the waiting screen of the background are inserted into the waiting screen of each stage. Very often the waiting screen contains only a message such as "Please wait until the experiment continues". In this case, this message can be defined in the BACKGROUND and the waiting screen in each stage can be left empty. The following figure shows the relationship between the elements in the BACKGROUND and the elements in the stages in an abstract example with two stages.

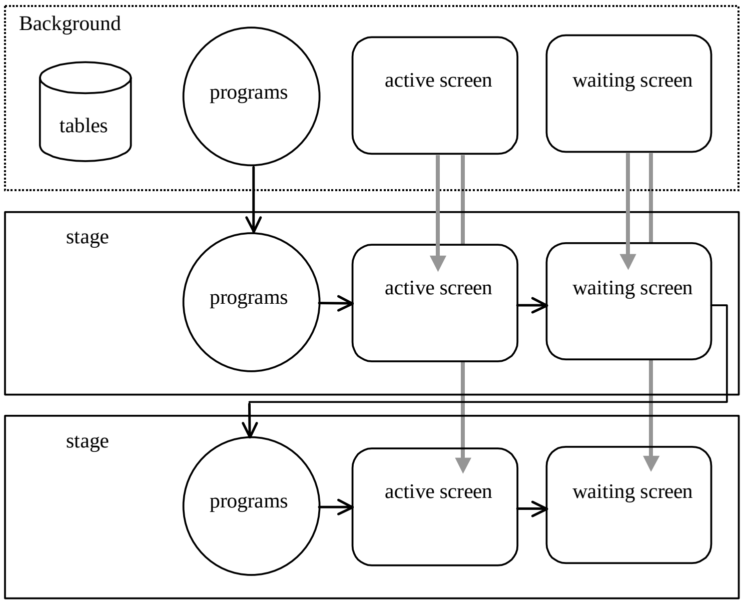


Figure 3.3: Each stage starts with some (zero or more) programs. Then subjects are shown the active screen and the waiting screen. The BACKGROUND contains the list of tables, programs that are executed at the beginning of the period and screen elements that are inserted in every stage.

3.1.4 The Stage tree

The stages of a treatment are depicted in the shape of a tree diagram, called the **stage tree**. Figure 2.6 above shows the stage tree of a public goods game. All elements of the treatment are arranged hierarchically in this figure: Stages contain programs and screens, screens contain boxes, and boxes contain items. In order to have an overview, you can fade out and fade in the lower hierarchy levels at will. By double-clicking, you can view and change the parameters of the elements. Besides this, the elements may be moved and copied. You can insert new elements with menu commands. New elements are placed either after the selected element on the same level or at the first position within the selected element. This means for instance, that you have to select the preceding stage to insert a new stage – if you have selected the WAITING SCREEN element, you will not be able to insert a new stage because a stage cannot be placed at the same or at a lower level of a screen.

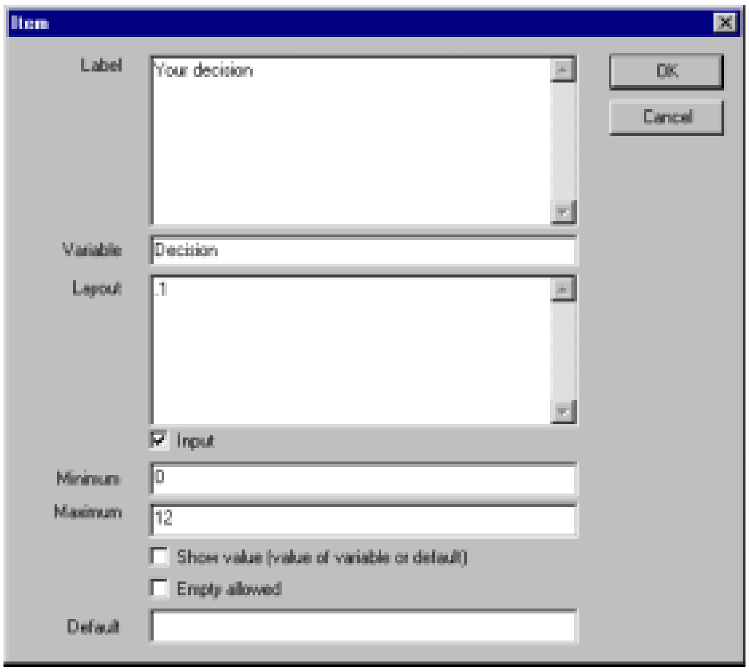
There are two "in" relations in the stage tree. A stage tree element x can contain other elements y, i.e. the elements y are placed within the element x. On the other hand, the properties of the element x are also within x – they can be found by double clicking the element. To make this difference more clear, we will use the "within" relation only for the elements and not for the properties, i.e. we will say that the elements y are within x. For the parameters, we do not use the "in" metaphor; we will say at most that a property of x is (defined) within the *dialog* of x.

3.1.5 Data Display und Data Input

The screen layout determines what data is displayed and what data input has to be made by the subjects. Screens consist of boxes – rectangular areas of the screen. There are different types of boxes. In this chapter, we only present the **standard box.** Standard boxes are particularly important because you can put **items** within them. An item is a stage tree element that allows you to display an entry of a table. The item dialog contains the label field where you can enter text. Into the variable field you can enter the name of a variable. If a standard box contains an item, then the value of the variable (in the record of the subject) will be displayed. This value is labeled with the text in the label field. The layout field is used to describe *how* the variable should be displayed. For instance, the number 12 can be displayed as 12 or as 12.00. In the layout field, we write the precision with which we display the variable. For example, it is 1 if we want to display 12 and .01 if we want to display 12.00. In the standard box, items are shown in a list form: The labels are right aligned and are placed to the left of the values of the variables. By inserting empty items, i.e., items with no label and no variable, you can create vertical space between items.

If we want subjects to make an entry, we also create an item. To declare that input must be made, we check the input checkbox. In this case, we call the item an **input item**. Other items are called **output items**. The name entered in the Variable field is again the name of a variable in the `subjects` table. It is not necessary that this variable be defined above since input items also *define* variables. When you check the input checkbox, more fields appear in the item dialog. In the Minimum and Maximum fields, we declare the lower and upper bounds of the value that the subjects may to enter. In the Layout field, we enter the precision of the number. If the number entered by a subject is not a multiple of the value entered in the Layout field, or if the value is not within the declared bounds, an error message appears on the subject's screen. In Figure 3.4, we show the dialog of an input item. This item defines a field on the subjects' screens. Into this field the subjects must enter a number. The value of this number must be between 0 and 12 and it must be a multiple of .1. Therefore, it is possible to enter 4.8 but it is not possible to enter –2, 15 or 4.55:

At the end of a standard box, we can place a BUTTON element. If so, a button appears in the standard box on the subjects' screens. If a subject presses the button, it is first checked whether all conditions for the input items are satisfied. If so, the stage is concluded.



Label: Your decision
Variable: Decision
Layout: .1
☑ Input
Minimum: 0
Maximum: 12
☐ Show value (value of variable or default)
☐ Empty allowed
Default:

Figure 3.4: Dialog of an input item. The value entered by the subject must be between 0 and 12 aud it must be a multiple of .1.

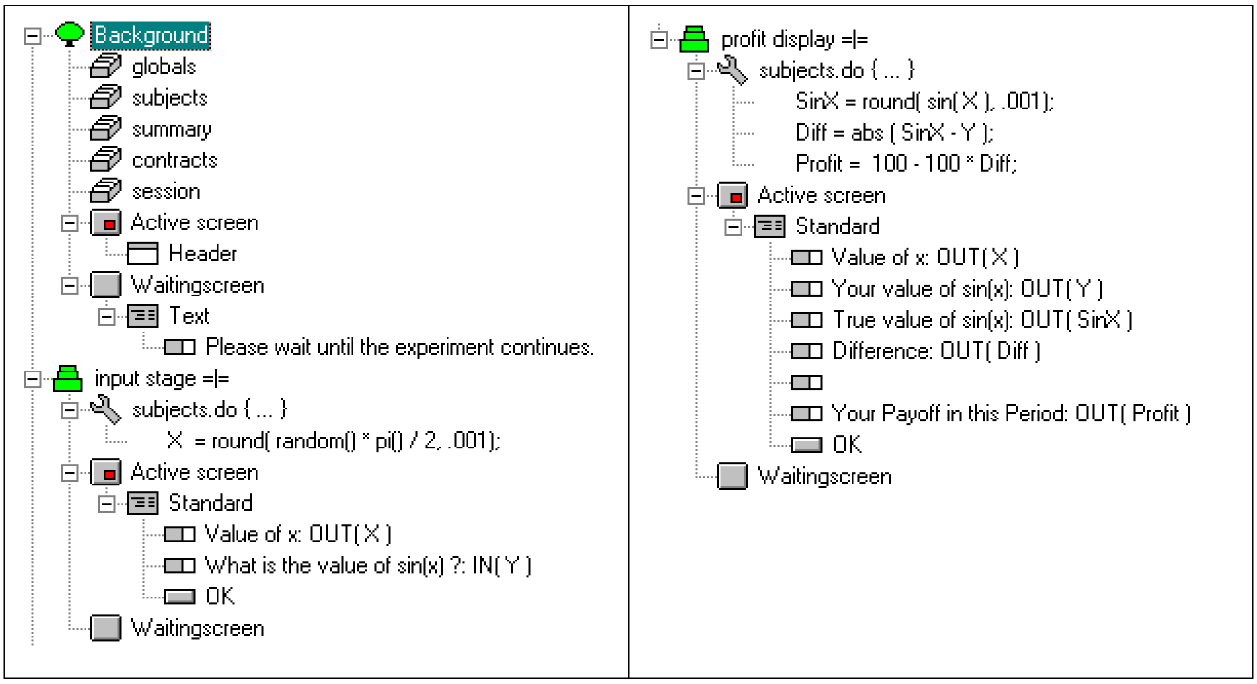
3.1.6 The Variables `Profit` and `TotalProfit`

In most economic experiments, people are paid based on their decisions in the experiment. In z-Tree, the bookkeeping of the subjects' earnings are automated. You only have to make sure that, at the end of a period, the variable `Profit` contains the number of points (a point is the experimental currency unit) earned in that period. At the beginning of a period, `Profit` is always set to zero, so if you do not change the variable, the subjects will earn nothing. During the treatment, the profit is summed up in the variable `TotalProfit`. It contains the sum of the values of the `Profit` variable in all periods in this treatment – including the current period. `Profit` and `TotalProfit` are variables in the `subjects` table. You can also display their values. However, you should not change the value of `TotalProfit`.

At the end of the treatment the value of `TotalProfit` is exchanged into the local currency unit (as SFR, €, or US$). The default value of the exchange rate is 1. It can be changed in the BACKGROUND dialog. We will explain in chapter 5 on ‘Conducting a Session’ how the sum of all profits can be accessed to pay the subjects.

3.1.7 Example: Calculation exercise (guess_sin.ztt)

Exercise: The subjects have to make a calculation – they have to calculate the sine function for a randomly determined value. They are paid according to the precision of their calculation.



```
Background
    globals
    subjects
    summary
    contracts
    session
    Active screen
        Header
    Waitingscreen
        Text
            Please wait until the experiment continues.
input stage =|=
    subjects.do { ... }
        X = round( random() * pi() / 2, .001);
    Active screen
        Standard
            Value of x: OUT( X )
            What is the value of sin(x) ?: IN( Y )
            OK
    Waitingscreen
profit display =|=
    subjects.do { ... }
        SinX = round( sin( X ), .001);
        Diff = abs ( SinX - Y );
        Profit = 100 - 100 * Diff;
    Active screen
        Standard
            Value of x: OUT( X )
            Your value of sin(x): OUT( Y )
            True value of sin(x): OUT( SinX )
            Difference: OUT( Diff )

            Your Payoff in this Period: OUT( Profit )
            OK
    Waitingscreen
```

Solution: In the stage tree in the figure you see a solution for this experiment. There are two stages: In the input stage the estimation has to be made and in the profit display stage the payoff as well as other information is displayed.

The value X is a number between 0 and ½π with a precision of .001: The function `random()` returns a uniformly distributed number between 0 and 1, `pi()` returns 3.14159… and the `round` functions rounds the resulting number to a precision of .001. This corresponds to a precision of three digits.

The input variable Y must be between 0 and 1 and we let the subjects enter a precision of at most three digits. This information has to be entered in the item dialog of the input item Y.

The payoff calculation is straightforward. First, we calculate the sinus function. Because the subjects can only enter a finite precision, we round the value to the same precision as that of the subjects' entries. Then, we calculate the absolute difference between the actual value and the guess entered. Finally, we calculate the profit in this period.

```
SinX = round( sin( X ), .001);
Diff = abs ( SinX - Y );
Profit =  100 - 100 * Diff;
```

In the active screen of the profit display stage, we show this information to the subjects.

3.2 Interactive experiments: Symmetric games

With the knowledge of the previous chapter, you should be able to program a treatment for an individual decision making problem. In this section, we explain how to program interactive experiments. The difference between an individual decision making problem and games is the fact that in games the payoff also depends on the decisions of the *other* subjects. In the terminology of z-Tree, this means that we need a feature to access cells in other rows than in the row we are currently calculating in. **Table functions** serve exactly that purpose.

In an experiment, we often want more than one group to play. For instance, if we conduct a public goods experiment, we will invite 24 subjects and we will form 6 groups of 4 subjects. You can do this easily with z-Tree. There is also great flexibility in how groups can be matched. An introduction is given in this chapter.

3.2.1 Table Functions

In the public goods example, we need to calculate the sum of all contributions made by all members of a group. The expressions described so far always refer only to the current record. In this example, we need to carry out calculations over the whole table. We call such calculations **table functions**. For instance, if `g` is the variable of the contribution, then

```
S = sum(g);
```

defines a new variable `S` as the sum of the contributions of all subjects. The variable `g` that appears in this expression now no longer belongs to the same record as `S`. If *i* is the number of the current record, then the expression above mathematically means

$$S_i = \sum_j g_j$$

Hereby, *j* loops over all records.

Of course, the argument of a table function may again be an expression. This expression is then calculated for every record of the table and these results are added in the case of the `sum` table function. In this way the program

```
x = sum( cos( a * b) );
```

corresponds to the mathematical expression

$$x_i = \sum_j \cos(a_j * b_j)$$

In every table function you can insert a condition as a first argument. This condition is checked for every record and the table function only applies to the records that satisfy this condition. Example:

```
z = average( x>0, y );
```

Here, we calculate the average of the variable `y` of the subjects with a positive `x` variable.

With the `find` function you get the value of a cell in another record.

```
v = find( b == 12, c+e );
```

Here a record is sought for which the variable `b` has the value 12. For the first record from the top that satisfies this condition, the value of `c+e` is calculated and assigned to the variable `v` (`v` in the *current* record).

3.2.2 Table functions in other tables

Table functions can also be evaluated in other tables. In order to do this you simply put the name of the table followed by a dot before the table function. For example, if you wish to calculate the average profit of all subjects and put it into the `globals` table, you write (in a program for the `globals` table):

```
avProfit = subjects.average( Profit );
```

3.2.3 The Scope Operator

Let us suppose you want to calculate the expression

$$x_i = \sum_j \cos(a_i * b_j)$$

It is the same expression as in section 3.2.1 except for the fact that we wish to use, in every summand of the sum, the variable a of the record in which the cell x_i lies, and not the a of the record of b_j. In order to express this, the variable a must be preceded by a colon. This colon is called the **scope operator**.

```
x = sum( :a * b );
```

Let us look at a table with three records in which the variable a has the values 2, 4 and 6 and the variable b has the values 5, 12 and 7. After the execution of the following program, the table contains the values as shown in the table below.

```
c = sum(  a *  b);
d = sum( :a *  b);
e = sum( :a * :b);
```

a	b	c=sum(a*b);	d=sum(:a*b);	e=sum(:a*:b);
2	5	10+48+56=114	10+24+14= 48	10+10+10=30
4	12	10+48+56=114	20+48+28= 96	48+48+48=144
8	7	10+48+56=114	40+96+56=192	56+56+56=168

Another intuition for the scope operator can be given by considering an example in which a table function is calculated in another table. So let us consider a variable v that appears in the tables A and B. Let us consider the expression `B.sum(v)` in the table A. In this expression, v is the variable v in table B. If we want access the variable v in table A, we have to use the scope operator. Because, A and B are different tables, is becomes more clear what the current record is. Outside of the table function, it is the current record in A, inside of the sum function, it is the current record when we calculate the sum. This record is in B. Nevertheless, also within the calculation of the sum, the current record in A can be accessed – with the scope operator.

So far, we discussed this table function as if there were only one record in the two tables. However, the same intuition applies if the tables contain more than one record. When we calculate an expression, we fix a current record. When we execute a table function, we calculate an expression for every record of that table. With every step another record becomes the current record. With the scope operator we may go back to the variables of the "old" current record, the record that lies *outside* the table function. With this respect, the scope operator gives us access to a wider scope of cells.

When a table function is carried out within another table function, this results in an expression of this table function with *three* records that are accessible. Let us consider the following expression where all three tables A, B and C contain a variable v. In the expression of the product, v is the variable in the current record of table C, the ‘scoped’ variable :v is the variable in table B and only by doubling the scope operator, ::v, do we reach the cell in the current record of table A. The line underneath the expression specifies which v is being used at a particular place.

```
x = v + B.sum ( v * :v - C.product ( v - :v - ::v) )
    A           B    A                C    B     A
```

In this example the variable v occurs in all tables. However, it may happen that a variable only occurs in one table. In this case you may omit the scope operator. Let us suppose that the variable a occurs only in the table A, b only in the table B and c only in C. Then the following expressions are equivalent:

```
x = a + B.sum ( b * :a - C.product( c - :b - ::a) )
x = a + B.sum ( b *  a - C.product( c -  b -   a) )
```

Note, however, that not using the scope operator can be dangerous. If at some time a variable a had been defined in table C, then the product of the second expression no longer goes back to the variable a in table A but to the a in table C.

3.2.4 Restriction of a Table Function to the Members of One’s Own Group

A common use of the scope operator consists of calculating a table function restricted to the members of the subject’s own group. The variable Group contains an ID for the group, i.e., it is 1 for the first group, 2 for the second, etc. In general, we want to restrict interaction to groups, i.e. we want to restrict a table function to the members of the own group. Suppose that we want to calculate the sum of the variables g in the own group. If we know that at most, groups 1, 2, 3 and 4 exist, we can calculate s without the scope operator:

```
s = if( Group == 1, sum(Group == 1, g),
    if( Group == 2, sum(Group == 2, g),
    if( Group == 3, sum(Group == 3, g), sum(Group == 4, g))));
```

This expression is complex, susceptible to error, and not general. In particular, if the number of groups is higher than 4, this expression is wrong.

With the scope operator, this kind of calculation can be simplified. The following expression calculates the sum of all `g` in the subject's own group.

```
s = sum(Group == : Group , g);
```

We can read this expression as follows: "(My) `s` is the sum over the `g`'s of those subjects whose `Group` is equal to my `Group`." In many contexts it is correct to translate the scope operator into "my" – as we did it her. However, the correct intuition is that the scope operator refers to the "my" in front of the `s`. The scope operator refers to the record that contains the `s` we are calculating. We need the scope operator where we want to access cells in the record belonging to the `s` we are calculating and not cells in the record belonging to the `g` in the table function.

As performing table functions on one's own group is something very common, the function **same** was specially introduced for such calculations. The expression `same(x)` is an abbreviation of `x == :x`. The expression above may therefore be written in the following way:

```
s = sum( same( Group ) , g );
```

Thanks to this operation, the scope operator becomes invisible. It is not necessary to understand the scope operator in detail in order to understand a program intuitively. However, if you want to write your own programs, a deeper understanding of the scope operator is crucial.

3.2.5 Basic Group Matching

The menu **Treatment** contains an item **matching** where the groups can be set up in partner or stranger designs. Partner matching is fixed matching. The first players constitute group 1, the next players group 2 and so on. The stranger matching is a random matching, i.e., in every period, the group is determined by the computer's random generator. By the commands in this menu, you fix the `Group` variable in the way you want. So, if you change the number of subjects or the number of groups, you have to reapply the command! More flexible matchings can be entered in the parameter table that will be explained in section 3.3.3.

3.2.6 Summary Statistics

The tables mentioned so far are reinitialized after each period (of course after being stored to disk). So, for the experimenter in the lab, they disappear form the screen. To keep an overview of the course of the experiment, you can use the `summary` table. The `summary` table contains one record per period but the table is not reinitialized at the end of a period. It is not initialized until the *treatment* ends.

If you run a program for the `summary` table, the program is only executed for the record of the current period.[2] Look at this example:

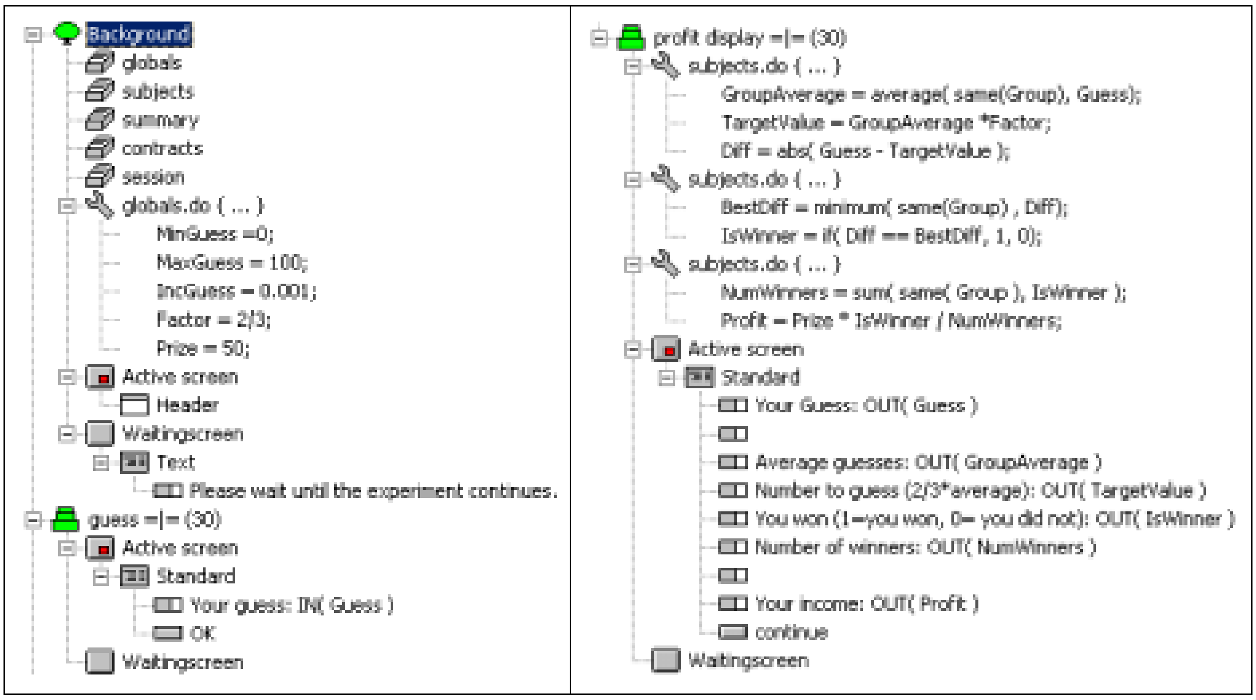
```
        AvProfit = subjects.average( Profit );
```

If you conduct the above program in the `summary` table, then in each period the variable `AvProfit` is calculated and in the `summary` table we can view at the average profits made in the treatment.

3.2.7 Example: Guessing game

The experiment: The subjects have to enter a number between 0 and 100. The subject who is closest to 2/3 of the average of all numbers entered wins a prize of 50 points. If there is a tie, the prize is shared equally among the winners.

The solution is shown in the following stage tree:



```
Background
  globals
  subjects
  summary
  contracts
  session
  globals.do { ... }
      MinGuess =0;
      MaxGuess = 100;
      IncGuess = 0.001;
      Factor = 2/3;
      Prize = 50;
  Active screen
    Header
  Waitingscreen
    Text
      Please wait until the experiment continues.
guess =|= (30)
  Active screen
    Standard
      Your guess: IN( Guess )
      OK
  Waitingscreen
```

```
profit display =|= (30)
  subjects.do { ... }
      GroupAverage = average( same(Group), Guess);
      TargetValue = GroupAverage *Factor;
      Diff = abs( Guess - TargetValue );
  subjects.do { ... }
      BestDiff = minimum( same(Group) , Diff);
      IsWinner = if( Diff == BestDiff, 1, 0);
  subjects.do { ... }
      NumWinners = sum( same( Group ), IsWinner );
      Profit = Prize * IsWinner / NumWinners;
  Active screen
    Standard
      Your Guess: OUT( Guess )
      
      Average guesses: OUT( GroupAverage )
      Number to guess (2/3*average): OUT( TargetValue )
      You won (1=you won, 0= you did not): OUT( IsWinner )
      Number of winners: OUT( NumWinners )
      
      Your income: OUT( Profit )
      continue
  Waitingscreen
```

[2] However, if you apply a table function or the `do` command, the program runs through all records. If you want to restrict the program to the current period, you have to add the condition `same(Period)`.

In the BACKGROUND, we define as usual the constants. In the first stage, the subjects enter their guesses. In the second stage, we perform the calculations. Let us explain each step of the calculations:

The group average is calculated with the table function `average`.

```
        GroupAverage = average( same(Group), Guess);
```

The value to guess, called `TargetValue`, is just the product of the average and the factor of 2/3.

```
        TargetValue = GroupAverage *Factor;
```

The relevant difference is the absolute value of the difference between the guess and the value to guess.

```
        Diff = abs( Guess - TargetValue );
```

The smallest difference, called `BestDiff` is the minimum of all differences.

```
        BestDiff = minimum( same(Group) , Diff);
```

It is important that this table function is placed into a new program. Programs are executed row by row. So, if we would place the calculation of `BestDiff` into the same program as the calculation of `Diff`, we would not calculate `BestDiff` correctly because we do not know the value of `Diff` for the records later in the table. In the current implementation, an empty value is set to zero. So, for all subjects except the last one, `BestDiff` would be calculated as zero. If we put the calculation of `BestDiff` into a new program, everything is done correctly. First, the first program is calculated for all records (i.e., for all subjects). Then, the second program is calculated for all records. At this moment, `Diff` is calculated for all records and we can apply the table function `minimum`.

In the next statement, we determine the winner(s). We put 1 into the variable `IsWinner` if a subject was closest and 0 otherwise.

```
        IsWinner = if( Diff == BestDiff, 1, 0);
```

The `if` function takes three arguments. The first argument is a condition: `Diff == BestDiff`. If this condition is satisfied, the second argument is the result of the `if` function (i.e., 1). If the condition is not satisfied, then the third argument of the `if` function is the result of the `if` function (i.e., 0).

Next, we calculate the number of winners to deal with the ties. We have to place this statement again into a new program because in the table function `sum`, we use a value that is calculated in the program above.

```
        NumWinners = sum( same( Group ), IsWinner );
```

Finally, we calculate the profit. It is zero for those subjects who did not win (`IsWinner` is zero) and it is `Prize*1/NumWinners` for the winners.

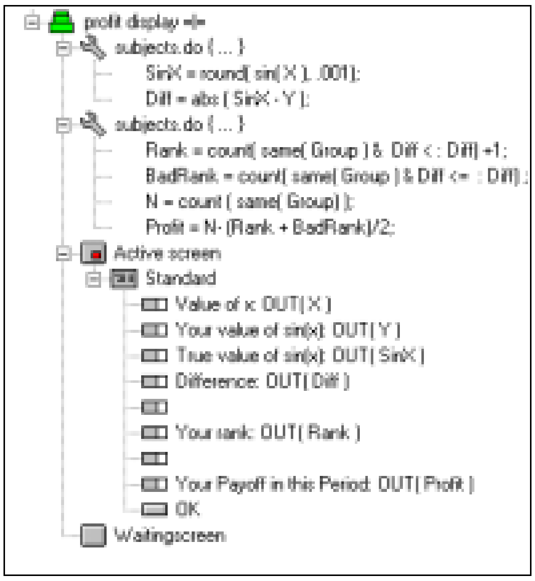
```
Profit = Prize * IsWinner / NumWinners;
```

3.2.8 Example: Rank dependent payment

The experiment: The subjects have to guess the value of a mathematical function as in example 3.1.7. However, subjects are paid according to their relative performance. The best player receives one point less than there are group members. The second player receives one point less than the first and so on until the last player who receives zero points. Finally, we apply a cost neutral tie rule. So, if for instance two players have rank 2 then both get a payoff according to their average rank 2.5. In the following table, we show an example. In the first line, we list an example of the rank. In the second line, we show the rank that results if we apply a tie rule that assigns the last rank in a group of players with the same rank. In the last line, we list the profit.

Rank	1	2	2	4	4	4	7	8	9	9
Bad Rank	1	3	3	6	6	6	7	8	10	10
Payment	9	7.5	7.5	5	5	5	3	2	.5	.5

Because only the profit display stage is different from the example 3.1.7, we show and explain only this stage.



```
profit display =|=
    subjects.do { ... }
        SinX = round[ sin[ X ], .001];
        Diff = abs [ SinX - Y ];
    subjects.do { ... }
        Rank = count[ same( Group ) & Diff < : Diff] +1;
        BadRank = count[ same[ Group ] & Diff <= : Diff] ;
        N = count ( same( Group) );
        Profit = N- (Rank + BadRank)/2;
    Active screen
        Standard
            Value of x: OUT[ X ]
            Your value of sin(x): OUT[ Y ]
            True value of sin(x): OUT[ SinX ]
            Difference: OUT[ Diff ]

            Your rank: OUT[ Rank ]

            Your Payoff in this Period: OUT[ Profit ]
            OK
    Waitingscreen
```

The second program is new in this treatment. The statements in this program cannot be placed into the previous program because we calculate table functions that use the variable `Diff` (see section above). Let us explain the program step by step.

```
Rank = count( same( Group ) &  Diff < : Diff) +1;
```

In this line we calculate the rank. The rank corresponds to the number of players who are strictly better than I am. This is expressed with the condition `Diff<:Diff`. The `count` table function can be read as follows: We count the number of players in my group whose difference is smaller than my difference. We have to add `1` because if *no* player is better than I am, then I am first.

```
BadRank = count( same( Group ) & Diff <=  : Diff);
```

In this statement, we count the number of players who are at least as good as I am. This corresponds to the rank with the 'unkind' tie rule.

```
N = count ( same( Group) );
```

This statement just counts the number of players in the group.

```
Profit = N- (Rank + BadRank)/2;
```

The relevant rank is the average if the two ranks calculated above. The payoff depends on this average rank as calculated in this formula.

3.3 Asymmetric, simultaneous games

In the previous chapter, all subjects had the same parameters. In this chapter, we explain how to implement different parameters for different subjects. There are essentially two ways. One way is to calculate the parameters conditionally with `if` functions and statements. The second way is to use the parameter table.

3.3.1 Conditional execution

In z-Tree, it is possible to execute programs depending on the value of the entries in the database. This is done by using **conditions**. Conditions are expressions that do not represent a number but a 'logical value', i.e., either true (`TRUE`) or false (`FALSE`). An example of a condition is `g>=h`. This condition results in `TRUE` if and only if the variable `g` is at least as great as the variable `h`. Another example, the condition `m==n` results in `TRUE` if and only if the variables `m` and `n` are equal. Note that in contrast to the equals sign in the assignment statement, the equals sign used in conditions consists of two equals signs.

Conditions may not be directly assigned to variables, however they may be used in expressions. With

```
if( c, x, y )
```

a **conditional calculation** can be carried out. Here, `x` and `y` are 'normal' expressions and `c` is a condition. If `c` evaluates to `TRUE`, the `if` expression returns the value of `x`, otherwise it returns the value of `y`. Because `x` and `y` can be any expression, they can also contain `if` functions. In this case we say that these `if` functions are nested. (You can of course nest any kind of function.) The following expression shows a nested `if` function that implements a profit function for a two person game in which both players can choose either 1 or 2. `Profit11`, `Profit12`, `Profit21` and `Profit22` describe the payoff matrix for the subject. The variable `Profit21`, for instance, contains the payoff for the subject if the subject chooses 2 and the other subject chooses 1.

```
Profit = if( Decision == 1,
           if( OthersDecision == 1, Profit11, Profit12 ),
           if( OthersDecision == 1, Profit21, Profit22 ));
```

If you wish to calculate a condition and keep it for future use, you have to store it in a normal variable, i.e. in a number. In this case you use the standard `0` for `FALSE` and `1` for `TRUE`.

An alternative to the `if` *function* is the `if` *statement*. It has the syntax

```
if( condition ) {
     statements1
}
```

or

```
if( condition ) {
     statements1
}
else {
     statements2
}
```

If the condition evaluates to `TRUE`, then all of the statements between the first pair of curly brackets are executed (statements1). If it evaluates to `FALSE`, then in the first case nothing happens and in the second case, all the statements between the pair of curly brackets after the `else` statement are executed (statements2). An assignment statement using an `if` function can be translated into an equivalent if statement. For instance,

```
z = if( c, x, y);
```

is equivalent to

```
if (c) {
     z=x;
}
else {
     z=y;
}
```

In this example, the second form is more complicated. However, it is better suited in cases where we use one condition in several expressions. For instance, this is the case when we have different types of players with completely different payoff functions.

3.3.2 Calculating Different Values for Different Subjects

There are variables called `Period` and `Subject` that are set by the z-Tree. `Period` contains the number of the current period where 1 stands for the first paid period. The variable `Period` is defined in every table.[3] The variable `Subject` is defined in the `subjects` table. It contains the number 1 for the first record, 2 for the second and so on. So, the variable `Subject` allows you to identify each subject.

You can now define variables in subject or period-specific terms by using these variables:

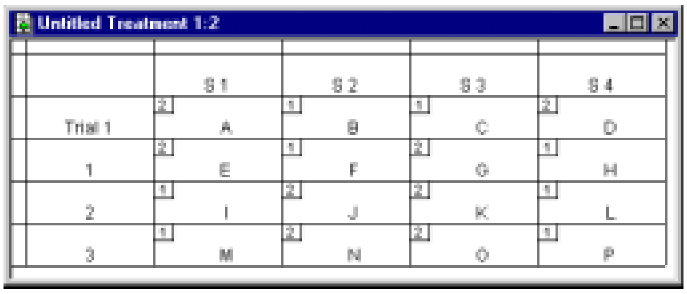
```
if( Period== 1 & Subject ==1 ) {
     p=11;
}
if( Period== 1 & Subject ==2 ) {
     p=12;
}
…
```

This method of defining subject specific values can be rather complicated. Therefore, a second method has been developed, the parameter table.

3.3.3 The Parameter Table

For each treatment, there is, besides the stage tree, a **parameter table** where subject-specific variables may be managed quite easily. In this table, periods are given in the rows and subjects in the columns.

[3] To be precise, the variable `Period` is defied in each table with lifetime period. Lifetime of variables is explained in Section 3.9.2.



Untitled Treatment 1:2

	S 1	S 2	S 3	S 4
Trial 1	2 A	1 B	1 C	2 D
1	2 E	1 F	2 G	1 H
2	1 I	2 J	2 K	1 L
3	1 M	2 N	2 O	1 P

The subjects are named S1, etc. and the periods are numbered. Trial periods are preceded by the term 'Trial'. In the cells of this table, programs can be entered. These programs are executed for the corresponding periods and subjects. For instance, the program in the cell S2 is executed for subject 2 in every period. This means that this program is executed in every period for the second row in the `subjects` table. In this cell, we define the **role parameters**. The program contained in cell F is also executed for subject 2. However, it is only executed in period 1, the first paid period. In this cell, we define the **specific parameters**. The program in the cell label with 1 (first column) is executed in the `globals` table. In it, we define parameters that are the same for all subjects (but differ from period to period). In this cell, we define the **period parameters**.

The programs in the parameter table can be viewed and changed with the command "Info..." from the menu "Treatment" or by double-clicking the field in question. Whole cells can be copied either with Copy/Paste or by selecting them and then dragging them over from one field to another.

In the cells of the specific parameters, there is a small number in the upper left corner. This is the group number. This number can be modified in the dialog of the cell or by applying the matching procedures in the **Treatment** menu.

For every program, it is checked that no undefined variables are used. Variables are defined in input items and when they get assigned a value. However, assignments in the parameter table are not considered as definition. This means that variables that are only defined in the parameter table are not considered as defined in the stage tree. You have to initialize them in a program in the BACKGROUND of the stage tree. This also guarantees that these variables always have a default value – even if they are not defined in every cell in the parameter table. Values that are assigned in the BACKGROUND can be overridden in the parameter table, because at the beginning of a period, the database is set up as follows:

1. Setting of standard variables as `Period`, `Subject` and `Group`
2. Running of programs in background
3. Running of subject program (in current period) in the subjects table

4. Running of role program in subjects table
5. Running of period program in globals table
6. Running of programs of first stage

Example: Let us suppose that there are different types of subjects, e.g., type 1 and type 2. In this case you first define a default value for the variable in the BACKGROUND; e.g.

```
Type=0;
```

(You can use an illegal value here only if you redefine the variable in *every* cell.) Then, you double-click in the fields in which you wish to change the value. In this dialog, you first give this cell a name that represents the type. This makes it easier to manage the parameter table. Then you enter the program in which the value is changed. For instance:

```
Type=1;
```

Remark: It is a good practice in programs to use names instead of numbers whenever this is possible. So, in the public good game we defined a variable Endowment that kept the value of 20. This makes the program easier to understand and easier to change. When we have values that distinguish different types, it is reasonable to define variables for the type options in the globals table. If we have proposers and responders in an experiment, we could define:

```
PROPOSERTYPE = 1;
RESPONDERTYPE = 2;
```

Then, you can define the type as:

```
Type= PROPOSERTYPE;
```

3.3.4 Importing parameters

If a parameter is different for all periods and subjects, it becomes impractical to double-click the field in question and to adapt the program for every cell of the parameter table. Instead you can set up a tab-separated table with the variable values in an editor and import it from the **Treatment** menu with the command **Import Variable Table…**. This command is available if the parameter table is the active window. If you choose this command, you first have to enter the name of the variable you want to import. Assume you name it `MyVar`. Then, you choose the file where you have entered the values. Then the file is processed: Wherever there is a nonempty entry in the table, then the following line is added to the program of the corresponding specific parameter. If the value in the table equals 45, for instance, the line added will be:

```
MyVar = 45;
```

You are not restricted to use numbers; you can also use variable names in the table. (It is even possible to import programs.)

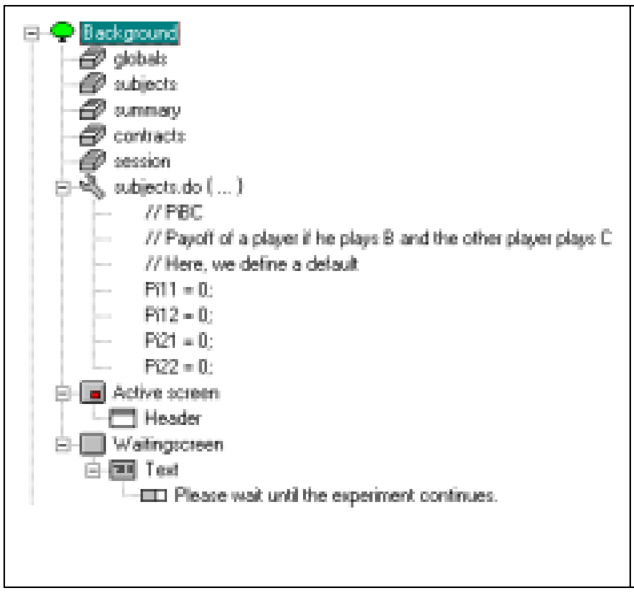
3.3.5 Group Matching

The group matching is determined by the variable `Group`. This variable is initialized at beginning of the period when the other standard variables are initialized. It is then set to the value in the upper left corner of the cell of the specific parameters. You can modify this value in the dialog or with the menus in the **Treatment>Matching** menu. So you can for instance define fixed (partner) or random (stranger) matchings. Whenever you apply a matching command, this command is applied to the selected area in the parameter table or – if there is nothing selected – to the whole parameter table. The matching commands directly modify the number in the upper left corner in the cell of the parameter table. They do not enforce a logical matching structure. So, if you change the number of subjects or periods in a treatment, you have to reapply the matching command.

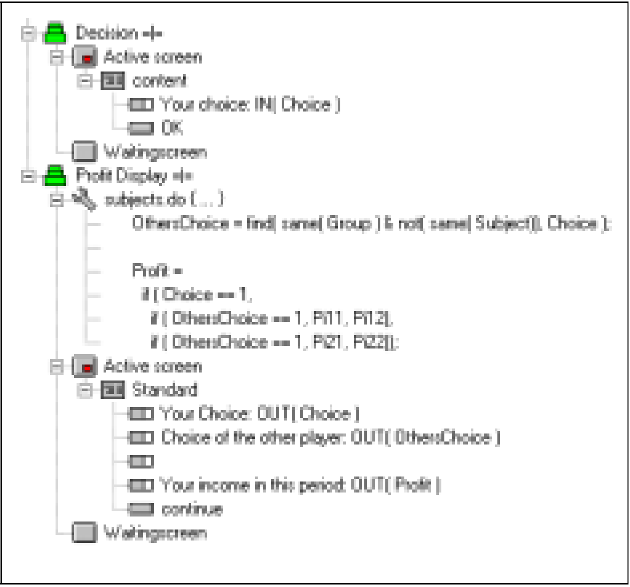
You can also change the Group variable in programs. This allows making endogenous matching or the definition of matchings that are independent of the number of subjects.

3.3.6 Exercise: General 2x2 game (game222)

In this example, we program a battle of the sexes game. We program the example in such a way that we can use it for any two player 2x2 game.



```
Background
  globals
  subjects
  summary
  contracts
  session
  subjects.do { ... }
      // PiBC
      // Payoff of a player if he plays B and the other player plays C
      // Here, we define a default
      Pi11 = 0;
      Pi12 = 0;
      Pi21 = 0;
      Pi22 = 0;
  Active screen
    Header
  Waitingscreen
    Text
      Please wait until the experiment continues.
```

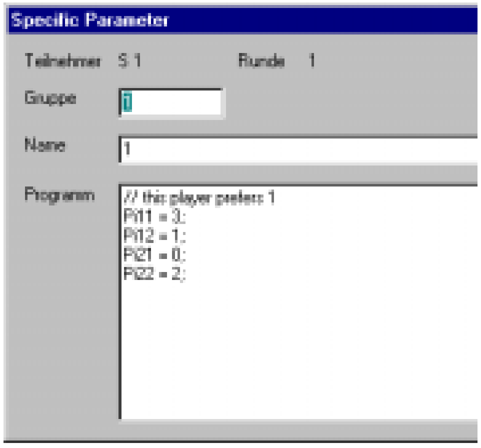


```
Decision =|=
  Active screen
    content
      Your choice: IN( Choice )
      OK
  Waitingscreen
Profit Display =|=
  subjects.do { ... }
      OthersChoice = find( same( Group ) & not( same( Subject)), Choice );

      Profit =
        if ( Choice == 1,
          if ( OthersChoice == 1, Pi11, Pi12),
          if ( OthersChoice == 1, Pi21, Pi22));
  Active screen
    Standard
      Your Choice: OUT( Choice )
      Choice of the other player: OUT( OthersChoice )

      Your income in this period: OUT( Profit )
      continue
  Waitingscreen
```

In this experiments, the players may have different roles. However, they differ only with respect to the parameters. They have different payoff functions. We implement this in the following way. First, we define in the BACKGROUND defaults for the payoff matrix. The variable `Pi`*`BC`* is Payoff of a player if he plays *B* and the other player plays *C*. For instance, `Pi21` is the payoff the player gets if he plays 2 and the other player plays 1. The actual parameters for the different subjects are then defined in the parameter table as shown in the following figures.

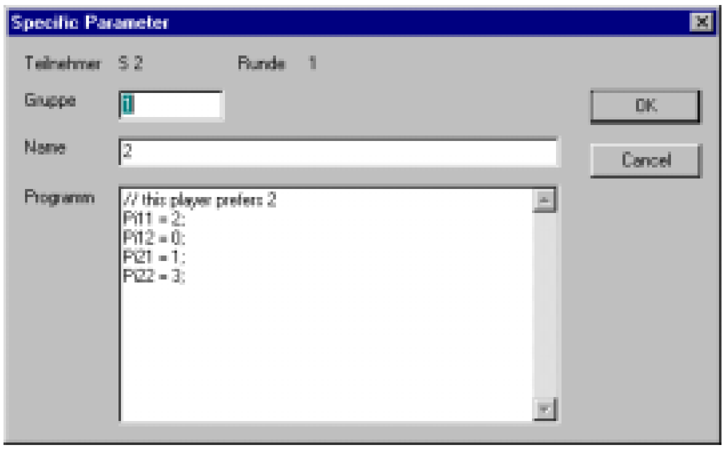


Specific Parameter

Teilnehmer	S 1	Runde	1
Gruppe	1		
Name	1		

Programm

```
// this player prefers 1
Pi11 = 3;
Pi12 = 1;
Pi21 = 0;
Pi22 = 2;
```



Specific Parameter

Teilnehmer	S 2	Runde	1
Gruppe	1		
Name	2		

Programm

```
// this player prefers 2
Pi11 = 2;
Pi12 = 0;
Pi21 = 1;
Pi22 = 3;
```

To calculate the payoff, we need to know the choice of the other player. Because the other player's variable `Choice` is not in our record, we have to apply a table function. The function `find` runs through the table and returns an entry as soon as some condition is met. The other player's record has the properties that the `Group` variable has the same value as ours and the `Subject` variable is different. This is expressed in the command:

```
OthersChoice = find( same( Group ) & not( same( Subject)), Choice );
```

Finally, the profit is calculated with a nested if function.

```
Profit =
  if ( Choice == 1,
    if ( OthersChoice == 1, Pi11, Pi12),
    if ( OthersChoice == 1, Pi21, Pi22));
```

In this payoff function, the parameters in the parameter table are used.

There are many solutions to solve a particular problem. Let us also sketch an alternative solution: We could define a variable `Type` for the two types of players. Then we define the two payoff matrices in the `globals` table. In the parameter table, we only set the `Type` variable. Finally, the payoff calculation is a bit more complex because we have to distinguish between the two types.

3.3.7 Exercise: Group definition in a program

As mentioned in section 3.3.5, the matching can also be programmed. We show here sample programs for fixed and random matching. We assume that the variable `NumInGroup` is defined and contains the number of subjects per group. Let N be the number of subjects. It can be calculated as `N=count();`.

The following program implements a partner matching:

```
Group = rounddown( (Subject – 1) / NumInGroup, 1)+1;
```

The random matching is a little bit more complicated. We first create a ‘random subject variable’, called `RandomOrder`. Each subject receives randomly one of the values 1 to N in this variable. We need the following programs:

```
RandomNumber = random();
```

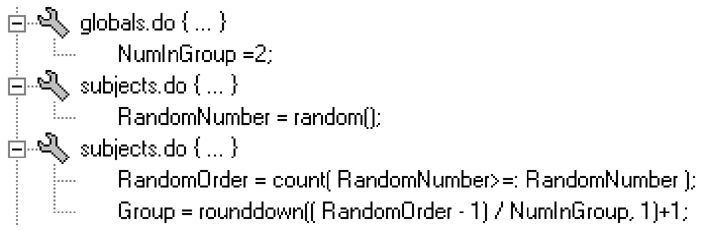
`RandomNumber` contains a value between 0 and 1. If we need not too many random values, we can assume that they are all different. The next statement, we have to place in a new program!

```
RandomOrder = count( RandomNumber>=: RandomNumber );
```

This determines the rank of the random number of the subjects – exactly what we want. We have no ties because the random numbers are different. Now we can determine the matching as above:

```
Group = rounddown( (RandomOrder – 1) / NumInGroup, 1)+1;
```

So, the following programs could be placed in the background:



```
globals.do { ... }
    NumInGroup =2;
subjects.do { ... }
    RandomNumber = random();
subjects.do { ... }
    RandomOrder = count( RandomNumber>=: RandomNumber );
    Group = rounddown(( RandomOrder - 1) / NumInGroup, 1)+1;
```

3.4 Screen layout

So far, you know how input is made with the help of input items in standard boxes and you know how to display variables. In this chapter an overview of more sophisticated layouts is given. In the *real* 'battle of the sexes' game, for instance, the players can choose between 'boxing' and 'opera'. In the implementation above, the subjects had to enter a number for their choice. In this section, we will present how input can be made with different forms of user interface elements such as text input, radio buttons, check boxes, and sliders. Furthermore, we show how information can be displayed and arranged on the screen.

3.4.1 Layout in the small: the item

Items are used for displaying and reading in variables. An item contains the name of the variable and information on how to display it. We call an item an *input item* if the checkbox Input is checked. In this case, subjects must make an entry. We call an item an *output item* if the checkbox Input is not checked. In this case, a variable will be displayed. Variables can only contain numbers. However, such a number can be displayed in different forms. These forms are defined in the Layout field of the item dialog. If a number, a variable, or an expression is entered here, then the value of that field determines how the variable is rounded when it is displayed. If, for instance, 0.2 is entered in Layout, and the variable contains the value 52.31, then 52.4 is displayed because 52.4 is the multiple of .2 nearest to 52.31. The value of the variable can also contain a code for displaying text, e.g., in the battle of the sexes game 1 could mean boxing and 2 could mean opera. We can display these words by using a *text layout*. In the Layout field, we enter the text:

```
!text: 1 = "boxing" ; 2 = "opera" ;
```

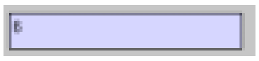
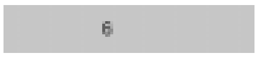
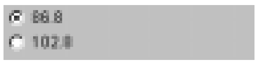
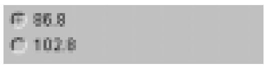
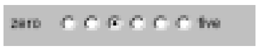
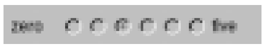
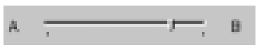
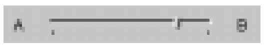
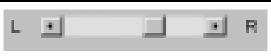
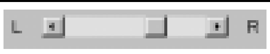
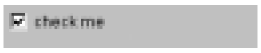
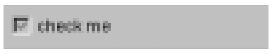
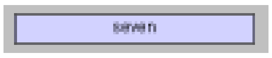
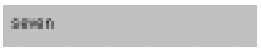
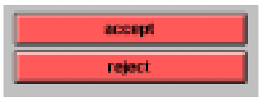
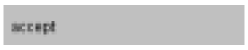
The exclamation mark indicates that the field does not contain simply a number (or an expression that represents a number). The word `text` is the form of output to be displayed. It instructs z-Leaf to display one of the words "boxing" or "opera" (without the quotations marks) depending on the value of the variable. If the value is 1, then boxing is displayed; if the value is 2, then opera is displayed. (To be precise, if the value is nearer to 1 than to 2, then boxing is displayed; otherwise, opera is displayed.) There are other options than `text`. The `radio` option allows you to display radio buttons. So, the following layout

```
        !radio: 1 = "boxing" ; 2 = "opera" ;
```

displays two radio buttons labeled with boxing and with opera. Depending on the value of the variable, one of the two radio buttons is selected. These kinds of layout options can also be used for input items. For an input item with the `text` layout as above, the input has to be entered in textual form. If "boxing" is entered, the variable gets assigned a value of 1. If "opera" is entered, the variable gets assigned a value of 2. If anything else is entered, such as "I do not know" or "BoXing", then an error message appears that says that only certain values are accepted. With the `radio` option, two radio buttons labeled with boxing and with opera are displayed. Selecting one of them sets the variable to the corresponding value.

The following options are available: text, radio buttons, line of radio buttons, check box, slider, horizontal scrollbar, and push buttons as shown in the following table. All options except the push button option can be used for input items and for output items. (The push button option can only be used for input items because a pushbutton cannot be displayed differently as selected or unselected.) How the different options are programmed is explained in the Reference Manual.

Examples of item layouts

Layout	input variable	output variable
`2`		
`!radio: 1 = "86.8"; 24 = "102.8";`		
`!radioline: 0="zero";5="five"; 6;`		
`!slider: 0 ="A"; 100= "B"; 101;`		
`!scrollbar: 0="L";100= "R";101;`		
`!checkbox:1="check me";`		
`!text: 2 = "two"; 3 = "three"; 5 = "five"; 7 = "seven";  11 = "eleven";`		
`!button: 1 = "accept"; 0 = "reject";`		

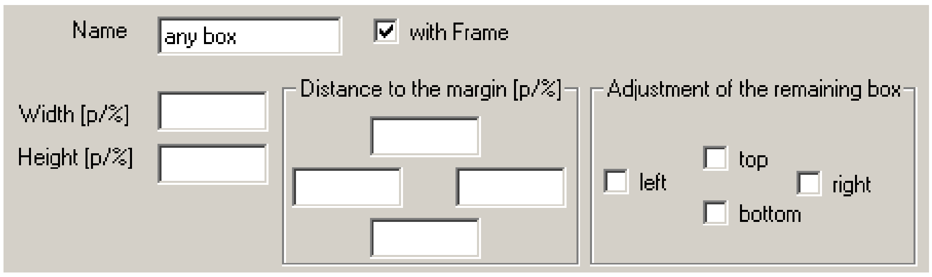
3.4.2 Layout in the large: Screen design and boxes.

In experiments, subjects work for only one to two hours at the computer. They cannot gain much experience and should be able to understand screens very quickly. For that reason, the screen layout of z-Tree is rather static. The screens are built of fixed rectangular areas called **boxes** that can be placed freely on the screen. Besides the standard box you are already familiar with, there are other boxes, for instance, a help box for explanations, a header box for information about the current period and remaining decision time, and a history box in which you can display information about earlier periods.

In this section, we first explain how boxes are placed on the screen. Then, we present some of the specialized boxes.

3.4.3 Placing boxes

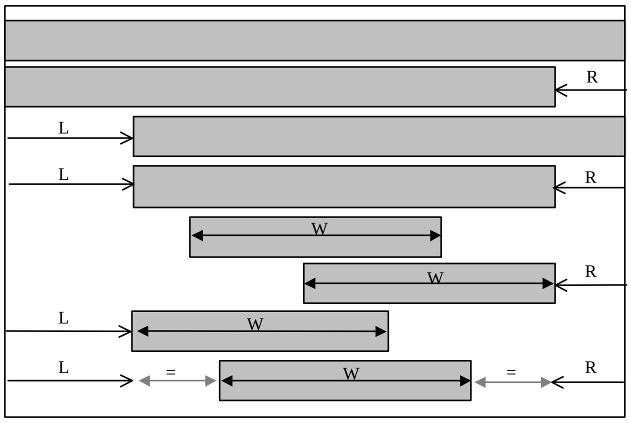
Boxes are positioned one after the other on the screen. By defining size and/or distance to the margin as shown in the figure below, you can determine where the box appears. Any size can be given in screen pixels (p) or in percent.



Name: any box
[✓] with Frame
Width [p/%]
Height [p/%]
Distance to the margin [p/%]
Adjustment of the remaining box: [ ] left [ ] top [ ] right [ ] bottom

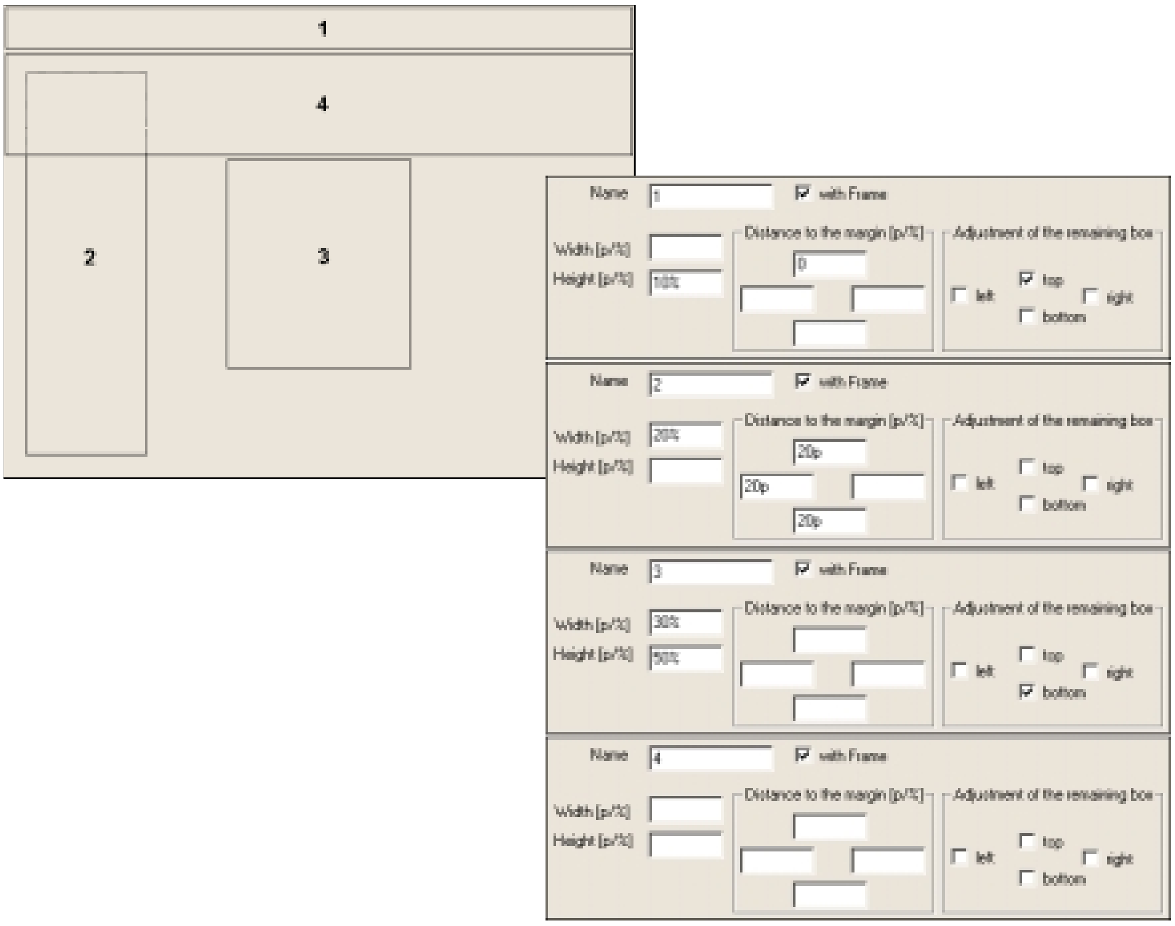
The positioning of boxes is always relative to the so-called ‘remaining box’. At first, the ‘remaining box’ constitutes the whole screen. Later, the ‘remaining box’ is adjusted according to the definitions of the boxes. For example, if you place a header box on the top of the screen and you want to place the rest of the boxes below this header box, then you can cut the top away. You just have to check the top checkbox of the Adjustment of the remaining box field in the box dialog. This means that the top of the remaining box is now the bottom of the header box.

Width, height and distance to the margin are optional. Depending on which fields have been filled, the box is placed within the remaining box. The graph below shows all cases for the fields width (W), distance to left margin (L) and distance to right margin (R).



The name of the box is used for documentation only. In the With frame field you define whether al line is drawn around the box. If there is a frame, then this frame is filled, i.e., if you draw a boy with a frame over another box, then this box will be covered.

In the following figure, we show definition and placement of four boxes. The first box is a header box. We define a distance from the top margin of zero and a height of 10 percent. The rest of the boxes should not intersect this box. So we adjust the remaining box and cut the top. The second box is positioned at the left side of the remaining box with distances of 20 pixels to the margin. We do not modify the remaining box. Hence, the third box where we define width and height is placed centrally with respect to the lower part of the screen. In this box we adjust the remaining screen in a way that the bottom part is cut. The remaining box is now the rectangular area between box 1 and box 3. In the fourth box we specify no positioning. Therefore it fills this remaining box. It in particular intersects box 2. The dashed line is shown here only for illustration. In the actual screen, the part of box 2 that is covered by box 4 is invisible.



Name 1 ☑ with Frame
Width [p/%]
Height [p/%] 10%
Distance to the margin [p/%]: 0
Adjustment of the remaining box: ☐ left ☑ top ☐ right ☐ bottom

Name 2 ☑ with Frame
Width [p/%] 20%
Height [p/%]
Distance to the margin [p/%]: 20p, 20p, 20p
Adjustment of the remaining box: ☐ left ☐ top ☐ right ☐ bottom

Name 3 ☑ with Frame
Width [p/%] 30%
Height [p/%] 50%
Distance to the margin [p/%]
Adjustment of the remaining box: ☐ left ☐ top ☐ right ☑ bottom

Name 4 ☑ with Frame
Width [p/%]
Height [p/%]
Distance to the margin [p/%]
Adjustment of the remaining box: ☐ left ☐ top ☐ right ☐ bottom

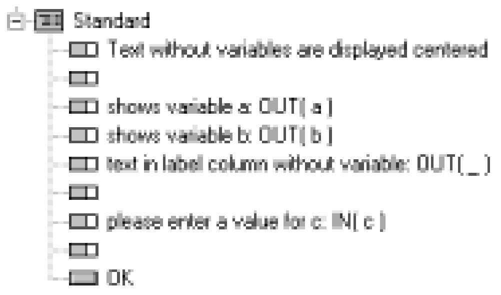
3.4.4 Basic box types

In this section, we present the basic box types: In these boxes you display texts and data from the `subjects` table.

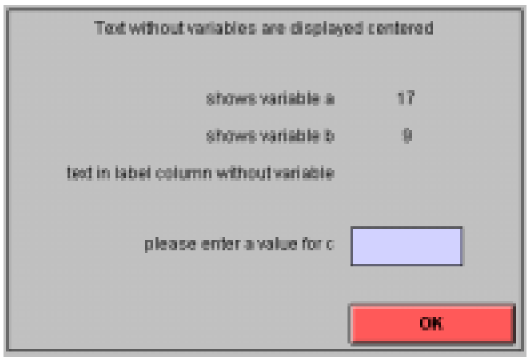
Standard Box

In a standard box, variables of the `subjects` table may be displayed or entered. The items are displayed from top to bottom. The window is divided into a label column (left) and a variable column (right). The variables are always displayed in the variable column. The label always appears in the left column if the variable is defined or if the variable consists only of an underscore (“_”). If the variable is empty, the label is regarded as a title and is written centered over the whole window. If the label consists of more than one line, then it is aligned to the left. If an item is completely empty, it generates a blank line.

The following example shows the definition and the resulting layout of a standard box

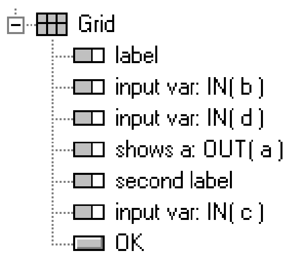


```
Standard
├ Text without variables are displayed centered
├
├ shows variable a: OUT( a )
├ shows variable b: OUT( b )
├ text in label column without variable: OUT( _ )
├
├ please enter a value for c: IN( c )
├
└ OK
```

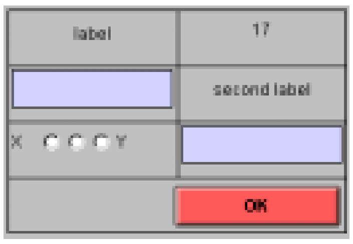

Text without variables are displayed centered
shows variable a
17
shows variable b
9
text in label column without variable
please enter a value for c
OK

Grid Box

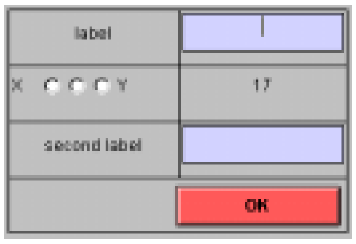
Like the standard box, the grid box can contain items. Unlike to the standard box, items in a grid box are displayed in a table. Each item belongs to one cell in the table. If the item contains a variable, this variable is displayed. Labels are only used for error messages that convey to the subject in which field he or she has made a mistake. If an item does not contain a variable, then the label of the item is displayed. In the grid box dialog, you define the number of rows and columns and the order in which the items are filled in. You can fill the items column by column or row by row. The following example shows the layout if the items are filled in column by column or row by row.



```
Grid
├ label
├ input var: IN( b )
├ input var: IN( d )
├ shows a: OUT( a )
├ second label
├ input var: IN( c )
└ OK
```



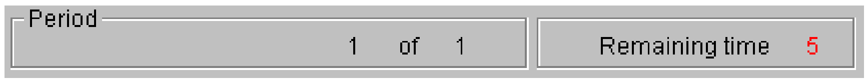
column by column



row by row

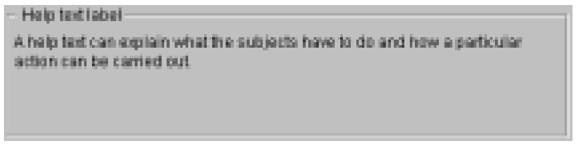
Header Box

In the header box, period number and time are displayed as shown in the following figure. All information is optional and can be defined in the header box dialog.



Help Box

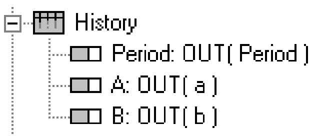
The help box displays text within a box. The size of the text is not restricted to the size of the box. If the help text is too long to appear in the area reserved for the help box, then a scrollbar appears. The help box can be labeled. The following figure shows an example.

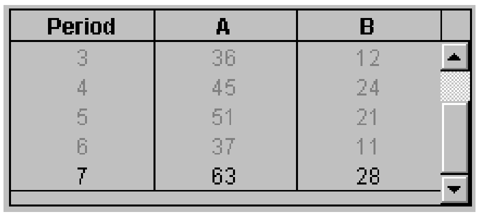

Help text label

A help text can explain what the subjects have to do and how a particular action can be carried out

History Box

In a history box, the results from previous periods are listed in a table. A label row contains the labels. If the table is too long, a scrollbar appears. The current period appears at the end of the table. The scrollbar is adjusted in such a way as to make this line visible at the beginning.


```
History
    Period: OUT( Period )
    A: OUT( a )
    B: OUT( b )
```

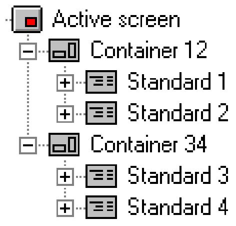


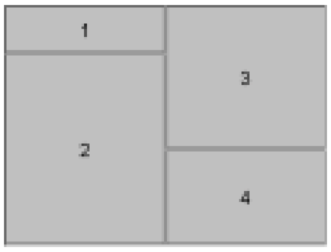
Period	A	B
3	36	12
4	45	24
5	51	21
6	37	11
7	63	28

Container Box

If you place many boxes onto the screen, you may lose track of things. To avoid confusion, you can use container boxes to structure the screen elements. A container box has, besides the frame, no visual representation on the subjects' screens. It only defines areas of the screen within which you can place boxes.

Container boxes also make it possible to define determinants of your screen layout at fewer places. So, if you change the layout, you have to change it at relatively few places. In the example below, we would like to define the width of box 1 and 2 at only one place. With container boxes, this is easy. We define two container boxes, 12 and 34. We define the width in the container box 12 and cut-off the remaining box at left. Now container box 34 fills the region on the right. In boxes 1 and 3 we define the heights and cut the remaining box off at the top. In this way we have not entered a size specification at more than one place, and even if we should change something, the structural appearance of the screen remains the same.





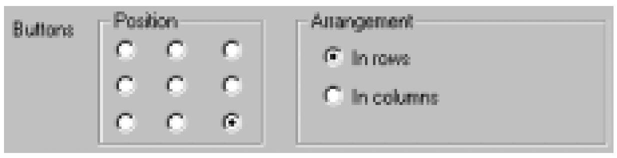
If we move a box from one place to another inside the stage tree, it is moved *after* the box where the mouse button is released. By moving a box over the *icon* of a container box it is moved *into* the container box and positioned at its beginning.

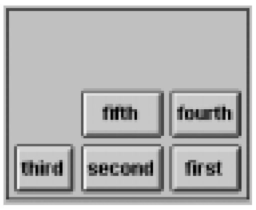
Calculator Button Box

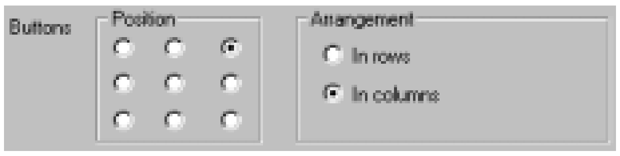
Defines a button as shown below by which the Windows NT Calculator can be called up. It serves as a substitute for subjects who have forgotten their calculators.

3.4.5 Button Placement

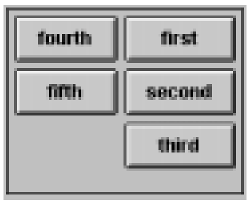
To confirm input and to conclude a stage, you use buttons. They can be placed into standard boxes and into grid boxes. By default they are placed at the bottom right corner. If you prefer another place in the box, then you can choose another option for Button position in the box dialog. If you have more than one button, then they are placed in a row at the bottom. If necessary, a second or more rows are created. In the box dialog, there are two fields where you can modify the button positioning. You can define where the first button is placed (Buttons/Position) and where the subsequent buttons are positioned (Buttons/Arrangement). In the following figure, we show how five buttons are placed with three different option combinations:

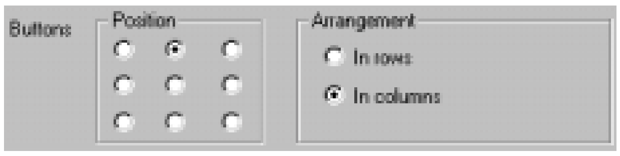

Buttons
Position
Arrangement
[✓ In rows]
In columns



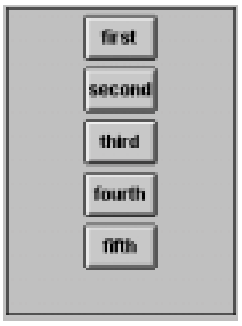


Buttons Position: ○ ○ ● / ○ ○ ○ / ○ ○ ○ Arrangement: ○ In rows [✓ In columns]





Buttons Position: ○ ● ○ / ○ ○ ○ / ○ ○ ○ Arrangement: ○ In rows [✓ In columns]



If you want to leave space in the size of a button without displaying a button, then use an underscore character (_) as the name of the button.

3.4.6 Background layout

On every screen, the boxes in the BACKGROUND are automatically placed first. For each screen, you can determine whether you want to use these background screens. This option (Background screen is being used) is available in the screen dialog. If this field is checked, then the windows of the background are placed first.

3.4.7 Insertion of Variables into the Text

Variables can also be inserted in the label and in the layout field of items, as well as in help and message boxes. Thanks to this you can, for instance, display a formula with the values inserted. Assume that we want to write “income = 23.5 points” where 23.5 is the value of the variable `Profit`. So that the variable does not stand alone on the right hand side, it needs to be integrated into the text. The example above is entered in the Label field in the following manner:

```
<>income = < Profit | 0.1 > points
```

The string “<>“ at the beginning of the text indicates that there might be variables in the text. The variable and its layout appear in smaller/greater brackets, separated by a vertical line. In our example, 0.1 is the layout. The value of profit is therefore given to one decimal place.

The option `!text:` is also allowed as layout. This works in exactly the same way as with output items. Example:

```
<>Your income is < profit |!text: 0="small"; 80 = "large";>.
```

Depending on whether the profit is nearer to 0 or to 80, either “Your income is small” or “Your income is large” is displayed.

Variables can also be inserted into the strings of the “text” option. There you need not begin the string with a second “<>“. If you want to define a layout in which, for negative values, text is written but, for positive values, the number (to two decimal places) is written, you write

```
<>!text: -1 = "negative"; 1 = "<Profit|0.01>";
```

This form has the disadvantage that you also need to write the name of the variable into the layout field. However, you can omit the name of the variable. In this case the variable of the *item* is automatically shown:

```
<>!text: -1 = "negative"; 1 = "< |0.01>";
```

This also works with variables inserted in the text. In the following text, the value of `Profit` is inserted at the place of `<|0.01>`:

```
<>Your profit is < Profit |!text: -1 = "negative"; 1 = "< |0.01>">
```

If you wish to insert the symbol < in the text, the symbol should be duplicated so that it is not confused with a variable expression.

Warning: Be careful when you use this option for items that can change their value. The width of an item is calculated when the screen is first displayed and it is not modified afterwards. So, if the value of a variable changes from 1 to 20, only 2 might be displayed. To avoid this problem, you can choose a sufficiently wide first value or place the item into a box with other items that are wider.

Note further that labels are evaluated only once, at the beginning of the stage. Variables in labels are not updated.

3.4.8 Text-formatting with RTF

In labels of items, in help boxes, and in message boxes, texts formatted with RTF can also be entered. The RTF format begins with "{\rtf " (with a blank space at the end) and ends with "}". In between is the text to which formatting instructions can be added. Formatting instructions begin with "\" and end with a blank space. If a formatting option is supposed to apply only to a certain range, Then you can place this range in curly brackets.

Examples

normal font size, **bold**, no longer bold	`{\rtf \fs18 normal font size, \b bold, \b0 no longer bold }`
Text *italic* no longer italic new line	`{\rtf \fs18 Text {\i italic} no longer italic \par new line}`
One word in ochre, rest in black	`{\rtf {\colortbl;\red0\green\blue0;\red255\green100 \blue0;} \fs18 One word in \cf2 ochre\cf0 , the rest in black.}`

For more complex operations it is best to format the text in a word processor and then export it as RTF. However, if you make the RTF code by hand, it will be shorter and much easier to read.

The insertion of variables is carried out *before* the interpretation of RTF. This makes conditional formatting possible as in the following example: When the variable `Bold` is 1, "hallo" should be shown in boldface but otherwise in plain text:

```
<> {\rtf <BOLD |!text: 0="";1="\b ";>hallo}
```

3.5 Sequential games

In a sequential game, not every subject has the same decision structure. In an ultimatum game, for instance, only the proposers decide what to offer and only the responders accept or reject. Furthermore, you may want to allow simultaneous entry of these different decisions. If you apply the strategy method, then proposers and responders should be able to make their entries simultaneously. However, they make their entries in different screens.

In z-Tree, every treatment is defined as a linear sequence of stages. However, it is not necessary for all subjects to go through all stages and it is possible that not all subjects are always in the same stage. How this is achieved is explained in this section.

3.5.1 Sequential moves

In z-Tree, the variable `Participate` in the `subjects` table determines whether a subject enters a stage or not. If this variable has a value of 1 then the subject enters that stage, i.e., the corresponding active screen appears on the subject's screen. Before the programs in a stage are executed, this variable is set to 1. So, if you do nothing, then all subjects enter that stage. If, however, this variable is zero, then the subject does not enter the stage. The subject automatically proceeds to the waiting state of that stage. The waiting screen of this stage, though, is *not* shown.[4] How can you change the value of `Participate`? This variable is a normal variable of the `subjects` table and you can set it to zero or one depending on a condition.

In the ultimatum game example, the proposers move first, making a decision on their offers. After this, the responders decide whether or not they want to accept the offers or not. Therefore, we define a PROPOSER DECISION stage and a RESPONDER DECISION stage. Only the proposers go through the former, and only the responders go through the latter. To this end, we write the following line in a program of the `subjects` table in the PROPOSER DECISION stage:

```
Participate = if( Type == PROPOSERTYPE, 1, 0);
```

This line sets the variable `Participate`. If it is zero, the subject does not go through this stage. He or she directly reaches the waiting state of that stage, without the display changing. To the subject, it looks as if he or she were still in a previous stage. For the RESPONDER DECISION stage, we use the program

```
Participate = if( Type == RESPONDERTYPE, 1, 0);
```

It is important to know that all programs at the beginning of a stage are executed for *all* subjects. Only when all programs are finished, is the variable `Participate` checked. The variable `Participate` does not determine program execution. It determines only what screens are shown.

[4] This is necessary because you may want to omit more than one stage. In this case the change from one waiting stage to the next should be invisible to the subjects.

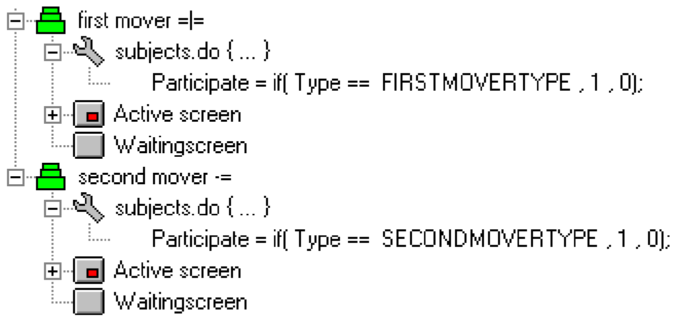
Tip: Always write the Participate statement in the form `if( ConditionForEntering, 1, 0)`. If you do not follow this convention, then you always have to pay attention to the second and third arguments of the if statement which can easily be overlooked.

3.5.2 Simultaneously in different stages

In this section, we show how to program a simultaneous stage. Consider the example of an ultimatum game with the strategy method or any game in which different types receive different feedback – for instance in the profit display.

In each period, the subjects go through all stages, one stage after the other. In each stage, they first arrive at the active state of the stage. In this state, the subjects see the active screen of this stage. In the clients' window, the active state of a stage is designated as "*** stage name ***". The active screen is left by means of an OK button or a time-out. When this happens, the waiting screen appears and the subject arrives at the waiting state of that stage which is designated as "- stage name -". Whether or not the subjects can begin with the next stage depends on how the options are set in the dialog of the next stage. The first of these options is the Start option. It determines the precondition for the subjects to enter the stage. The first two options are the options most often used. Wait for all is the default. If it is the value of the start option, then the subjects cannot enter the stage until all subjects have finished the previous stage. If the option is set to Start if possible, then there is no restriction at all. So, as soon as a subject has finished the previous stage, she can enter the stage. The option Start if... allows you to enter a condition. This feature is only needed for complex move structures. Most experiments can be programmed with the first two options.

The start option allows you to define stages that are executed simultaneously. Consider a two stage sequential game where the strategy method is applied, i.e., the second mover has to make a decision for every possible choice of the first mover. In this case, the second and the first movers can make their decisions simultaneously. We define this in z-Tree as follows: We define two stages, one for the first mover, and one for the second mover. We set the `Participate` variable in the two stages so that only the first movers enter the FIRST MOVER stage and only the second movers enter the SECOND MOVER stage. Finally, we select the individual start option in the second stage. So, the second movers will not participate in the first stage and because they do not have to wait to enter the second stage, they enter the second stage (essentially) at the same time as the first movers enter the first stage. The following figure shows this situation. At the end of the stage name, there is a symbol that shows the start option.


```
first mover =|=
  subjects.do { ... }
    Participate = if( Type ==  FIRSTMOVERTYPE , 1 , 0);
  Active screen
  Waitingscreen
second mover -=
  subjects.do { ... }
    Participate = if( Type ==  SECONDMOVERTYPE , 1 , 0);
  Active screen
  Waitingscreen
```

3.5.3 Ending a stage

For each stage, the time available to the subjects is fixed in the Timeout field. The expression entered is calculated in the globals table. If different stages should have the same time-out times, define a variable in the globals table and enter the name of this variable in the time-out field.

After the time set for a stage has run out, the continuation of the game depends on the option set in the stage. The default behavior depends on whether entries have to be made in this stage or not. If the subject does not have to make any entries on this screen, then the stage ends when the time set for the stage has run out, i.e., the subject arrives at the waiting state of the stage and may go on from there. If entries have to be made, the time displayed is simply a guideline. When the time has run out, a message saying "Please make your decision now" appears. However, the game does not continue until the entries are made. If no timeout should occur, you could enter a negative number. The subjects will then have an unlimited amount of time at their disposal. If the time-out is set to zero, then the subjects will arrive at the waiting screen straight away.

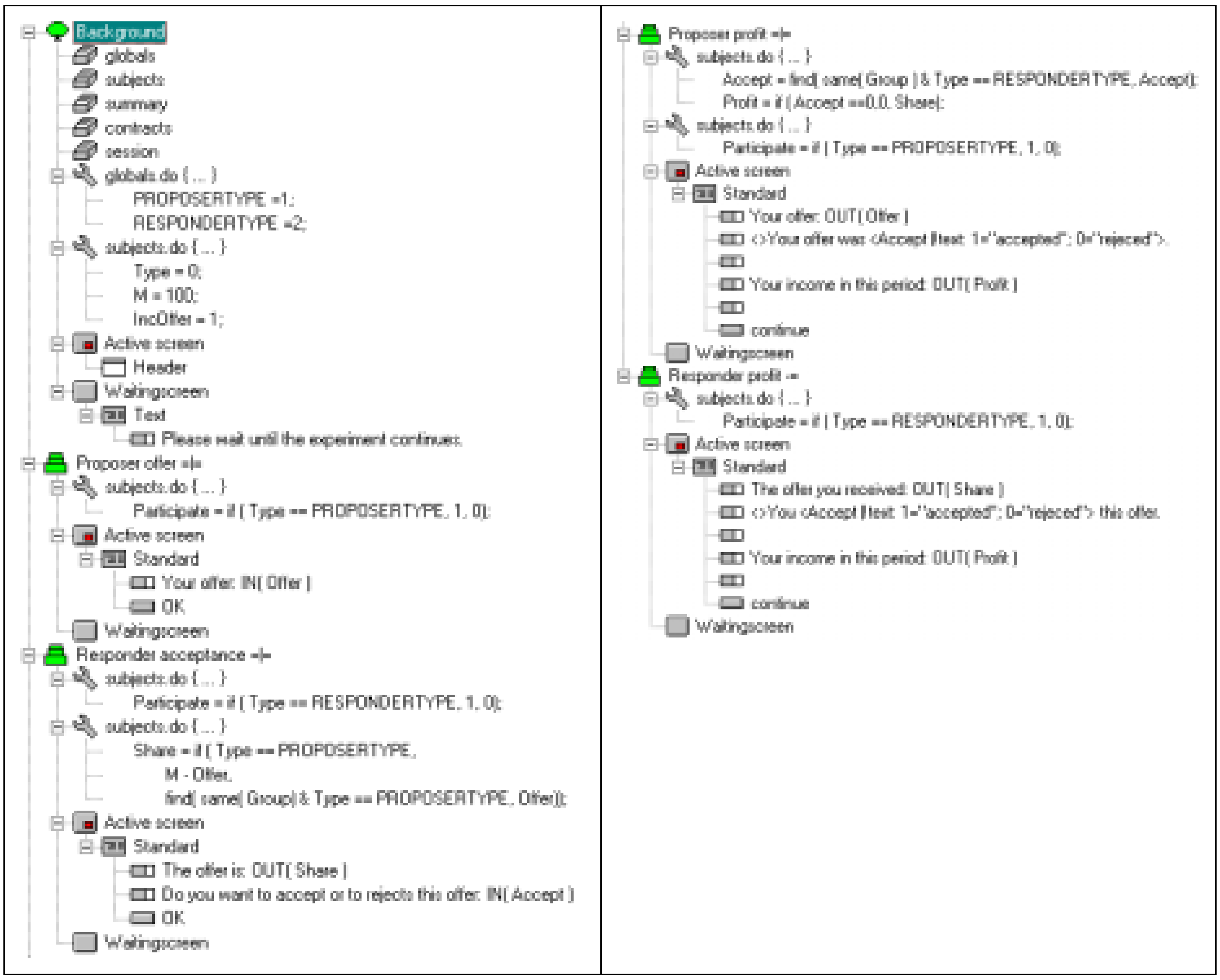
If you do not want to leave a stage automatically, then you set the option to No in the option Leave stage after timeout. If you want to leave the stage even if no input has been made, then you enter Yes in this option.

Warning: If you use the Yes option in Leave stage after timeout, you want to set the value of input variables to the appropriate default value.

3.5.4 Example: Ultimatum Game

Player 1 (the proposer) can propose a division of a pie of 100. Let the proposal be (share1, share2). Player 2 (the responder) can accept or reject this division. If player 2 should accept this proposal, then both players receive their shares. If player 2 rejects, then both receive nothing.

As shown in the following figure, we define four stages for this treatment: The decision of the proposers, the decision of the responders, the profit display of the proposers and the profit display of the responders. We now go through the main steps in programming this experiment.



```
Background
    globals
    subjects
    summary
    contracts
    session
    globals.do { ... }
        PROPOSERTYPE =1;
        RESPONDERTYPE =2;
    subjects.do { ... }
        Type = 0;
        M = 100;
        IncOffer = 1;
    Active screen
        Header
    Waitingscreen
        Text
            Please wait until the experiment continues.
Proposer offer =|=
    subjects.do { ... }
        Participate = if ( Type == PROPOSERTYPE, 1, 0);
    Active screen
        Standard
            Your offer: IN( Offer )
            OK
    Waitingscreen
Responder acceptance =|=
    subjects.do { ... }
        Participate = if ( Type == RESPONDERTYPE, 1, 0);
    subjects.do { ... }
        Share = if ( Type == PROPOSERTYPE,
            M - Offer,
            find( same( Group) & Type == PROPOSERTYPE, Offer));
    Active screen
        Standard
            The offer is: OUT( Share )
            Do you want to accept or to rejects this offer: IN( Accept )
            OK
    Waitingscreen
Proposer profit =|=
    subjects.do { ... }
        Accept = find( same( Group ) & Type == RESPONDERTYPE, Accept);
        Profit = if ( Accept ==0,0, Share);
    subjects.do { ... }
        Participate = if ( Type == PROPOSERTYPE, 1, 0);
    Active screen
        Standard
            Your offer: OUT( Offer )
            <>Your offer was <Accept |!text: 1="accepted"; 0="rejected">.

            Your income in this period: OUT( Profit )

            continue
    Waitingscreen
Responder profit -=
    subjects.do { ... }
        Participate = if ( Type == RESPONDERTYPE, 1, 0);
    Active screen
        Standard
            The offer you received: OUT( Share )
            <>You <Accept |!text: 1="accepted"; 0="rejected"> this offer.

            Your income in this period: OUT( Profit )

            continue
    Waitingscreen
```

First, we define the constants `PROPOSERTYPE` and `RESPONDERTYPE`. We use these constants as names for the numbers 1 and 2. Then, we define the variable `Type`. We set it here to an illegal value and define the correct value in the parameter table. To make the treatment more flexible, we define the size of the pie as a variable. We call this variable `M`. The variable `IncOffer` contains the increments that are allowed when making offers. We allow any integer number and therefore, we set this number to 1. In the PROPOSER OFFER stage, we set the Participate variable so that only proposers make offers.

In the `Responder acceptance` stage, we have to copy the proposer's offer into the responder's row of the `subjects` table. We do this by defining a new variable `Share`. It contains both subjects' shares as suggested by the proposer. As you can see in the formula, it is `M-Offer` for the proposer and it is the offer of the proposer for the responder. To get the offer of the proposer, we use the following find expression

```
find( same( Group) & Type == PROPOSERTYPE, Offer)
```

In the RESPONDER ACCEPTANCE stage, the responder can either accept or reject. We use radio buttons for input, i.e., in the Layout field of the `Accept` item, we write:

```
!radio: 1="accept"; 0="reject"
```

In the PROPOSER PROFIT stage we calculate the profits for both types. The interesting command is in the program where we copy the `Accept` variable from the responder row to the proposer row:

```
Accept = find( same( Group ) & Type == RESPONDERTYPE, Accept);
```

With this line, we overwrite the value in the `Accept` variable for both types. Be cautious with overwriting, you may destroy valuable data. Here, we do not destroy anything. For the responders, we overwrite the value with the old value, i.e., we do not change it. For the proposers, the value of `Accept` had no meaning before. The use of this kind of overwriting makes a program easier to read because it makes a variable that was available only for certain types available to all types in the group.

We place the calculation of the `Participate` variable into a separate program. We have to use this program in two places and if we separate it, we can simply copy and paste the whole program.

The profit display is placed into different stages for proposers and responders, as they each have a different display. In the two stages we set the `Participate` variable in such a way that only the proposers enter the proposers' profit display and only the responders enter the responders' profit display. Finally, we change the start option in the responders' profit display stage to “individual start”. If we omit doing this, the responders only receive the profit display after the proposers have been shown the profit. This creates an unnecessary wait.

3.6 Continuous Auction Markets

3.6.1 Concepts

In the treatments discussed so far, the basic structure was very simple: In each stage the players enter their decision, confirm the decision with a button and then wait until the experiment can proceed, i.e., they wait until they can proceed to a next stage. With this mechanism it is possible to implement any kind of normal form or extensive form game. However, many economic experiments use market institutions that cannot be implemented as games. Consider for instance a double auction. In this market institution, sellers and buyers can make offers. These offers are shown to all market participants. The buyers can accept the sellers' offers and the sellers can accept the buyers' offers. This institution differs from the treatments we have programmed so far in the following ways:

- The subjects do not make a *predetermined* number of entries. We do not know in advance how many offers will be made.
- We do not know in advance in what order offers are made, i.e., we do not know who will be the next who will decide.
- Even subjects who do not make an entry are informed about other players' decisions, i.e., all offers are shown to all subjects.
- The auction is automatically terminated after a certain time

To deal with continuous auctions, we use the following concepts in z-Tree:

- In the stage tree dialog, there is an **option for terminating a stage automatically when the time has expired**. This also changes the default behavior of buttons. Buttons no longer conclude the stage. So, subjects can make an indefinite number of entries.
- The **`contracts` table** is a table that contains an indefinite number of rows. In the `contracts` table we store the subjects' entries.
- There are new box types, the **contract creation box**, the **contract list box** and the **contract grid box**. These boxes allow subjects to make, select, view, and edit offers. By using several boxes, it is possible to do different things on one screen – in one stage. For example, subjects can make offers in one box and accept offers in another box.
- By **placing programs into buttons**, it is possible for subjects to take different *actions* based on the same data. For example, the entry of a price can mean sell for this price or buy for this price.

In the following sections, we go through the development of a simple double auction. In this double auction, sellers can sell a product that has a cost of zero to them and a value of 100 to the buyers. Each subject can trade at most one item. We first show how the entries of the subjects are stored in the contracts table. Then, we show how to make the box for making an offer, the box for displaying the offers and the box for selecting an offer.

3.6.2 A double auction in the contracts table

In an auction, the records of the `contracts` table contain the offers made by sellers and buyers. If an offer is accepted the corresponding record is updated. This information is stored in the variables `Seller`, `Buyer`, `Creator` and `Price`[5]: If a seller makes an offer, the variable `Seller` is set to the `Subject` variable of this subject. (Since this number is unique, we also say that this is the subject ID.) The variable `Buyer` is set to –1 which means that this is an open offer. The variable `Price` is the price the seller enters as her offer. When a buyer accepts a seller's offer, the number `–1` in the variable `Buyer` is replaced by the subject ID of the `Buyer`. The analogous changes to the contracts table are made if a `Buyer` makes an offer and if a `Seller` accepts a buyer's offer. As you have seen, it is easy to map the creation and acceptance of offers in the `contracts` table.

If an offer is accepted, it is not sufficient to mark the offer as accepted. Think of a seller who makes decreasing offers whose lowest offer is eventually accepted. It is clear that the other open offers of this seller are no longer valid. For instance, it is possible that this seller has no more goods to sell. Therefore, we have to delete the outdated offers. Obviously, we do not want to actually delete these offers in the table. We only want to mark them as deleted. We do this by replacing the `–1` in the variable `Buyer` by `–2`. So, this offer is no longer open and is deleted. In this way, we keep all the information about what happened during the auction.

In a double auction, there remains one more problem. If an offer is accepted, we no longer know who made the offer, because in such an offer both the `Buyer` and the `Seller` variables are positive. We solve this problem by using a variable `Creator` that contains the subject ID of the subject who made the offer. We could set this variable when the offer is accepted. However, it is more natural to set it when the offer is made.

The following table summarizes the different situations that occur in a double auction. The value of the variables allows us to reconstruct what happened during the auction.

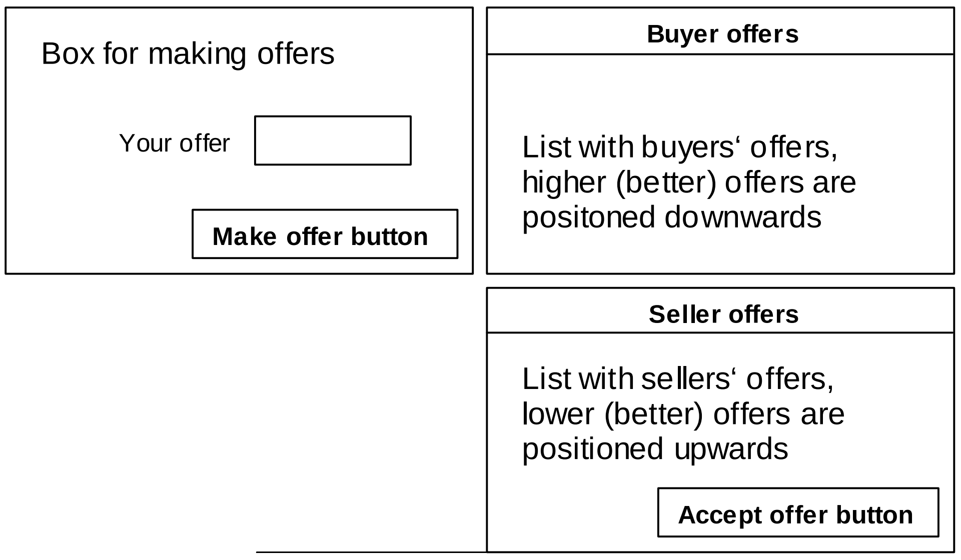
	`Seller`	`Buyer`	`Creator`	`Price`
Open offer of seller	ID of seller	-1	ID of seller	Offer
Accepted offer of seller	ID of seller	ID of Buyer	ID of seller	Offer=Trading price
Offer of seller that is no longer valid	ID of seller	-2	ID of seller	Offer

[5] This is not the only way to program an auction. However, this is very convenient. It is in particular a good practice to always use the same set of variables. It makes it much easier to understand your treatments.

Open offer of buyer	-1	ID of Buyer	ID of Buyer	Offer
Accepted offer of buyer	ID of seller	ID of Buyer	ID of Buyer	Offer=Trading price
Offer of buyer that is no longer valid	-2	ID of Buyer	ID of Buyer	Offer

3.6.3 Preparing the layout for the auction

Before we start to define the boxes, let us discuss the layout. Because the layout of sellers and buyers are very similar, we only discuss the layout for the buyer. The buyer can make offers, he can view the offers of the other buyers and in a third box, he can view the offers of the sellers and accept them. We will arrange the screen as shown in the following figure:


Box for making offers
Your offer
Make offer button
Buyer offers
List with buyers‘ offers, higher (better) offers are positoned downwards
Seller offers
List with sellers‘ offers, lower (better) offers are positioned upwards
Accept offer button

The box for making the buyers' offers and the box of the buyers' offers are positioned next to each other. This is is positioned side by side

3.6.4 Making an offer

In this section, we describe how to design a box for making an offer. Offers are made in a contract creation box. You insert a new contract creation box as any other box from the menu. The dialog of the contract creation box contains some fields that are specific to this type of box. These options are explained in detail in the reference manual. For this example, the options are set properly. The only thing that you have to change is the positioning.

Similar to the standard box, contract creation boxes can be filled with (input) items and buttons. In our contract creation box, the offer must be entered, i.e., we place an input item for the variable Price into the box. To confirm the offer, the subjects have to press a button. This button is also placed into the contract creation box. Now, when a subject presses this button, a new record in the contracts table is created. The data in the input items, i.e., the data in the variable Price is stored in this record. How do we fill the other information into the record? We do not want the subject to have to make an entry for all the information in this new record. This seller should not have to enter the `Seller` variable, for instance. In general, this subject even does not know her subject ID. We want data that has no direct meaning to the subjects to be entered automatically. The solution to the problem consists of a program that is placed into the button. Such a program is executed when the button is pressed. It is easy to write the program that sets the `Buyer` variable to –1:

```
        Buyer = -1;
```

But how do we set the Seller variable? How can we access the ID of the subject who pressed the button? The solution uses the scope operator. The program in the button is executed in a 'scope environment'. When a program runs in the subjects table, we can access the globals table with the scope operator. Similarly, if we run a program in the button of a contract creation box, we can access the record of the subject who pressed the button with the scope operator. (A double scope operator reaches then the globals table.) Now it is easy to set the variables Seller and Creator:

```
        Seller = : Subject;
        Creator = : Subject;
```

Note that we have to define the table in which the program should run. We want to run it in the newly created record of the contracts table. So, in the program dialog, we select the contracts table.[6]

3.6.5 Viewing offers

The open offers are shown in a contract list box. Such a box shows a part of the contracts table. The items that are contained in the box define which columns are shown. A condition defines which records are displayed. Since we want to show all open offers, the condition is

```
        Buyer == -1
```

[6] Different to a program at the beginning of a stage, a program in a button is not necessarily executed for all records of that database.

(Because this is an expression, i.e., something that has a value, it does not end with a semicolon.) The offers should be sorted according to prices. Because seller should decrease the offers, we display decreasing prices. Hence the sorting expression is

```
        -Price
```

The minus sign makes the decreasing order. For increasing order, we do not have to use the plus sign. because increasing order is the default. By the way, you can also interpret the order created by `–Price` as increasing order with `–Price`.

3.6.6 do Statement

Before we proceed, we have to explain a new concept: the `do` statement. With the `do` statement calculations can be carried out in all records of a table. With

```
        do { statements }
```

the program ***`statements`*** is executed for all records in the current table.

As in the table function, it is possible to precede the do-statement with a table name. By doing this, calculations are carried out in the table in question. As in the table functions, you may also use the scope operator here.

Example: Assume the following line is executed in the globals table.

```
        subjects.do {
             Money = :InitialMoney;
             :Done = Subject;
        }
```

It sets, in all records of the subjects table, the variable `Money` to the variable `InitialMoney` in the globals table. Furthermore, it sets the variable `Done` in the globals table to the value of the `Subject` variable in the subjects table. The program runs through the whole subjects table, so finally `Done` will contain the value of the `Subject` variable of the last subject. (By the way, this equals the number of subjects.)

3.6.7 Accepting an offer

To make it possible to accept an open offer, we only have to add a button to a contact list box. As in the contract creation box, we have to put a program into the button. This program marks the offer as bought (and sold). As in the contract creation box, we can access the own record with the scope operator. Therefore, we need the statement

```
Buyer = : Subject;
```

With this line the offer automatically disappears from the list of open offers since now, Buyer is different from –1. However, we are not yet trough. First, we have to delete the offers that are no longer valid, i.e., the open offers of the Buyer and Seller of this offer. After the statement above, we put

```
contracts.do{
     if ( Buyer == :Buyer & Seller == -1 ) {
           Seller = -2;
     }
     if (Seller == :Seller & Buyer == -1 ) {
           Buyer = -2;
     }
}
```

First, with `contracts.do` we proceed through the contracts table. As in a table, the scope operator allows us to access the own record. In this case, it is the record we have selected. So, the condition `Buyer == :Buyer & Seller == -1` is satisfied for any offer of the Buyer who pressed the button that is not yet accepted. We mark these offers as deleted by setting Seller to –2. These offers then disappear from the list of open offers.

In general, we have to put more statements into the program of an accept button: We must update the stock of goods and money of the players to reflect the purchase. We could for instance assume that the variables Goods and Money contain the players' amount of goods and money. In this case, we add the following statements to the program:

```
subjects.do{
      if ( Subject == :Buyer ) {
            Goods = Goods +1;
            Money = Money - :Price;
      }
      if ( Subject == :Seller ) {
            Goods = Goods -1;
            Money = Money + :Price;
      }
}
```

This program ensures that the buyer receives one good and the seller receives the money.

3.6.8 Checking subjects' entries

Often, there are restrictions on the offers that can be made or accepted. It is often not allowed to go into dept. Furthermore, one often requires offers to be improving, i.e., buyers must make a higher bid than the highest current bid. Conditions such as this are stage tree elements that can be defined and placed into buttons.

Consider first the short selling condition. The condition is simply `:Goods>=1`. This means that the seller has at least one Good to sell. The text that should appear is “You have no more goods to sell”. Since there is only a “no” button, the subject cannot disregard the message. When the checker is created, it can be placed into the contact creation box as well as into the contract list box where an offer is accepted.

The improvement rule is implemented as

```
Price > contracts.maximum( Seller == -1, Price )
```

Note that it is necessary to prefix the “do” command with the name of the `contracts` table. In the contract creation box, a program as well as a condition is executed only in the new record not in the contracts table as a whole.

Important: You must put a condition into an accept button that guarantees that the offer selected is still available. It is possible that several subjects accept an offer at (virtually) the same time. However one of the acceptances will be processed earlier. **You must prevent the second acceptance from being processed.** There is a general way to achieve this. Put the condition that the displayed records satisfy into a checker. In our example this is

```
Buyer==-1
```

Use the text “Someone else was quicker” as the message.

Where do the messages appear?

In general messages appear in a new window and the subject has to click OK in order to close the window. This has the advantage that it focuses the subject’s attention but if a client crashes, all the messages appear again and have to be clicked away. As an alternative, you can insert a message box in which the messages are displayed. In this case, the subject does not have to click the messages away but you risk some subjects overlooking such messages.

3.6.9 Auction Stages in detail

We call a stage with an automatic timeout an auction stage. An auction stage forms an auction with all the following auction stages whose start option is “start if possible”. Such an auction is started and ended simultaneously for all subjects. Before the stages start, all programs of these stages are executed. This is necessary in order to determine which subjects will participate in which stages.

Example: Though buyers and sellers in a double auction take part in the same auction, they are shown different screens. To this end, we define two stages: The first, an auction stage for the sellers, and the second, an auction continuation stage for the buyers. These two stages are started simultaneously, the first for the sellers, the second for the buyers. For that we insert the following program lines into the two stages:

```
Participate = if( type == SELLER, 1, 0); // into first auction stage

Participate = if( type == BUYER, 1, 0);
           // into second auction stage
```

Generally, auction stages are concluded when the time has run out. There are two exceptions, however:

- If the variable `AuctionStop` is set to 1 in the globals table, the auction is terminated immediately. This is useful, for example, when all possible transactions are exhausted.

- If the variable `AuctionNoStop` is set to 1 in the globals table, the auction is not concluded even if the time is up.

3.6.10 Creation of New Records in the Program

In the contracts table, new records can also be added with the “new” statement. The syntax is

```
table.new{ statements }
```

New records can only be entered into the contracts table and into user defined tables. The statements in curly brackets can contain initializations of the record. The following example is taken from the program of the subjects table of a certain stage. When a user reaches this stage, a record is added in the contracts table. In this record, the variables Seller, Buyer and Price are fixed. (The value `Buyer` is the Subject ID of the subject for which the program is run.)

```
contracts.new {
     Seller = -1;
     Buyer = :Subject;
     Price = 25;
}
```

3.6.11 Adding quality

Assume the subjects have to specify some property, such as quality, of the good traded after the double auction has ended. If each subject trades only a known number of goods, you can copy the information into the `subjects` table. The subject can then enter this information in the `subjects` table – for instance in an ordinary standard box. There is however an easier way of doing this. In a contract grid box, you can display a part of a contracts table and unlike to the contract list box, input variables can be added for *each* record.

So you can create a contract grid box that displays the records of the `contracts` table that correspond to the trades of a subject. By adding an input item with a variable `Quality`, the subject can enter a quality to each of the trades.

3.7 Posted offer markets

In a posted offer market, one market side, say the sellers, makes offers. The other market side – in our case the buyers – can then, one after the other, select one of the offers. In z-Tree, this sellers' stage is easily implemented with a normal (non auction) stage in which the sellers make their offer in a contract creation box.

The buyers select the contract in a contract list box. Now, to allow buyers to act sequentially, we use the Number of subjects in stage option in the stage dialog. By setting it to At most one per group in stage, only one subject per group may enter the stage and therefore, the subjects enter the stage one after the other. If nothing is specified, the order in which they enter is random. If the variable `Priority` is set, the subjects with lowest value (best rank), enter first.

If a subject has to go through several stages before the next subject may enter the sequence of stages, the second and all following stages must have the "...and in previous stage(s)" option set.

3.8 Clock Auctions

3.8.1 The Dutch Auction

In a Dutch auction, when a good is sold, there is a clock showing a price. This price decreases with time. As soon as one person decides to accept, she get the product at the current price.

To implement this institution in z-Tree, we need a flexible clock. We must be able to define the steps of the price decrease as well as the time interval between price decreases. This can be done with a `later` statement. If you start with a price of 1000 and every 3 seconds you want do decrease the price by 10, you write the following program (for instance into the globals table):

```
Price = 1000;
later ( 3 ) repeat {
     Price = Price – 10;
}
```

The later statement has the following general form:

```
later( expression ) repeat { statements }
```

1. The expression is evaluated. Let t be the value of the expression.
2. If t is smaller than zero nothing more happens.
3. If t is at least 0, then after t seconds, the statements are executed and then we go back to 1.

There is a second form of the later statement for the situation where we want the statements to be executed at most once. In this case, we write:

```
later( expression ) do { statements }
```

Also, in this case the expression is evaluated. If the value t is negative, nothing happens. If the value is greater than or equal to zero, then after the corresponding amount of seconds pass, the statements in the curly brackets are executed.

Application 1: Auction Trial Periods

You plan to conduct an experiment with a double auction. Because the user interface will be rather difficult, subjects should be able to learn how it works. This is best achieved with some trial periods. A problem with trial periods is that they allow interaction between the subjects and uncontrolled learning before the actual experiment starts. To allow for controlled learning, you can simulate the entries of the other subjects in the trial period auctions. This can be done with a sequence of `later` statements.

Application 2: Double auction with an external shock.

Suppose you want to change the economic environment during a double auction. Let p be the parameter you want to change from 100 to 50 after a minute. You write the following program:

```
P=100;
later ( 60 ) do {
     P=50;
}
```

3.8.2 Leaving a stage

Suppose you conduct a Dutch auction with multiple groups. You would like the group to proceed to the next stage as soon as the good is sold. You can achieve this as follows:

First, you initialize a variable `Accepted` in the `subjects` table to 0. Then, you create the later command as above. You show the price in a standard box. In this box there is a button and in this button you put the following program.

```
if ( sum ( same (Group), Accepted ) == 0) {
                  // no other player else was quicker
      Accepted = 1;
      subjects.do {
            if ( same (Group ) ) {
                  LeaveStage =1;
            }
      }
}
```

The variable `LeaveStage` allows you to force subjects to leave a stage. In the program above, all subjects in the group of the subject who accepted the price immediately leave the stage.

3.9 Advanced Concepts

3.9.1 Games depend on previous results

If you want to program an auction market, you may want the subjects to keep their endowments of money and stock from period to period. In z-Tree, the subjects database is set up freshly in every period, so the information about earlier periods is not available directly. However, the tables of the previous period can be accessed with the prefix OLD. So if you want to want to copy a variable `Stock` from the previous period, you write

```
if (Period >1 ) {
     Stock = OLDsubjects.find( same( Subject ) , Stock );
}
```

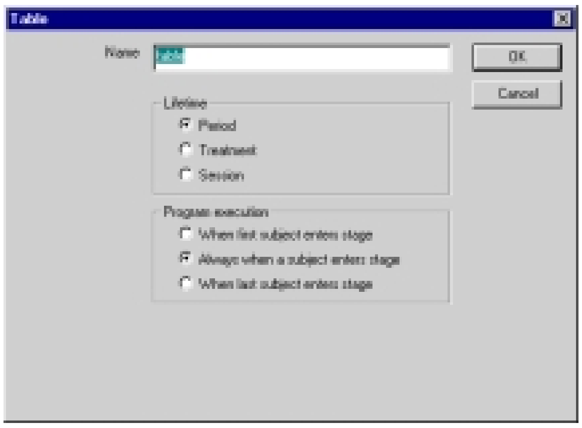
There is no way of accessing data from earlier periods.

3.9.2 Definition of new tables

In z-Tree it is possible to define more tables that the standard subjects, globals, summary , contracts, and session tables. You can use these for different purposes:

- Multiple contracts-like tables for different markets.
- Random period payment: A table that contains the profits made in each period. This table can cover more than one treatment.
- Storing a discrete function.
- Using a calculator and an auction simultaneously.

To use a new table, you have to define it at the beginning of the background with the menu item "New Table..." in the "Treatment" menu. A dialog opens where you can enter the properties of the table.



Table

Name: table

OK

Cancel

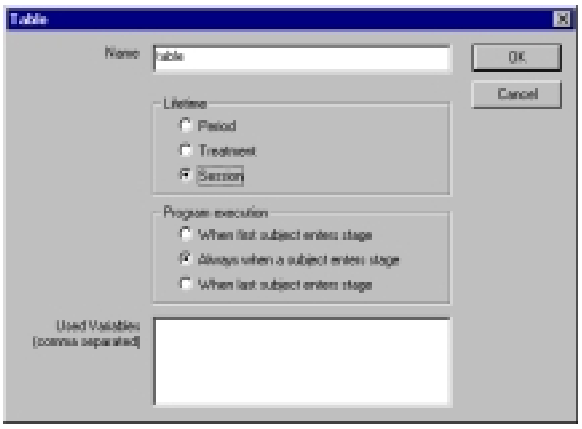
Lifetime
- [✓ Period]
- Treatment
- Session

Program execution
- When first subject enters stage
- [✓ Always when a subject enters stage]
- When last subject enters stage

Lifetime: If a table's lifetime is "**Period**", the table is reset whenever the end of a period has been reached, i.e. all records of non standard tables are deleted. (Of course, after they have been saved.) The subjects, globals, and contracts tables have a period lifetime. As in the subjects, globals, and contracts tables, the data that was in a table x in the previous period can be accessed in the table called OLDx.

Tables with lifetime "**Treatment**" are reset after the end of the treatment. They do not change after the end of a period.

Tables with lifetime "**Session**" are never reset. Nevertheless, they are stored at the end of each treatment. To be able to access variables defined in a treatment that was executed before, you have to declare the variables in the field "Used Variables".



Table

Name: table

OK

Cancel

Lifetime
- Period
- Treatment
- [✓ Session]

Program execution
- When first subject enters stage
- [✓ Always when a subject enters stage]
- When last subject enters stage

Used Variables (comma separated)

Program execution: If subjects do not enter a stage simultaneously, programs in a table can be executed whenever a subject enters, or only when the first enters the stage, or only when the last subject enters the stage. You can define the option for your databases. The programs in the globals table are executed for the first subject, programs in the summary table for the last subject. Programs in the subjects and in the session table are executed for each subject who enters the stage.

3.9.3 Conditional Execution of Statements

If you wish to carry out something only under certain conditions, you can utilize the if-statement. It may be utilized in the following forms:

```
if ( condition ) { true_statements }
if ( condition ) { true_statements } else { false_statements }
```

The condition is evaluated first. If it is TRUE, the instructions in the true_statements are carried out. If the condition is FALSE, nothing is done in the first case, and in the second case the instructions in the false_statements are carried out.

Example:

```
if ( type == FIRM ) {
     Profit = v * e - w;
     othersProfit = w - c;
}
else { // type == WORKER
     Profit = w - c;
     othersProfit = v * e - w;
}
```

You may also carry out conditional calculations by using the `if` *function*. However, carrying out many calculations in this way requires constant repetition of the condition. This is inefficient and can become confusing.

Note: The if-statement makes it possible to calculate a variable for some records and not for others. This can result in undefined cells.

Consider a situation where you have 4 types of players and for the different types you have different profit functions. You can program this with a nested if statement as follows:

```
if ( type == 1 ) {
     // first calculation
}
else {
     if ( type == 2 ) {
           // second calculation
     }
     else {
           if ( type == 3 ) {
                 // third calculation
           }
           else {
                 // fourth calculation
           }
     }
}
```

It is not easy to keep track of all the necessary closing brackets. With elsif, there is an easier way to express the above program (there is no e in elsif):

```
if ( type == 1 ) {
     // first calculation
}
elsif ( type == 2 ) {
     // second calculation
}
elseif ( type == 3 ) {
     // third calculation
}
else {
     // fourth calculation
}
```

3.9.4 Loops: while und repeat

There are two forms of loops in z-Tree:

```
while( condition ) { statements }
```

The condition is evaluated and as long as the condition returns TRUE, the statements are executed and the condition is reevaluated.

```
repeat { statements } while ( condition );
```

The statements are executed. Then, the condition is evaluated and as long as the condition returns TRUE, the statements are executed again and the condition is reevaluated. With the repeat statement, the statements between the curly brackets are executed at least once.

Be cautious. Loops are frequent sources of program bugs. For example, if the condition in a loop never returns FALSE, then the loop runs forever. You can leave a loop with <Ctrl>-alt>-[F5]. This, however, results in undefined results in z-Tree and should only be used when testing a treatment.

Another way to program loops is by using **iterators**. An iterator creates a new small table that contains one variable. When a table function or a do-statement is applied to such an iterator, it corresponds to a loop over the values contained in the table.

An iterator has the syntax:

```
iterator( varname )           // runs from 1 to number of subjects
iterator( varname, n)         // runs from 1 to n
iterator( varname, x, y)      // runs from x to y
iterator( varname, x, y, d)   // runs from x to y with steps of d.
```

This syntax defines a table**.** This table has a variable *varname* and records for which the variable *varname* is filled with, in the last case, for instance, the values x, x+d, ... y.

Example:

```
// two ways of calculating: squareSum = 1+4+9+16+25 = 55
squareSum = iterator( i, 1, 5).sum( i*i );
// or
squareSum =0;
iterator(i,5).do {
     :squareSum = :squareSum + i*i; // iterator has its own scope!
}
```

3.9.5 Complex move structures

It is possible to implement any kind of move structure in z-Tree. For more complex move structures, one uses the start option Start if... We show how to use this option in a treatment with the following decision structure: There are three players, 1, 2 and 3. Players 1 and 2 move first. Then, player 3 moves. However, player 3 gets to know only the move of player 1. So, one would like to allow 3 to decide as soon as player

A has decided. This would be reasonable, if the decision of player B were much more time consuming than the decision of player A.

Let `Type` be the variable that contains the type. Let `DecisionA` be the input variable of player A. Let `DecisionB` and `DecisionC` be the input variables of player B and C. Assume that the variable must be positive so that if we initialize the variables with –1, we know who has made her entry. We define 3 stages, stageA, stageB, and stageC. In stageA, we wait for all. In stageB, we start immediately. In stageC, we start if

```
count( Type ==1 &  DecisionA  >0) == count( Type ==1 )
```

With this condition, the players 3 start when all players 1 have finished. With

```
count( same (Group) & Type ==1 &  DecisionA  >0) == 1
```

player 3 starts when her players 1 has finished. (Of course, we also have to set the `Participate` variable accordingly.)

3.9.6 Treatments of Indefinite Length

If you want to apply a random stopping rule, the treatment will have no definite length. This can be implemented in z-Tree with the variable `RepeatTreatment`. You define a one period treatment. At the end of the period you decide whether to continue the treatment or to stop. This decision is written into the variable `RepeatTreatment` in the `globals` table. If the variable is not defined, the treatment is terminated. This is the normal case. If you define the variable `RepeatTreatment`, it depends on the value of the variable whether the treatment is terminated or whether it is repeated once again. If the value is smaller than or equal to zero, the treatment stops. If the value is greater than zero, the treatment is repeated.

If the treatment is repeated then the period counter is incremented as if there was one treatment. This is implemented by modifying the number of (trial) periods. So, if you repeat a treatment with three periods, the first treatment has 3 Periods and no trial periods. In the second run of the treatment, the treatment's number of periods is changed to 6 and the number of trial periods is changed to -3.

3.9.7 Example: Strategy Method (using arrays)

Let us consider a treatment in which we wish to use the strategy method for a simple game. There are two types of players: A and B. The A players select a number `a` between 1 and 10, the B players select a number `b` for every possible `a`, i.e., they select b1, b2, b3,...b10. Let the profit be the absolute difference between the two numbers `a` and `b`. The formula for the calculation of profits looks something like this:

```
a = find( type == A, a);
Profit = if( a == 1, abs(a-b1),
         if( a == 2, abs(a-b2),
         if( a == 3, abs(a-b3), ...  )));
```

This is very long-winded.

In such cases, it is wisest to define an array, i.e., an indexed variable. In an index set, every finite, equidistant subset of numbers is allowed. The index set needs to be defined by an array instruction before the first use of the array. The array instruction can have the following forms:

```
array arrayvar[ ] ;          // defines an array with indices from 1 to number of subjects
array arrayvar[ n ];         // defines an array with indices from 1 to n
array arrayvar[ x, y ] ;     // defines an array with indices from x to y
array arrayvar[ x, y, d];    // defines an array with indices from x to y with distance d.
```

The access to array elements is carried out with `arrayvar[indexvalue]`. The expression *`indexvalue`* is rounded to the nearest possible index and the corresponding value in the array is fetched or filled.

Example:

```
array p[ 10, 20, 5 ];  // the index set is {10,15,20}
p[ 10 ] = 1;
p[ 15 ] = 5;
p[ 20 ] = 2;
p[ 30 ] = 3;          // sets p[20] to 3
x = p[10] + p[ 20 ];  // x is 4
```

The example introduced at the beginning of this sub-chapter would now look like this:

```
array b[10];      // within a background program
                  // b[1], b[2], etc are entered by the subject
a = find ( type == A, a);
Profit = abs( a - b[ a ]);
```

3.9.8 Turn information on and off

Boxes can be shown and hidden dynamically. There is an display condition option in the box. A box is shown if the condition is empty or if it equals TRUE. Whenever data changes, z-Leaf checks whether a box must be shown or hidden. You can use this feature for instance to control access to information.

3.9.9 Copying data from treatment to treatment with the session table

Consider the following Experiment: First you make a public goods treatment. The income earned in this public goods treatment can then be used in a second treatment in a market game. Because the session table survives the end of a treatment, you can solve the above problem elegantly. In the first treatment, you store the total profit in the variable `Income` in the session table and in the second treatment you can access this variable again.

How you store data in the session table: In programs in the session table, the corresponding record of the subjects table can be accessed with the scope operator. Therefore, in your first treatment, you can enter the following line into a program of the session table:

```
Income = : Total Profit;
```

How you retrieve data from the session table: You can access variables in the session table as in any other table. There is only one problem: Z-Tree always checks that you do not use variables that are not defined earlier in the treatment. Therefore, you have to declare what variables in the session table you will use in a treatment. You cannot define a variable with an assignment statement because this would destroy the value of the variable. Therefore you have to define the variables at the beginning of the treatment – within the session table definition: All tables are listed at the beginning of the background. To define the variable you need to use in a treatment, you double click at the session table icon and enter the variables into the field "Used variables".

4 Questionnaires

4.1 Overview

Each session ends with a questionnaire. Generally, name and address are asked first so that the payment file can be written. This is the file that contains a list of the names and earnings in the experiment. Next, additional questionnaires are inserted. Then a screen that displays the subject's earnings is inserted and finally a good-bye screen appears where subjects get instructions where to get their payment.

A questionnaire consists of a series of **question forms**. One question form displays the individual **questions** in the same way as the items in the standard box of a treatment. However, the questions do not appear vertically centered and adjusted to the screen size. Therefore, question forms may be larger than the screen. If this is the case, a scrollbar appears. The answers in questionnaires are of no consequence. In particular, no earnings can be made in questionnaires. Furthermore, the answers are only saved as text and can not be used in programs.

The question forms are worked through one after the other. When all subjects have answered all question forms they are in the state "ready" and you can start a further treatment or a further questionnaire. It is generally impractical to start several questionnaires one after the other because this leads to unnecessary waits. It is better to integrate all forms into one questionnaire.

The last question form remains on the users' screens until you continue, i.e., until you select a new treatment or a new questionnaire or until you shutdown the computer. Therefore, the final question form must not contain a button.

4.2 Making questionnaires

4.2.1 Address Form

The address prompt asks for the subject's address. The fields "name", "first name" and "button" are compulsory if the address form should appear. The other fields are optional. If the question texts in the optional fields are left empty, the question is not sent to the clients. If the fields "name" and "first name" are left empty, then no form appears. The payment file is written whenever the last subject has completed the address form.

When all subjects have completed the address form, the payment file is written. This is a tab-separated file containing names, computer identification, and earnings in the experiment. This file can be printed and payment of the subjects can be based on this list. In section 4.3, we describe how to produce individual receipts.

4.2.2 Question forms

Question forms consist of a list of questions. The layout of the questions is adjusted with rulers. All question forms except the last must contain a button.

If you have conducted an experiment where subjects played different roles, for instance, there were proposers and responders, then you may also want to use different questionnaires for motivational questions. In the new version of z-Tree, you can *omit* question forms. Hence, if you want to present two different questionnaires to two different types of subjects, you put the question forms for both types into one questionnaire document and omit the forms selectively for the type you want.

The procedure to omit question forms is similar to the procedure to omit stages in a treatment: You set the variable "Participate" to zero if you do not want the subject to fill out that question form. Since there is no subjects table available when you run a questionnaire, the variable Participate has to be set in the session table. You can do this in the program that can be entered in the question form dialog.

Important: In the treatment, you have to copy the type variable (the variable that determines whether the subject is a proposer or a responder) into the session table.

4.2.3 Questions

Questions in questionnaires correspond to items in treatments. In items, display options are defined in the field layout. Here they can be set with radio buttons. The fields are as follows:

Label: Name of the question as it is shown to the subjects. If no variable is set, the label appears as text over the whole width of the screen.

Variable: If it is an input variable, this is the name of the question as it should appear in the data file. If input is not set, it may only be one of the variables described in Section 4.2.4.

Type: Layout of the question.

Text, number: A text field appears into which input is entered. In the case of the number option there is a check to see if it is really a number that has been entered. Furthermore, the check determines whether the number is in the valid range. If the option “Wide” is selected for a text field, the entry field may consist of several lines. The number of lines can be selected.

Buttons: A button is created for every line in the field Options. Of course, only one question with this option is permitted per question form.

Radio buttons: A radio button is created for every line in the field options. These buttons are arranged vertically one on top of the other.

Slider, Scrollbar: The slider and scrollbar make a quasi continuous entry possible. Minimum, maximum, and resolution need to be entered. In the wide layout, a label can be placed at the margins. These labels are entered in the field options. The maximum is always the value at the right border, the minimum is the value at the left border. Maximum may therefore be smaller than minimum.

Radioline: A radio button appears for all possible answers determined by minimum, maximum and resolution. In the wide layout, a label can be placed at the margins. These labels are entered in the field Options. The middle button can be separated from the others by entering a number greater than zero for "distance of central button". The maximum is always the value at the right most button, the minimum is the value at the left most button. Maximum may therefore be smaller than minimum.

Radiolinelabel: Label line for radio button lines. Labels can be placed above the buttons at the margins. The texts for this action are in the options field.

Checkbox: A checkbox is created for every line in the options field. These checkboxes are arranged vertically one on top of the other. The resulting value is a list of all options checked.

Wide: In the wide layout, the entry region for the question appears under the label over the whole width and does not only appear to the right of the label in a second column.

Input: This field determines whether a question is presented to the subjects (input is set) or whether a variable is only shown (input is not set).

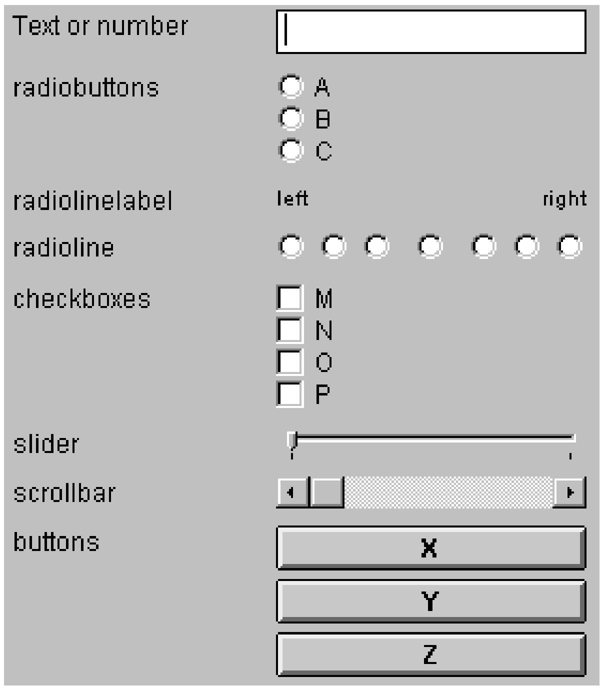
Empty allowed: This is set when the question does not need to be answered.

Minimum, maximum, resolution: When numbers have to be entered, these values are used for checking. With sliders, scrollbars and radio lines these values are used for converting the position into a number.

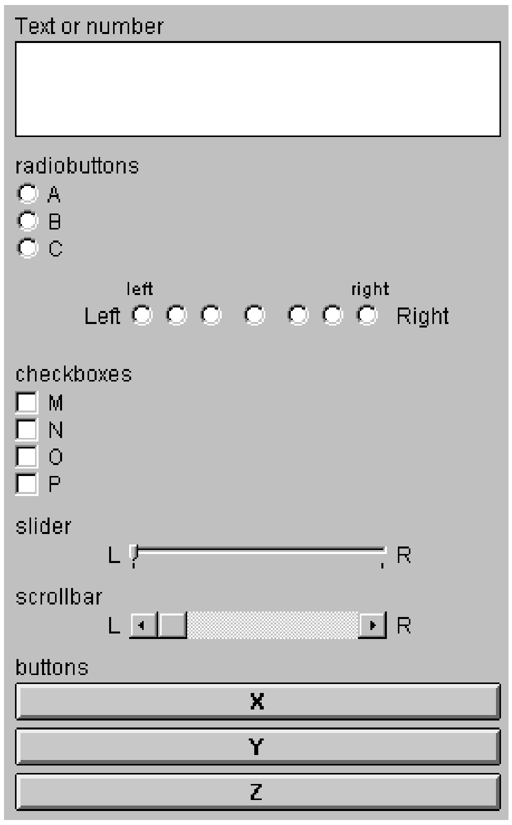
Num. rows: In the wide layout for text entries, this is the number of rows that can be entered.

Options: The labels of buttons, radio buttons and, in the wide layout, of sliders, scrollbars and radio lines.

Example of normal layout (not wide):



Example of wide layout:



4.2.4 Profit Display

If you wish to display the earnings made in the session to the subject, you use a question form where you can also show variables from the session table. These include:

FinalProfit	Sum of all earnings from the treatments
MoneyAdded	Money injected by the subject. (see bankruptcy procedure)
ShowUpFee	Amount of show-up fee.
MoneyToPay	Showupfee + FinalProfit + MoneyAdded.
MoneyEarned	Showupfee + FinalProfit.

4.2.5 Rulers

Rulers are used to set the regions where labels and questions are positioned.

Distance to left margin: The position can be given in points or in percentage of the screen width.

Distance to right margin: The position can be given in points or in percentage of the screen width.

Distance between label and item: This is the horizontal distance between the label and the question. Half of this distance is also kept to the left and right margins.

Size of label: This is the maximum width of the label. When the display is given in a percentage, this refers to the width between left and right margin and not to the width of the whole screen.

4.2.6 Buttons

Buttons conclude question forms. You cannot define more than one button per form. However, this button can be positioned anywhere on the form. The label of a button can be selected by you.

4.3 Running a Questionnaire

At the end of a session, the experimenter always starts a questionnaire with the command "Start Questionnaire" which is described below. This questionnaire must include a prompt for the subject's address (address form). When all subjects have entered their addresses, z-Tree writes the payment file. You may also insert a questionnaire at another point in the session. However, as questionnaires do not contain information regarding the number of subjects, they can be started at the earliest point after the first treatment (e.g., the welcome treatment). If more than one questionnaire contains an address form, the address only needs to be entered in the first questionnaire. In subsequent questionnaires only the payment file is updated.

A questionnaire is started with the command "Start questionnaire". This command is located at the place where the command "Start treatment" is located when a treatment window is open. While subjects are answering the questionnaires the clients remain in the state "questionnaire". The window "subjects table" shows which questionnaire is currently being answered.

4.4 Making Individual Receipts

In this section, we describe how to produce individual receipts based on a mail merge document. In the following, we explain this procedure step by step (for MS Word).

Before the Session starts.

- Make a Mail Merge document. You can use the fields "Subject", "Computer", "Interested" "Name" and "Profit" and the variable you added to the payment file.
- You can open the word processor and open the mail merge document.

When the payment file is written.

- Open the mail merge document (or bring it to the front).
- Open Mail Merge (“Seriendruck” in the German version of Word) in the menu Extras.
- Select in point 2 of the mail Merge manager the option "Open Data Source" and select the payment file. You find it in the directory where you started z-Tree or in the directory you defined with the privdir option. The easiest way to find it is to view the directory in details mode and sort it according to "Modified".
- Select point 3 to generate the merged document.
- Check whether the document is correct and print it.

More information in the payment file

Sometimes you need more output in the payment file. Consider the case where in an individual decision making experiment each subject is paid for a random period. For this decision, you want to use a physical random device just when the subjects get their payment. For that you need more information in the payment file than the cumulative profit. By adding questions to the address screen, you can add any variable of the session table to the payment file. The questions must only be of type "number" and "input" must be unselected.

5 Conducting a Session

5.1 Quick Guide

The running of a session consists of the following steps:

1. preparation of treatments and questionnaires
2. start-up of the experimenter PC
3. start-up of the subject PCs
4. arrival of subjects
5. start of the session and the first treatment
6. observing the course of the session
7. start further treatments
8. conclusion of session with a questionnaire
9. payment
10. switching-off of machines
11. data analysis

While conducting an experiment, we mainly work with the menu “Run”. The order of commands in the run menu corresponds, on the whole, with the course of the session.

5.2 Preparation of Treatments and Questionnaires

Before the session begins, all treatments and questionnaires in z-Tree are defined and saved on the file server. How you define treatments and questionnaires is explained in sections 3 and 4. Treatments are different for different numbers of subjects. Hence, in case you are not sure whether all subjects will show up, you would like to have the possibility of running a session with varying numbers of subjects - depending on how many show up. In simple cases the number of subjects, which is fixed in every treatment, can be changed at the beginning of the session, i.e., before the start of the first treatment. In cases where changing the number of subjects involves more than simply changing this number[2], you have the option of preparing treatments for different numbers of subjects.

[2] Within a Stranger design, for example, the group matching has to be redone.

We recommend that you make a list of the treatments and questionnaires. It should include the names of all of the treatment files for all possible numbers of subjects. Alternatively, you may list how the treatment files are to be adjusted for all possible numbers of subjects.

5.3 Start-up of the Experimenter PC

Switch on the experimenter PC, log in[3] and start "z-Tree". Z-Tree has a window that displays which clients have started up and established contact. This information is displayed in what is called the clients' table, which will be described in more detail in next section.

5.4 The Clients' Table

The clients' table shows which clients are connected to the server. It also contains information on the state of the clients, i.e., which screens are currently being displayed to the clients. You may open the clients' table with the first command from the Run Menu.

Each line corresponds to a subject. This subject is seated at a PC on which the client is running (column "Clients"). As long as no treatment is started, the clients can be moved to other lines. They can be sorted and shuffled with the commands "Sort Clients" and "Shuffle Clients" respectively. As soon as you start a treatment, the order of the clients is fixed.

The clients' table has five columns:

Client: Name of the client. In general this is the name of the subject PC. If a client is no longer connected, its name appears in brackets. If a client with the same name should reconnect, it can continue at the same place where the previous subject left off. The caption to the clients' table gives the number of clients currently connected.

State: The state in which the subject is at the time. Example: "Ready" when the subject is waiting for the next treatment or for the next questionnaire.

Time: Displays the remaining time as it is shown to the subjects.

[3] The name under which one logs in depends on the installation. Typically one will set up a user account for all persons who carry out experiments.

5.5 How a Client Establishes a Connection with the Server

A session begins when the experimenter starts the server program z-Tree on the experimenter PC. After this, the client program z-Leaf is started up on the subject PCs. Z-Leaf establishes contact with z-Tree on the experimenter PC.

How does the client know the server's address?

Z-Leaf can determine the server's TCP/IP address with command line option `/server` *`ipaddress`*. If, for instance, z-Leaf is started with the command

Zleaf /server 100.20.10.233

then z-Tree must run on a PC with the TCP/IP address 100.20.10.233. This method is very useful if you want to run an experiment without a file server. If z-Tree is located on a file server, however, there is a more comfortable way: Z-Tree automatically writes it's TCP/IP address into the current directory in the file “server.eec”. If z-Leaf is started from the same directory (on the same PC or on another) it reads this file. If z-Leaf does not find this file, it looks for it in the directory “c:\expecon\conf” on its computer. If it is unable to find either, it can take the TCP/IP address of the local machine. It can now seek the server at this address. If the server is not found here either, the person who starts the client is asked if he or she would like to try again.

5.6 How to fix the name of a client

If z-Leaf is started up by means of a name on the command line (e.g., z-Leaf /name T1), this will be the name of the client (e.g., T1). Otherwise the name will first be sought in the file “name.eec” in the local directory. In this way you can run several clients on one PC. If the file “name.eec” does not exist, the host name of the PC is used. The host name is entered in the network control panel under TCP/IP.

5.7 During a Session

In the windows “globals table”, “subjects table”, “contracts table” and “summary table” the content of the appropriate table is displayed. In these tables the sizes of lines and columns can be changed. These tables are automatically exported to the xls file.

5.8 Concluding a Session

Each session ends with a questionnaire. Generally, the address is asked first so that the payment file can be written. Next, additional questionnaires are inserted and finally a good-bye screen appears which also displays the profit. How questionnaires are made is explained in section 4.

5.9 Dealing with a crashed or blocked subject PC

All data is stored in various files in readable form as soon as it is complete. Besides this, all messages exchanged between server and clients are stored in a (non-text) file, the GameSafe. With the help of these files it is generally possible to continue the experiment after a computer crashes.

In the case of a problem on a subject PC, you proceed as follows: If the PC is still working, it can simply be restarted. It then automatically receives at accelerated speed all messages it had received before. After this, it is in exactly the same state it would have been in, had the crash not occurred. Of course, this also means that it can be further than before: If, in the meantime, the time set for a stage has passed, then the treatment moves ahead by a stage, even if not all the clients are connected.

There are several possible reasons why a subject PC may not resume a treatment. It is possible that the server is very busy. In this case you simply have to wait a little. However, if the server is in a wrong state due to a programming error, the server has to be restarted.

If the subject PC is not functioning, you can start a new PC. Discard the old PC from the clients' table with the command "Discard Client" from the Run Menu. As soon as the new client appears, it can be manually assigned to the subject who was working on the old computer. To do so, you select the client field of the new client in the clients' table and move it over the field of the old client.

5.10 What to do if the experimenter PC crashes

After a crash of z-Tree, you have to do the following:

- Restart z-Tree.
- Open the clients' table.
- Restart all clients with the Menu "Run/Restart All Clients". If some clients do not connect, go to the client and reconnect the client manually. To reconnect a client manually, you quit z-Leaf with <alt>-[F4] and start z-Leaf again.
  If no client connects, then you have four possibilities:

  - You can restart all clients manually.
  - You can wait a while and try to restart the clients later. (It can take up to 4 minutes.)
  - You can shut down and restart the experimenter PC. (Exiting windows and logging on again is not sufficient.)
  - You can start z-Tree on another computer.

- With the Menu "Run/Restore Client Order" you sort the clients in the same order as they had been in the crashed session.
- With the Menu "Run/Reload database" you restore all tables. Since the tables are stored after each period you can restore the state when the last period was finished.
- Check how may periods have been played. You find this information for instance in the summary table or in the subjects table.
- Open the treatment you were running before. If n periods have been played, set the number of practice periods to –n (minus n).
- With the Menu "Run/Start treatment" you start the treatment again.

5.11 What Happens if Subjects Suffer Losses?

Losses by subjects can be covered by the following sources:

1. Previous profits.
2. Lump-sum payment.
3. Show-up fee.
4. Money injected during the session.

When a subject suffers losses, messages appear on the screen. The text of these messages is determined in the treatment (in the background).

If the first two sources cannot cover the losses, but the show-up fee can, a message appears on the subject’s screen. It informs the subject that he or she can now choose either to use the show-up fee or drop out of the game. If the subject uses the show-up fee, he or she may simply play on.

If, on the other hand, the subject chooses not to use the show-up fee to cover his/her losses, he or she reaches the state “BancruptShowupNo” and you have to release the subject from the server. If the show-up fee is used and exhausted, a message appears on the subject’s screen informing him/her that he or she has incurred a loss. This message can be concluded with a question. If the subject answers this question with “yes”, he or she arrives at the state BancruptMoreYes. By answering “no”, he or she arrives at the state BancruptMoreNo. In this case also, the experimenter has to release the subject from the server. A dialog appears by double-clicking the “State” field. There are three possibilities:

1. You allow the subject to continue. In this case you need to type a number into the field “Amount injected” that is higher than the current loss. You either make the subject pay this amount or, alternatively, consider it to be the credit limit. You take it into account when ascertaining whether subject’s earnings are negative. However, it is not shown together with the subject’s profit, nor is it considered in the payment file. This means that you need to add the amounts injected to the amount indicated in the payment file. In this way, the payment file lists the net amounts paid out to the subjects.

2. Another subject takes on the role of the subject released. In this case, all profits are reset to zero and the new subject is able to continue the experiment at the PC of the released subject.

3. The subject drops out. You need to be careful about this as it may change group sizes and the information on the active subjects may therefore no longer be correct.

5.12 The Files Generated by the Server

The following files are created in the course of the experiment. They are distinguished by their suffixes.

pay Payment file. Name of subject and profit.

adr Addresses of the subjects.

sbj Questionnaire responses. Without subjects’ names.

xls Contains all tables shown in the course of the session. As the tables are stored chronologically, you need some preparation to work with them. The first column contains the treatment number, the second the name of the table and the third column contains the period variable. It is a good idea to sort the whole file by these three columns. Then the tables can be copied into separate tables.

gsf GameSafe in binary form. It can be exported into a readable file but this file will be gigantic.

The starting time of the session is used as file name. The format is YYMMDDHM. YY means year, MM month, DD day, H hour and M minutes. In this way the start of the session can easily be reproduced. The hours correspond to the following table:

Code	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Hour	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23

The minutes correspond to the following table

Code	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
Hour	00	02	04	06	08	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58

5.13 Data Analysis

The xls file can easily be used for data analysis. However, there are tools in the tools menu that facilitate this process considerably. These commands allow you to separate the different tables into separate files and merge similar files into a big file for analysis. The best way to do this is to copy all the data files into a new directory, then apply **Separate Tables...** to all files. (You can select all files in one dialog by holding the ctrl key when clicking.) Then you apply the **Merge Files...** command. A dialog appears and in this dialog, you add all files which have a similar structure – for instance all subjects tables. It is not necessary that all merged tables have the same variables. The merge tool creates a new table that contains all variables that appear in any of the tables.

6 Installation

6.1 A Simple Installation with a File Server

The easiest way to install z-Tree is to put z-Tree and z-Leaf in a common directory on a file server. The server must have both read and write access to the directory while the client should only have read access. When z-Tree starts, it writes the IP-address of the server PC into the file server.eec in this directory. The z-Leaves read the address of the server in this file and connect with z-Tree on this machine.

6.2 Installation without File Server

Z-Tree and z-Leaf can run on different computers that do not have access to a common file server. This option is needed if you want to run experiments over the Internet. As opposed to an installation with a file server there is no easy method for z-Leaf to know where z-Tree is running. Therefore, z-Leaf must have the file server.eec (the file contains the IP address of the server) in it's directory or in the directory c:\expecon\conf. Z-Tree's IP address can also be set with the option /server when the client starts up.

If z-Tree and z-Leaf do not run from a common directory, the function "Restart all clients" does not work. In case there is a problem, the clients have to be started up manually.

6.3 Setting up a Test Environment

You do not need several computers to test programs. You can start z-Tree and more than one z-Leaf on one computer. However, you have to give the z-Leaves different names. For this you use the command line option `/name`. For example, you can start z-Leaf twice, once with "`z-Leaf /name A`" and once with "`z-Leaf /name B`".

6.4 Running an Experiment in Different Lanuages

All commands in z-Tree are in English. Most texts for the subjects are defined by the experimenter and can therefore appear in any language. The error messages in z-Leaf, however cannot be overridden. The default language in z-Tree is German (the language witten in Zurich). You can change the language in z-Tree in the menu or with command line options. To change the texts in z-Leaf, you have to use the command line option. It has the form /language *lan.* lan can have different values such as "en" English or "de" for German (see all available languages in the reference manual). If you want to change the language in your environment, you can create a shortcut for z-Tree that initializes the language to your choice.

6.5 Running more than one z-Tree

If you want to run more than one z-Tree on one computer, the different instances must use some identification to be distinguished from each other. This distinction is called the channel. The channel is set with the command line option /channel *ch*. It determines the channel through which the z-Leaves establish contact to z-Tree. The actual channel is 700+ch. The z-Leafs and the z-Tree that belong to one session must use the same channel.

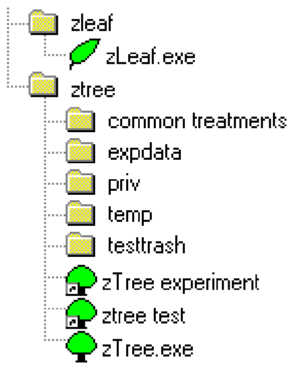
Tip: If you run several, but not too many, z-Trees simultaneously, wait at least 2 minutes until you start the next z-Tree, so that each instance has another session ID. This will make data analysis easier.

6.6 A Sample Installation

In this section, I describe how z-Tree is installed in the lab at Zurich. We use the command line options used to tell z-Tree where the files should be stored. These options are:

/xlsdir	Sets the directory where the xls file is stored.
/sbjdir	Sets the directory where the subjects file is stored.
/datadir	Sets the directory where the xls and the subjectsfile are stored.
/adrdir	Sets the directory where the address file is stored.
/paydir	Sets the directory where the payment file is stored.
/privdir	Sets the directory where the address and the payment file are stored.
/gsfdir	Sets the directory where the GameSafe is stored.
/tempdir:	Sets the directory where the temporary files @lastclt.txt, @db.txt and @prevdb.txt are stored.
/datadir	Sets the directory where the treatment and questionnaire files are automatically stored (before they run).
/leafdir	Sets the directory where the file server.eec is stored.
/dir	Specifies a directory where z-Tree put all files.

The installation in Zurich has the following directory structure:



"ztree experiments" is the shortcut:

```
z-Tree /datadir expdata /leafdir ..\zleaf /privdir priv /tempdir temp
/gsfdir temp
```

and "ztree test" is the shortcut:

```
z-Tree /dir testtrash /leafdir ..\zleaf
```

Experimenters have read and write permission in all directories. Subjects have read permission in the zleaf directory.

We have placed z-Leaf in a different directory from the directory where z-Tree is located. The subjects only have access to this directory and they have only read access to it. This prevents subjects from getting access to your experimental data. To be able to automatically start the z-Leaves wherever z-Tree is started, we set the leaf directory in the options of z-Tree.