Design and development of an object-oriented test generator for black box system testing

Besjes, R.H.T.J.

Award date: 1995
Master's Thesis:

Design and development of an object-oriented test generator for black box system testing

R.H.T.J. Besjes

Coach at Siemens : W. Roemer
Coach at the TUE : Dr. Ir. A.C. Verschueren
Supervisor : Prof. Ir. M.P.J. Stevens
Period : August 1994 - April 1995

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses.
R.H.T.J. Besjes

Siemens AG,
Berlin, 04/06/95

OOTG:
An Object-Oriented Test Generator for Black-Box Functionality Tests.
Preface

As a conclusion to my studies, electrical engineering at the Technical University of Eindhoven, I did my graduation project at Siemens Berlin. This report contains a description of the results of seven months working in a good atmosphere at the department PN STI 16.

The subject of my work was the design and development of an Object-Oriented Test Generator that can be used for black box functionality testing. This subject is based on an idea of Mr W. Römer who wondered why there were no known techniques that use the advantages of Object-Oriented programming for black box functionality testing.

The goal of my job was to examine the way in which Object-Oriented techniques can be used for system testing over the user interface. A method had to be created, that can be the base for a test generator that has to be developed by my successors in the OOTG project. I wish my eventual successors good luck.

I want to thank all people that supported me in doing this project. Special acknowledgements go to:

W. Römer for the basic idea of OOTG and for his enthusiasm in leading the project.
M.P.J. Stevens for being my graduation professor and for his stimulating criticism.
A.C. Verschueren for adding his ideas and for shaping and correcting mine.
R. van Gaalen for the helping hand she offered in arranging all secondary matters.
Summary

The goal of the OOTG project is to design and develop a test generator that uses the Object-Oriented capabilities of C++ for black box system testing. The final system has to allow a flexible automatically generation of functionality tests.

This master thesis is the first report on the subject of OOTG. It reflects the state of the project after seven months. During these, a method was developed that can be the base for the rest of the project.

Black box system testing is used to test the readiness of a system for sending it to the customers. The whole system with exception of its user interface is seen as a black box. The goal of this testing is to examine if the system behaviour matches with the system specifications.

The basic idea behind OOTG is that the specifications of a system can be defined in an object-oriented prototype of the user interface. In this prototype all possible stimuli (user actions) as well as all expected reactions of the system have to be defined. The correct functionality of the system can be examined by an automatic comparison of the prototype and the real interface.

The reason that Object-Oriented methods are used, is that they have proven to have good qualities for prototyping. The structure of most user interfaces is very standard. An effective reuse of code can be achieved by typical Object-Oriented techniques as inheritance, aggregation and class library utilisation.

To check the concept of OOTG and to experience the difficulties of defining a prototype, the methods of OOTG were tested in practice. A dummy test driver was developed and a prototype interface for a simple Windows application was defined. The results of the tests were positive. It is too early to conclude that a commercial version of OOTG is reachable, but further research is recommended.
Contents

Preface .................................................. I

Summary ................................................ II

Contents .............................................. III

1 Introduction ......................................... 1

2 The life cycle of systems and the role of testing ................................................... 2
   2.1 The Waterfall Model .............................. 2
   2.2 The different types of testing in the different phases .......................................... 4

3 System Testing ......................................... 5
   3.1 Functionality testing over the user's interface .................................................. 5
   3.2 The Object-Oriented approach to testing ....................................................... 5
   3.3 The depth of testing ................................ 6
   3.4 The Object-Oriented approach to system testing ............................................... 7

4 Object-Oriented Programming (OOP) ................................................................. 8
   4.1 The Terminology of OOP .......................... 8
   4.2 An Example ......................................... 9

5 OOTG ...................................................... 11
   5.1 Basic ideas of OOTG ................................ 11
   5.2 Defining a user interface .......................... 11
   5.3 The Object-Oriented description of an interface ............................................. 12
       5.3.1 The Objects in the example 'Clock' ................................................. 14
   5.4 The principles of an OOTG test run ............................................................. 15
       5.4.1 The interactive mode ................................................................. 16
   5.5 A closer look at Test Stimuli and Objects ...................................................... 16
   5.6 Error Coverage ....................................... 19
   5.7 The Decision Making Device ....................... 20
       5.7.1 Graph representation of the Object-Oriented prototype ............... 21
       5.7.2 Randomising a systematic test strategy ........................................... 22
   5.8 Definition of the basic OOTG Classes in C++ ............................................... 23
       5.8.1 Graph representation of the Object-Oriented prototype ............... 23
       5.8.2 The structure of an Object-Oriented prototype .................................. 23
1 Introduction

The Object-Oriented approach to software development has gained a lot of attention in the last few years. The main reason is that it can offer a considerable saving of time, by an effective reuse of code. Even though the use of Object-Oriented Programming (OOP) has grown in many directions, there is currently no known technique that uses the advantages of Object-Oriented methods for generating functional tests for systems. In OOTG the Object-Oriented methods and black-box functionality testing are brought together. This results in a test tool that allows a flexible automatic generation of tests, by making use of the Object-Oriented capabilities of C++.

OOTG is designed for black box, specification based, functionality tests. These types of tests do the job of the former manual system tests. In the process of software development these tests are used to examine the readiness of a product for sending it to the customer. The only part of the system that is supposed to be visible is the user interface.

The basic idea behind the tool is that a user interface -independent of the underlying program technique- can be specified in an Object-Oriented way. The resulting specification is a kind of prototype-interface, that can be used for testing the functionality of the system. The prototype of the interface in combination with an automatic test executor will offer the possibility of testing a system fully automatically. An interactive mode in which the user controls the direction of the tests will also be supported.

The Object-Oriented approach of black box testing is not effective for all types of systems. The best results are achieved for the classes of systems for which the state of the machine can be determined by the state of its user interface. For systems with a lot of internal states, for example data banks, the definition of an Object-Oriented prototype of the user interface is very hard.
2 The life cycle of systems and the role of testing

The development of a modern complex soft- and hardware system can be seen as an evolutionary process. The life cycle of a system is usually characterised by an in time growing quality and functionality. Urged by competition and the wishes of customers, manufacturers keep on bringing out new versions with extended functionality and less errors.

The more complex the system, the more open it is for internal conflicts. A small change in one part of a system may cause problems in a totally other part. This is the reason that every change in the soft- or hardware has not only to be followed by a test of the part in which the change was made, but also by a test of the whole system. A model in which this is made clear is the waterfall model.

2.1 The Waterfall Model

The waterfall model is a model that was originally developed to describe the life cycle of software [BOE81]. Since the implementations of soft- and hardware components in a

![Figure 2.1: The waterfall model of the software life cycle.](image-url)
The model is illustrated.

Every rectangle in the picture stands for a phase in the life cycle of the system. As can be seen, each phase is concluded by a verification and validation activity whose object is to eliminate as many problems as possible in that phase. If it is necessary to go back to an earlier phase, iterations take place as much as possible in adjoining phases.

**Example:** If in the Operation and Maintenance phase a small error is found in a software component, a change usually has to be made in the code of it, so it is necessary to go back to the code phase. As can be seen in the model, the change in the code is followed by a Unit Test. The next step is the reintegration of the unit, concluded by the verification of the software product. Then the improved product is implemented in the system again, followed by a System Test. If the results of all tests are satisfying, the improved system can go back to operation.

It is obvious that in the example the costs of repairing one software error are relatively high. To fix the error we had to go back to the Code phase and pass all the succeeding phases again. The fixing would have been a lot of cheaper if the error was detected earlier.

![Image](image.png)

**Figure 2.2:** Relative cost to fix a problem as a function of the phase in which it was detected. The line marks the average that was found on some larger projects in the late seventies.
In practice the growth of cost in time turns out to be exponential. As an example the relative cost of correcting a software failure as a function of the phase in which it was detected is shown in figure 2.2. In this picture the drawn line represents the average values that were found in some larger projects that were examined in the late seventies [BOE81].

As is shown in the picture the costs can easily rise up to 100 times as much as that they would have been, if the fault was detected earlier. A good example, that shows it can even be worse, can be seen in the Pentium disaster at Intel in 1994 [HÜS95]. In this case a relatively small failure at lower level was found after 6 million processors were sold. Intel had to exchange the bad processors for good ones. Total costs: $300,000,000.-.

2.2 The different types of testing in the different phases

In paragraph 2.1 eight different phases were distinguished in the life cycle of the software of a system. Every phase was terminated by a validation or verification action. There is quite a difference between the first four phases and the others. Until the Code phase is started, there is not really anything that can be tested. In the first phases the specifications are developed that are the base for the testing in the later phases.

If we do not see the operation phase as a test, we have three different types of testing:

1) Unit Testing.
2) Integration Testing.
3) System Testing.

The Unit Testing in the Code phase and the Integration Testing (product verification) in the Integration phase are usually done by the development departments themselves. The people that find the errors are the same as the ones that solve them. The System Testing is done by a separate test department however.

After the implementation of a software (or hardware) product, the system and its specifications are given over to a separate test department. The task of this department is to check whether a system functions as described in the specifications or not. The goal is to reach a certain level of quality. If errors are found, the conditions under which they occur are traced. Descriptions of the errors are given back to the developers who will try to solve the problems. Until the test department cannot find any more failures this process is iterated. After a system has passed all tests, it can be taken in operation by the clients. In most cases there are still errors found during operation. The repairing of these belongs to the maintenance of the product.

Since the Object Oriented Test Generator (OOTG) is a test tool that can only be used for system testing, we will concentrate on this issue in the next chapter.
3 System testing

In this chapter a short description of system testing in general is given. Some often used techniques are described and some terminology is made clear.

As stated before, system testing is done by separate test departments. By comparing actual system operation with its specifications the tester examines the functional correctness of a system. It is important that next to the actions that a user can make all the expected reactions of the system are also described. Good specifications form the base for good testing.

3.1 Black Box Testing

The most common approach to system testing is called Black Box Testing. The meaning of this term is, that a system is tested just by looking at its functionality. What the underlying soft- or hardware is, is outside our scope. The whole system with exception of its user interface is seen as a black box. This is made visible in figure 3.1. In this picture 'specs' stands for specifications and stimuli stand for the input of user actions.

Since no tools are drawn in the scheme of figure 3.1, it might be suggested that the testing is done manually. Although the manual testing of systems is still used, this is not always true. As described in the Waterfall Model (see 2.1) the whole system is tested after every change that is made. This means that it is necessary to do the same tests over and over again. Fortunately there are test tools available for the testing of most types of interfaces. By using these a lot of time can be saved.

3.2 Automatic Test Tools

The most used types of test tools make use of test scripts that are written by the tester. The investment of writing this script is paid
back by the fact that it can be used over and over again. The actual testing is now done
by an Automatic Test Executor (ATE) which uses test scripts as input and has the test
results as output. In figure 3.2 this configuration is made clear by a simplified scheme.

Another variant is
drawn in figure 3.3. In
this case not only is the
testing is done
dismissed but also
the writing of test
scripts is automated.
Scripts are written by a
test plan generator (TPG) instead of by the user[BAU79]. This configuration is only
possible, if the specs are written in a formal specification language that can be
understood by the TPG.

A test script consists of a list of stimuli and tests. During a test run the stimuli are sent to
the interface by the ATE. The stimuli simulate the actions of a virtual user of the system.
After every stimulus in the script tests can be defined, the ATE executes these as well.
The tests usually examine the state of (parts of) the interface. Results are written into a
so called log file.

For complex systems the number of test scripts can be very large. If this is the case, it is
practical if the test executor is able to run all scripts in series without human
intervention. Problems can be caused by the fact that the start conditions have to be the
same every time a script is executed. If something goes wrong in the execution of one
script, the state of the system is not defined anymore and the execution of all following
tests becomes meaningless. A solution that is used in the practice is that the system under
test is fitted out with a reset function, that can be used by the ATE. After every
execution of a script the ATE resets the system. In this way the start conditions for the
next script can be assured.

The percentage of the errors found of the total number of errors present is called the
Error Coverage of a test. In practical systems an error coverage of 100% is never
reached.

3.3 The depth of testing

The need for testing is clear, but to what depth we have to test in what phase is not so
easy to say. The cost to fix a problem rises as it is detected later in the development
process (see figure 2.2). The cost to find a problem increases as the number of
remaining problems gets lower. The number of problems remaining in a certain phase can
get so low that the cost of finding another error becomes higher than the extra cost of
fixing it in a later phase. When this situation is reached it seems to be more economical
to go to the next phase than to look further for more errors.
As we will see, this approach is too simple, especially for the system testing. In practice a lot of more economical and practical factors can play a role.

A very important marketing factor is the reputation of the company. If problems are left in a system, because it is cheaper to fix them during operation than to find them in the system tests, clients will see this as a lack of quality, even if all problems are solved during operation. As a result clients can become dissatisfied, which will damage the reputation of the company. The optimal ratio between quality and cheapness depends very much on the market position that has to be defended.

Another factor that can play a role in determining the test depth is the impact that an error can have. An error in a system that is essential in a production process, for example, can result in a temporary stop of production. The costs of this stop can get a lot higher than just the costs of fixing the problem. In special applications like in avionics even human lives can be at stake. Reliable software is of the highest importance here, since no company wants to be responsible for a plane crash that was caused by a system error.

As it might have become clear, the depth to which we have to test is very product dependable. In planning the whole development process of a product, the depth of testing is also defined in advance. Usually the number of test cases needed is estimated as well as the budget. The testing can be stopped if less than an agreed number of open errors is left.

Experience has shown that the budget needed for integration and system tests typically amounts to 25% of the total development costs. With the growing complexity of systems the complexity of testing seems to grow faster than linear. The expectation is that the relative budget for testing will grow in time. For highly reliable systems like in avionics the percentage is of course a lot of higher.

3.4 The Object-Oriented approach to system testing

In almost all cases the system testing is done with Black Box testing techniques, in which only the user interface is part of interest. Most user interfaces are built of standard structures that are used over and over again. A very common structure is for example a button. The strength of Object-Oriented Programming is exactly that, after a structure is defined once, it can easily be reused as often as necessary. The link between OOP and system testing on this base is very logical.

In the OOTG project a method is developed to describe specifications in an Object-Oriented way. This description can be used, in the same way as specs in a formal language (figure 3), for creating tests fully automatically.
4 Object-Oriented Programming (OOP)

For a good understanding of OOTG, some knowledge of Object-Oriented programming (OOP) is required. This chapter contains a short description, that will provide just enough knowledge for the reading of this report. It contains a set of definitions followed by a simple example.

4.1 The terminology of OOP

The following set of definitions forms the base of OOP:

- **Class**: A type that defines the structure and behaviour of a collection of objects. A class acts like a template for constructing objects. The definition of a class includes specifications of its attributes and methods.

- **Attribute**: A data value, or variable, associated with all the objects in a class. A class may have an arbitrary number of attributes.

- **Method**: A routine, or procedure, to carry out some behaviour for an object. A method can be activated by a message from another object. Each class has an implied method to create a new object of that class (constructor).

- **Object**: A dynamic instance of some class. Objects are created at run time. All objects in a given class exhibit the same structure and behaviour. The object encapsulates its internal attributes and provides a set of methods for manipulating these.

- **Message**: A message is a way of communication between objects. One object sends a message to another which specifies the desired method to execute and any desired arguments for that method. This is similar to a subroutine call in a conventional programming language.

- **Encapsulation**: Any object can be seen as a black box. The only way to access or manipulate an object's attribute data is via the methods associated with that object. The implementation of the methods is hidden from the rest of the system.

- **Inheritance**: A class may inherit structure and behaviour of another (higher) class. By defining methods or attributes in a higher class, all subclasses can make use of them. A definition in a lower class can override a definition in a higher class.

- **Aggregation**: Classes can have a 'part of' relationship. One class can be built of other classes, in a way like a table is built of four legs and a top.

- **Polymorphism**: The same operation behaves differently on different objects.
The objects, built up of attributes and methods, form the centre of OOP. Groups of objects with the same qualities are divided to classes. Two objects of the same class share the same attributes and methods. Their attributes can have different values, but have the same names and are of the same types. The classes themselves can be seen as objects as well. They can inherit the methods and attributes of higher classes. In this way a hierarchical tree of classes is formed.

4.2 An example

To make the definitions from paragraph 4.1 more clear, a simple example is given below. In this example the class 'Terminal' with its subclasses is treated.

In figure 4.1 a possible tree of classes is drawn. Be aware of the fact that this is not the complete tree, it is just a simple example to demonstrate an OOP structure.

Every object is an instance of a certain class. It possesses not only the qualities of this class, but also the methods and attributes of all higher classes in the same hierarchy. This means, that an instance of the class pay phone, as shown in figure 4.2, can use the method dial number even if this is not defined in pay phone itself.

When the message Dial Number (4653469) reaches, the object Pay Phone Lynarstreet, the search for this method starts in the class Pay Phone. Because of the fact that it cannot be found there, the search continues in the class Telephone, where it is found.

A definition in a lower class can overrule a definition in a higher class. This means that, if a Method is defined in two different classes in the same hierarchy, the definition from the lowest class is used.
As can be seen in the picture, every attribute, that is defined in the higher classes, has a value in the object:

1. Number: 030-4653469 (defined in Terminal)
2. On/Off Hook: Off Hook (defined in Telephone)
3. Tone: Dial Tone (defined in Telephone)
4. Actual Credit: 2.85 (defined in Pay Phone)

For more information on the topic of Object-Oriented Programming [BOO94] is recommended.
This chapter contains a description of OOTG. It can be seen as the core of this report. The functionality of the test generator and the methods that are supported are covered.

5.1 Basic ideas of OOTG

The functionality of the systems we are interested in is largely defined by the behaviour of its user interface. By examining this behaviour the whole system can be tested.

The basic idea of OOTG is that a user interface can be described as an Object-Oriented structure, independent of the underlying program technique. The tester rewrites the specifications of the interface in an object oriented way. The result can be seen as a kind of prototype of this interface. All possible user actions that are of any interest, as well as the expected reactions of the interface are defined. The testing with OOTG will exist of comparing the behaviours of the prototype and the real user interface.

The Object-Oriented description is independent of the underlying hardware or program technique. If a system itself is programmed in an Object-Oriented way, then it is still very possible, that the tester defines other objects in his object oriented description, as the programmer did.

5.2 Defining a User Interface

As stated in 5.1, a tester has to define an Object-Oriented description of the user interface. It is impossible to do this without a clear definition of what a user interface is.

The most common definition might be:

The user interface is that part of the system that can be used to exchange information between user and system.

For the testing this definition includes too much. A tester has to try to keep the interface as small as possible, so only the parts that are (instead of: can be) used are of interest. As we will see, this means that the interface of the same system is not the same for every test!

In figure 5.1 the functionality of a private exchange with x telephones has to be tested. Because of the similarity of all phones, the tester might define for a certain test, that the interface exists of just two phones (see figure 5.1A). Functionality tests like simple calling can be done over this interface. When the tester wants to test a more complex function like conference calling, for which a connection between three or more users at the same time is built, the interface will have to be redefined as is done in figure 5.1B.
Remark: Especially in cases as shown in figure 5.1 the Object-Oriented qualities of OOTG are practical. In this case: Adding a telephone means just adding an object of an existing class.

5.3 The Object-Oriented description of an interface

In describing a user interface in an Object-Oriented way, a tester has to start by splitting the interface into functional parts that can be seen as separate units. In the Object-Oriented prototype these units will be simulated by so called Test Objects.

Definition:

Test Object: A Test Object is an object in the Object-Oriented prototype that stands for a functional part of a user interface and that can be seen as a separate unit.

The Object-Oriented prototype has to have the same qualities as the real interface. This means that for every part of the real interface that can have different states, attributes have to be defined in the prototype that can describe these states. All the attributes together must unequivocally define the state of the total interface.
Next to the same qualities, the prototype also has to show the same functionality. This means that all the reactions from the real interface on actions of a user, have to be specified in the prototype. This is only possible if the user actions themselves are defined as well. Every Test Object has to contain a list in which all the user actions that are of any interest in that object are described.

The description of a user action in a test object is called a \textit{test stimulus}. In the form of a linked list of test stimuli all possible user input is defined in a test object.

\begin{definition}
\textbf{Test Stimulus:} A test stimulus represents an action or a combination of actions of a user, that affects the user interface. A test stimulus simulates users input.
\end{definition}

A test run consists of the execution of a series of test stimuli. These series represent the actions of one or more virtual users. The order in which the stimuli are executed cannot be totally random. In most interfaces only the stimuli of one object at a time are part of interest. By definition at any moment during a test run of OOTG only the stimuli of one test object can be used. This object is called the \textit{active object}. Throughout the execution of a stimulus the activity can be handed over to another object.

\begin{definition}
\textbf{Active Object:} At any moment only the stimuli of the active object can be called. During the execution of a stimulus, the activity can be handed over to another object.
\end{definition}

As described in chapter 4, the functionality of an object in OOP is defined by its methods. When a test stimulus is executed, usually multiple methods are called. It is not necessary that all these methods are defined in the active object. By sending messages, methods (not stimuli!) of inactive objects can also be used.

Three different kinds of test stimuli can be distinguished:

1) \textit{Internal stimulus:} Solely methods of the active object are called. No other objects are affected.

2) \textit{External stimulus:} Methods from the active object as well as from other objects are called. The same object stays active.

3) \textit{Transition stimulus:} Like an external stimulus, but this time another object becomes active.

As an example of the Object-Oriented description of a user interface, the program 'Clock' is treated in the next paragraph. The interface of clock is the well-known Windows system.
5.3.1 The Objects in the example 'Clock'

The program Clock is a Windows application. Its interface has a very simple structure. It exists of only four different windows. The most logical step is to define these as the objects in our description of the interface. In figure 5.2 all the windows (objects) are pictured. Next to them there is a listing of their attributes and test stimuli. To keep things easy, the system menu (the button in the upper-left corner) is ignored.

![Diagram of the Clock interface with labeled objects and test stimuli](image)

Figure 5.2: The objects in the interface of the program clock
It is obvious that the definition of attributes and test stimuli is very subjective and depends on the actual circumstances. In our example only the changeable flags of menu items and Windows were considered as attributes. It is very possible that a tester also wants to test things that are supposed to stay the same, for example one can test something like the co-ordinates or the size of a window. In that case more attributes have to be defined.

For the first two windows it is clear which test stimuli are important. In the third window the number of possible user’s actions is large and many actions are quite similar. The tester decided that the differences between the fonts are not interesting in the tests. He accepted only the actions, that are structurally different as test stimuli.

Remarks:

- A test stimulus can exist of several user's actions. Sometimes a certain combination of actions is important. By defining this in one stimulus, it ensures that it will be tested.

- The fifth stimulus in the third window tests what happens, when a text is entered that has no meaning. As Myers said: 'Test cases must be written for invalid and unexpected, as well as valid and expected input conditions.' [MYE79].

5.4 The principles of an OOTG test run

In the last paragraphs it was stated what components are important in defining an interface in an Object-Oriented way. Test Objects, Attributes, the different kinds of Test Stimuli and Active Objects were introduced. In this paragraph the use of all of these will be made clear.

The Object-Oriented description can be seen as a prototype of the user interface: It shows the same functionality and has the same qualities as the real interface has to show. A test run exists of the parallel execution of the prototype and the system under test. By sending the same stimuli to both interfaces and by comparing their states their functionality is examined. In figure 5.3 this test strategy is made clear.

During a test run stimuli are executed. The order in which this is done is determined by a so called Decision Making Device (DMD). As can be seen in picture 5.3, this DMD selects a stimulus and passes it on to the Automatic Test Executor at the beginning of every test cycle. The choice of the DMD is narrowed by the fact that at any time only the stimuli of the active object can be used. More information on DMD's can be found in paragraph 5.7.

The ATE performs the stimulus on the real system as well as on the prototype. To do this, emulated users' actions are sent to the real system and messages that call methods to the prototype (the steps and 4 in figure 5.3).

We still have to ask: What has to be tested? and when?
It is not easy to give the answers to these questions. If the limited processor speed was of no concern, it would be best if every attribute of every object was checked after every stimulus. The problems are, that in practice this takes too much time and that under an interface like Windows not all objects exist at any one time. Attributes that do not have anything to do with the executed stimulus do not really have to be tested. It is however to much of a job, to define for every stimulus what attributes are of importance. The compromise that is used in OOTG is, that after every stimulus execution, only the attributes of the active object are examined. In every test object a Self Test is defined. The ATE calls the self test of the active object in the prototype and executes it on the real interface (the steps 5-7 in figure 5.3). The results of the tests are written into a LogFile (step 5.3).

5.4.1 The Interactive Mode

Figure 5.3 describes a test cycle of a normal fully automatic OOTG test run. To make the testing more flexible an Interactive mode is also supported under OOTG. In this mode the tester takes over the functions of the Decision Making Device and the LogFile. Now in every test cycle the tester himself selects the stimulus to be executed, so he has full control over the direction of the test run. The stimulus is executed and the corresponding tests are performed as usual by the test executor. The results of all tests are directly returned to the tester instead of written in a LogFile. The interactive mode can be practical, especially for analysing a found error.

5.5 A closer look at Test Stimuli and Objects

As was seen in the picture of figure 5.3, the input of the prototype is not the same as the input of the real interface. The prototype expects messages that call methods, while the real system expects (simulations of) user actions. In every stimulus both kinds of input have to be defined.

Often it is impossible to send signals directly to a user's interface. In these cases an interface driver is necessary. This can be a kind of library that contains functions that can be sent to the interface directly. It is also possible that next to software extra hardware is needed as well.
1) The Test Stimuli of the Active Object are handed over to the Decision Making Device (DMD).
2) The DMD selects one Stimulus and passes it to the Automatic Test Executor (ATE).
3) By calling methods the stimulus is executed in the Object-Oriented prototype.
4) By the simulation of user actions the stimulus is performed in the actual user interface.
5) After the stimulus execution the self test of the active object is called by the ATE.
6) The prototype returns the self test in which tests are defined for all attributes of the active object.
7) By the test execution the state of the real interface is compared with the state of the prototype.
8) The Results are written into a LogFile.

Figure 5.3: Schematic picture of one cycle of an OOTG test run.
Test stimuli can sometimes only be used under certain conditions. The best place to state conditions is in the stimuli themselves. With these conditions and the two types of input the total number of components of a stimulus becomes three.

### A stimulus exists of three different kinds of components:

1. **List of Method Calls:** This is a listing of all methods that have to be called to perform the stimulus execution in the Object-Oriented prototype.
2. **Simulation of user actions:** This is sent to the actual user interface of the system. If it is not possible to communicate with this interface directly, an interface driver has to be used.
3. **Conditions:** If the stimulus cannot be used under any circumstances, conditions have to be defined. The definition of the conditions is done in the form of a test which returns TRUE if the stimulus can be called and FALSE if not.

Remark: The third component is optional.

Just like a stimulus a Test Object is built of multiple different components. The most important of these were already seen earlier in this report. **Attributes, a Self Test and Stimuli** are the components that form the test objects. There are structures in many interfaces that have all the same qualities as test objects, but that are too small or too connected with other structures, to be defined as test objects. Since the same structures are used often, it is practical to introduce an extra type of components: **Items.**

**Items** represent functional parts of Test Objects that can be reused more often. Every item belongs to a Test Object or another Item, which is called **parent.** The Item itself is called **child.** A child can add stimuli and attributes to its parent. All attributes and stimuli defined in the children can be seen as qualities of the parent itself. Just like the Test Objects Items have a self test function. If the self test of the Test Object is called, all self tests in the children are initiated as well.

The use of Items is that they allow an effective reuse of code. If for example a certain type of button is used often in an interface type, it only has to be defined once. From then on it can be used in every Test Object.

Items are built of the same components as Test Objects. They exist of Attributes, Methods, Stimuli, a Self Test and recursively an Item can even contain **Child Items.** The main difference between a Test Object and an Item is that a Test Object has an Active Flag. During a test run it can become active, an Item can not.
A Test Object or an Item is build of five different kinds of components.

1. **Attributes:** These define the qualities and the state of the test object or item. One attribute that is always present in every Test Object is the Active Flag. This flag is used to define what object is active. At any moment only the flag of one object can be set.

2. **Methods:** Methods are needed to manipulate the attributes. They can be called during the execution of a stimulus.

3. **List of Test Stimuli:** In any object or Item the stimuli that belong to it are defined in the form of a linked list.

4. **List of (Child) Items:** Not all attributes and stimuli have to be defined directly in the Test Object. Items can be used to add attributes and stimuli in an indirect way.

5. **Self Test definition:** Every Test Object or Item can test itself. By calling the self test the values of the attributes in the Object or Item are compared with their counterparts in the real interface. When a Self Test is called, automatically all Self Tests of Child Items are called as well.

---

### 5.6 Error Coverage

The quality of a test is for the main part determined by the error coverage that is reached by it. This coverage is the percentage of errors found on the total number of errors present. The five factors under OOTG that have the largest influence on the error coverage are:

1) **The depth of separate tests:** Under OOTG all attributes of the Active Object are examined in every test cycle. The tester also has the possibility to define extra tests.

2) **The percentage of attributes defined by the tester:** A tester defines what testable qualities of the user interface are important enough to be attributes. The more attributes defined, the more qualities can be tested.

3) **The percentage of stimuli defined by the tester:** A tester defines what user actions are important enough to be stimuli. The more different stimuli defined, the more different user actions can be tested.

4) **The order in which stimuli are executed:** The chance that an error occurs in a certain test cycle is in most cases not only dependent of what stimulus is called, but also of the actual state of the interface. This state is set by the order of the stimuli that were executed before. The part of OOTG that is responsible for the order of the stimulus execution is the Decision Making Device. An efficient routine for the DMD has to try to execute stimuli as much as possible in different interface states.

5) **The number of test cycles:** It is obvious that a higher error coverage can be reached with a larger number of test cycles. Usually time is the limiting factor for the number of steps in a test.
In a lot of cases the last two factors can be seen as one. By the fourth factor the efficiency of the test is defined. The ratio between the efficiency of the test and the number of test cycles is in fact the factor that affects the Error Coverage.

5.7 Decision Making Devices

For creating test plans automatically, a device is needed that makes a decision at any moment that there are multiple options. Under OOTG the DMD is used at the beginning of every test cycle to make the decision of what stimulus has to be executed. In this way it has full control over the order in which stimuli are called during a test. As stated before, the efficiency of the testing is largely determined by this order. The core of a DMD is usually a routine by which systematically or randomly decisions can be made.

A practical model on which a lot of routines for automatic test plan generators are based is the Finite State Machine. This is a graph representation of a system in which every node stands for a state of the system and every branch for a transition between these states. A test routine has to try to walk through the graph in such a way that all different states are reached and all transitions are used at least once, within a number of test steps that is as low as possible. In figure 5.4 a simple example of a state machine is shown. It describes the behaviour of a traffic light system [VRA94].

To use the state machine model, all states of a system have to be known. Under OOTG however at any moment only the current state of the interface is defined. This means that the finite state model in this case is not very practical. One could think of trying to determine all different states but this is very difficult for black box specifications and the number of states is often so large that it would get out of bounds. It seems more logical to look for another model.
With black box testing a 100% test of a complex system is impossible. Since it is not known what different system states do exist, it is never sure if all states have been reached. The goal of a test routine must be to reach as many possible different states related to the number of test steps.

### 5.7.1 Graphical Representation of the Object-Oriented Prototype

The optimal routine for the DMD of OOTG is still under study. An aid in finding a good method can be a graphical representation of the Object-Oriented prototype. In this graph the test objects serve as nodes and the stimuli as branches. For the interface of Clock this graph is drawn in figure 5.5.

![Graph](image)

**Figure 5.5**: The graph belonging to the Object-Oriented prototype of the user interface of the program ‘Clock’.

The job of the DMD is in fact to walk through the graph in such a way that as many different states as possible are reached within the number of test steps available (often a question of time). The problem is that it is not known what different states do exist. For a test generator that is based on the finite state machine the goal is to reach all different
states at least once. In this case we would like to have the same kind of goal, this is however impossible since it is never known how many states there are.

What we do know is which test objects exist and what stimuli there are. A minimal demand will be, that every test object is activated and every stimulus is called at least once. This would be easy to do if there were no conditional branches in the graph. Since stimuli can sometimes only be used under certain conditions, the related branches will also not be available at any time (in figure 5.5 stimulus 2.3 is conditional). In the graph there is no definition of how a state can be reached in which all conditions for a certain stimulus are satisfied. The finding of this state is sometimes a puzzle that is hard to solve by a systematic method. Next to systematic routines random routines can also be used.

Statistical laws tell us that if we use a random routine and we have an endless number of steps and an endless number of tries, all possible states will be reached in the end. In practice however the number of steps and tries is always bounded. If the number of test steps can be relative large (this is a question of the available time and the complexity of the system), a random method can achieve good results.

5.7.2 Randomising a systematic test strategy

Until now it might have been suggested that a test technique is either systematic or random. In this paragraph we will see that a combination of systematic and random is also possible. Often a systematic test technique can be strengthened by introducing a random factor!

A lot of systematic selection routines in automatic test generators are based on some basic rules. These rules are used for making any decision. The goal is often to call as many different stimuli as possible within a low number of test steps. Until all unconditional stimuli are called once, this strategy is probably the best. After that we have to concentrate on trying to call stimuli under as many different circumstances as possible. In doing this these basic rules often produce poor results. The problem is that every time a certain state is reached, the same decision is made. This can easily result in loops in which the same pattern is repeated time after time. If we try to prevent this with extra rules, the chance that another kind of more complex loops will occur is very large.

A solution for the problem stated above can be a random factor. In a test run the stimulus selection is not only done by the rules, but sometimes also by coincidence. A concept might be that in every test cycle there is a chance of for example 5% that the selection by the rules is overruled by a random selection.

If a pure random factor is used, the tests are not repeatable anymore. Since repeatability is very practical for analysing occurred errors, this is not desirable. The solution is found in a so called pseudo random factor. The meaning of this term is that a kind of list is used with random selections. Every time the script is run with the same list, the same random selections are made at the same places. In practice so called pseudo random functions can be used to generate these lists.
As said before, the optimal routine for the DMD of OOTG is still being studied.

5.8 Use of the basic OOTG Classes in C++

One of the most important advantages of Object-Oriented programming is, that, by the utilisation of classes and libraries, an effective reuse of code can be reached. So far we have concentrated on what we have to define in the Object-Oriented description of the interface. In this paragraph the main questions will be how? and where? The implementations of the four basic classes of OOTG, TestObject, Item, Stimulus and MethodCall, under C++ are treated. For a better understanding, some programming techniques are also described.

5.8.1 Linked lists of objects

A typical structure that is used very often under OOTG is the so called Linked List of Objects. A linked list is built in such a way that, if only a pointer to the first object is known, all the objects are reachable. To make this possible, every object in the list contains a pointer to the next. The last object contains a NULL pointer. A graphical representation of a linked list with n objects is drawn in figure 5.6. With very simple functions we can walk through the list.

Under C++ every instance of a class is created by a constructor function. There is always a standard constructor that can be used for just creating the instance. It is also the possible to define a constructor in which other functions are called. This quality can be combined with the linked list technique. We can define such a constructor, that an object is immediately placed in a linked list, at the moment it is created. If we make sure that the pointer to the first object is known, we can always reach all others.

How all this is used in OOTG is shown in the next paragraph.

5.8.2 The structure of an Object-Oriented prototype

Paragraph 5.5 described what components are needed in stimuli, testobjects and items, to give them their functionality. In this paragraph we will take a look at the global structure of an Object-Oriented prototype. The location of the different types of objects will be the main topic.

At the highest level of the prototype we have a linked list of TestObjects. The list starts with a global pointer to the first object. This pointer is the handle, that is used for reaching all. At the moment a TestObject is created, it is immediately put in the list by its constructor function.
In every testobject there is a pointer to a linked list of stimuli and a pointer to a linked list of items. These lists can be seen as sub lists of the list of testobjects. Via the list of testobjects also all lists of stimuli and items can be reached. Recursively sub lists can start in the items and stimuli as well. In a stimulus there is a list of MethodCalls. In an item there is another list of stimuli and a list of child items. In a child item there is a list of stimuli and another list of child items. Independent of the number of objects and the depth of the lists, all method calls, stimuli, items, child items and testobjects can always be reached, as long as the pointer to the first TestObject is known.

By using a hierarchical structure, under OOTG, the overview can be kept even as interfaces get more complex. In figure 5.7 a schematic overview is given of the structure of a virtual system. In every stimulus there is a list of method calls. For the clarity these lists are not drawn in the picture.

Figure 5.7: Scheme of the structure of an Object-Oriented prototype under OOTG.
6 The Dummy Test Driver

A dummy test driver was developed to prove the concept of OOTG. The goal was to see whether the basic methods of OOTG are achievable in the practice. It was not the intention to develop a total test generator in detail. Simplicity was the motto.

At first a system to test was needed. It had to be simple, it had to be present, it had to be controllable without a complex interface driver and last but not least it had to have a clear user interface. A system with all these qualities was found in the program ‘Clock’. The user interface of Clock is the well-known ‘MS Windows’.

As described in chapter 5, the following functional parts are needed for the use of OOTG:

- An Object-Oriented Prototype of the user interface
- An Interface Driver
- A Decision Making Device
- An Automatic Test Executor

In appendix 2 there is a full description of the Object-Oriented prototype of the user interface of the program Clock. The other parts of the Dummy Test Driver are treated in the following paragraphs.

6.1 The used ATE and Interface Driver

The use of an interface driver is to send simulated user actions to a system. In this case our system is a Windows application. Two usable techniques were found for controlling Windows applications:

**DDE:** Under Windows communication between different applications is possible by the use of so called Direct Data Exchange (DDE) techniques. If the Automatic Test Executor (ATE) is developed as a Windows application (which is quite easily to do with Visual C++), DDE can be used to send stimuli and tests directly from the ATE to the application under test. No further Interface Driver is needed in this configuration.

**WTT:** A totally different option makes use of the so called Windows Test Tool (WTT). WTT is a tool which is developed at Siemens for doing black box testing over a Windows interface. The input for every test run is a test script that contains a list of simulated user actions (stimuli) and tests. In this option the ATE does not test anything. All stimuli and tests that were normally sent to the user interface are now written into a WTT test script. This script is later used by WTT to do the actual testing. In this way there is no parallel execution of the prototype and the real interface, the basic methods of OOTG however can also be proven with two serial
runs. In this case the MS Test Driver [MIC93] which is used by WTT can be seen as Interface Driver.

Although the first option is certainly the most subtle and effective one, the second was chosen. The main reason was the already present knowledge of WTT, -to use DDE foregoing studies would have been needed. Since the Dummy Test Driver had to be simple, WTT was preferred. For later versions of OOTG however the DDE option is recommended.

The choice for a technique without a parallel execution of the prototype and the real interface means that the scheme of a test run as drawn in figure 5.3 is not valid for the dummy test driver. In the figures 6.1 and 6.2 the schemes for the two serial runs used by the dummy are depicted.

First the part of the Object-Oriented Prototype is run. Every test cycle in this execution consists of the seven steps that are listed below (see also figure 6.1):

1) The stimuli of the active object are handed over to the Decision Making Device (DMD).
2) The DMD selects one active stimulus and passes it to the Automatic Test Executor (ATE).
3) By calling methods the stimulus is executed in the Object-Oriented prototype.
4) The Simulated user actions are written into a WTT test script.
5) The self test of the active object is called by the ATE.
6) The prototype returns the self test in which tests are defined for all attributes of the active object.
7) The tests are passed on by the ATE and written into the test script.

![Diagram of test cycle](image)

**Figure 6.1:** Scheme of one test cycle in a run of the OOTG part of the Dummy Test Driver.

The test script that is generated during the run of the prototype, is used as input for the
The execution of the WTT part. The test cycles in this execution consist of the following steps (see also figure 6.2):

1) Simulated user actions are read out of the test script.
2) The simulation of user actions is performed in the user interface of the system.
3) The Tests are read out of the Test Script.
4) The state of the interface is examined by the execution of the tests.
5) The results are written into a LogFile.

Although the use of two serial runs seems to be quite different from one parallel run as described in chapter 5, there is a great similarity. If we see WTT as the interface driver, the only difference between the configurations is the use of a test script in the serial one. The script is necessary to bridge the time between the first and the second run.

6.2 The Decision Making Device

The optimal DMD for OOTG is still part of study. To keep the dummy test driver simple, we looked for a DMD that could be implemented quite easily. The most simple DMD is a random generator that totally randomly selects one of the active stimuli in every test cycle. The problem with this is that a random test is not repeatable. This makes the analyses of errors difficult. An option that is still quite simple, but has better results, was found in the so called pseudo random generator (PRG).

A PRG can be used to generate a list of ‘random’ numbers. These numbers are not totally random, since they are created with the help of a stable routine. This means that every time the generator is started, the same ‘random’ list will be the result. To get the
possibility of creating multiple different lists, most PRG's can be initiated with a certain number, which is called the seed. If a generator is started twice with different seeds, the outputs will be two different lists. If the same seeds are used, the lists will also be the same.

Under OOTG no functions that have coincidental factors are supported. This means that if a pseudo random generator is used, tests are repeatable. There is the possibility of executing different tests by starting the generator with different seeds.

**Remark:** Under visual C++ a Pseudo Random Generator is predefined in the standard library (stdlib).
7 Parallel Processes

OOTG is designed for black box testing over the user interface. For this type of testing we are not interested in the internal processes of systems. An exception was made for the so called Parallel Processes. Because of the fact that quite a lot of system errors are caused by these, we gave them some extra attention. In this chapter a short description is given of the parallel processes in general and of the different kinds of errors that can be caused by them. At the end we have some propositions that can be used in the further development of OOTG.

7.1 General

In control theory it is often assumed that a system's inputs and outputs can be ordered a priori in time. In practice many systems are not sequential in this sense [AND92]. Especially for distributed systems parallel processes can play a major role.

All distributed systems share some resources, including data. Where a resource cannot be simultaneously used by all processes, some of these processes may be forced to wait for others that are using the resource. This can lead to problems like dead-locks, live-locks and stochastic changes in the output order. In the next paragraphs the different kinds of problems will be explained by the use of examples.

7.2 Dead-Locks and Live-Locks

As multiple parallel processes in a system share the same resources, a state might be reached in which two or more processes cannot continue because they are blocking each others resources. The resources cannot be released, if the processes do not continue and the processes cannot continue, if the resources are not released. Depending on the degree of freedom that is left for the system, a state like this is called a dead- or a live-lock.

We speak of a dead-lock as a state is reached in which the process starts an endless wait for the resource. No actions are performed anymore.

In the state of a live-lock the process is still active in the sense that actions are performed. The problem this time, is an endless loop of actions that can only be left by the use of the needed resources.

An example of an unreliable construction in which a dead-lock can easily appear, is drawn in figure 7.1. Because of the similarity of dead- and live-locks only one example is used to explain both of them.

In the construction in this picture two two-way buffers are linked together [MIL89]. They can transmit any number of messages in either direction repeatedly. If a message
enters the system at the left, it will be handed over from the left to the right buffer and then leave the system at the right. For messages from the right we have the same story with exchanged directions.

Figure 7.1: Example of a construction in which a deadlock can appear.

As long as the messages enter from one direction at a time no problems will appear. The difficulties arise as two messages arrive at the same time from different directions. Both buffers will be filled now. The left buffer will try to hand its message over to the right one. This is impossible since the right buffer is already occupied by the other message. On the same ground the right buffer cannot deliver its message either. There is no way of continuing. The state that is reached is called a dead-lock.

7.3 Stochastic changes in the output order

Stochastic changes in the output order can appear as in a data transfer process multiple channels are available for the serial transmission of data. In picture 7.2 a configuration is drawn of three exchanges. In this picture only the channels that are used in our example are drawn.

Figure 7.2: Direct and indirect data transfer from exchange A to B.

The communication principle that is used by the exchanges in the picture is the ISDN packet transfer [STA92]. Small packets of data have to be sent from exchange A to B.
The direct route is preferred, but if the connection between exchange A and B becomes blocked or occupied, an alternative route via exchange C is available.

We look at the situation that the direct line is occupied and data packets are sent to B via C. At a certain moment the direct line is released and the packets can take the normal route again. In figure 7.3 the transfer timing is shown for the packets that were sent just before or just after the moment of releasing.

As can be seen in the picture, the two packets that were sent just after the releasing arrive before the packet that was sent just before it. The order of the output was clearly changed by the influence of a parallel process that occupied the direct connection between A and B.

**7.4 The impact on the system level**

As stated before, for black box system testing we are not really interested in the internal processes of a system. What we want to know is what impact the internal problems, that are caused by parallel processes, have on the external behaviour of the system. The key questions are:

1. Do the problems described in the proceeding paragraphs always result in errors on the system level?
2. Do we have to adapt OOTG to the system behaviour caused by parallel processes?

We will answer these questions for the problems described in the previous paragraphs.

### 7.4.1 The impact of dead- and live-locks

The state of a dead- or a live-lock inside of the system often results in a lock of the whole system. This situation can easily be detected by the use of a timer. This timer reports a time-out if an expected system reaction is not noticed within a certain time.

A lock inside a system does not always result in a total lock. In this case the behaviour of the user interface will deviate from normal behaviour. Like for all errors this will be detected by OOTG when a state of the user interface is reached that differs from the expected state.

We can conclude that for the detection of system errors, caused by dead- or live-locks, the normal test strategy has to be complemented with a timer. The timer can also be useful to detect run time errors that were not caused by parallel processes.

### 7.4.2 The impact of stochastic changes in the output order

On the contrary to locks, stochastic changes in the output order of internal processes do not always result in errors on the system level. In fact there are three different cases:

1. The internal change of order does not affect the external behaviour of the system.
2. The internal change of order results in a bad system behaviour.
3. The internal change of order results in a different but correct behaviour of the system.

For the first two cases the normal test strategy of OOTG suffices. In the first case the test will just continue. In the second case the usual error handling procedure will be started.

In the third case the test strategy of OOTG cannot be used. The problem that we have is that we cannot prototype a random factor. Under OOTG we compare the behaviour of the real system with the behaviour of a kind of prototype. In the prototype all system reactions on stimuli are defined. If these reactions are not always the same, our technique will fail.

It is possible to define multiple different reactions on one stimulus in the prototype. This still does not offer a solution to our problem since we can not predict which behaviour the system will show during a test.

The solution might be found in the use of a kind of feedback. In the OOTG test technique the system and the prototype are parallel executed. The same stimuli are sent to both systems. After every stimulus execution the states of the systems are compared. The possibility has to be examined if we can change the strategy in such a way that the
stimuli are no longer sent to both systems at the same time. It might be possible to check what state is reached after the execution of the stimulus in the real interface. After this check the correct variant of the stimulus can be sent to the prototype.

A disadvantage of the proposed technique is that the run of the prototype will no longer be independent of the run of the real system. Some research on this subject will be needed to solve the whole problem.

7.5 Propositions

Following out of the examination of parallel processes two changes in OOTG are proposed:
- At first the test generator has to be fitted with a timer. In the case of a serial test run the detection of time-outs are very important since an endless wait for the continuing of a system does not only block the current run but also all the next.
- At second there has to be examined if the test strategy can be changed in such a way that also systems with random factors can be tested.
8 Conclusions and Recommendations

As a first conclusion we can say that the goals of the graduation work were reached. A method was developed in which Object-Oriented techniques are used for black box system testing. A theoretical base was created for the further development of the Object-Oriented Test Generator. Next to this, a dummy test driver was developed which was used to prove the achievability of the methods of OOTG in practice. It is too early in the project to conclude that a commercial version of OOTG is achievable, but, -based on the positive results of the tests, further research is recommended.

OOTG's approach of testing is not effective for all types of systems. For systems with a lot of internal states the definition of a prototype of the interface is very difficult. In such cases all black box testing methods will fail.

For the definition of a prototype of a user interface the Object-Oriented methods turned out to be powerful. Especially the use of libraries with standard classes was effective. A lot of interface parts can be standardised. The most time consuming job in defining a prototype is the definition of Method Calls. As proposed in appendix II this might be improvable by the use of a matrix. Since the definition of a prototype is an investment that has to be done for every system we want to test, research on this issue is strongly recommended.

In the first version of the test generator a pseudo random test routine was used. This routine needs a relative large number of steps for testing a system. The efficiency can be improved by the use of a systematic test routine. A lot of systematic routines are based on walking through a graph of the system. Because of the fact that the stimuli -defined in the Object-Oriented prototype- can be conditional, the definition of a graph is very difficult in this case. The type of graph that is used for the Markov model in appendix II might offer a solution, but we are afraid that this type of graph explodes as the systems become more complex. The size of the graph grows exponential with the number of conditions.

In programming the dummy test driver some concessions were made to keep it simple. One of the first things that has to be changed is the way in which stimuli are sent to the system. At this moment a test consists of two serial runs. In the first run the prototype is executed and a WTT test script is created. In the second run the WTT test script is sent to the actual system by the MS-Test driver. By the use of Direct Data Exchange (DDE) techniques, a parallel execution of the prototype and the system under test is possible. Next to the advantage that we do not need test scripts anymore, this gives us also the possibility of solving the problems that can be caused by parallel processes. At least a timer for the detection of run time errors has to be implemented. Next to this, the possibility of using a test strategy that allows random factors has to be examined.

Until now OOTG was developed for testing over the user interface. It might be possible to use it for testing over other interfaces as well. An interface between two systems also can be prototyped. The stimuli that we have to define now simulate actions or reactions of another system instead of of a user, but for the rest it is all the same.
References

[MYE79] Myers, G.J.
The Art Of Software Testing
New York: Wiley, 1979

Test Plan Generation Using Formal Grammars
In: Proceedings 4th International Conference on Software Engineering,
Munich, Germany, 17-19 September 1979
Munich Germany, IEEE, 1979, P. 425-432

[HET88] Hetzel, B.
Welesley, Massachusetts: QED Information Sciences, 1988

[HEL91] Held, G.
Objektorientierte Systementwicklung
Berlin; München: Siemens Nixdorf Informationssysteme, 1991

[AND92] Andersland, M.S. and D. Teneketzis
Information partitions, deadlock, and non-sequential stochastic control
In: Proceedings of the 31st IEEE Conference on Decision and control,
Tucson, AZ, USA, 16-18 Dec 1992
New York, NY, USA, IEEE, 1992, P. 1850-5 vol. 2

[GOR92] Gordon, I. and B. Hwong; M. Klinger; W. Sherman
Object-Oriented Analysis And Design Methods
Siemens Corporate Research, Inc., Software engineering department
Company Confidential Report.
Princeton, USA, Dec. 1992

[STA92] Stallings, W.
ISDN and Broadband ISDN -second edition-

[WOO92] Wood, J.
Automatic Test Generation Software Tools
Siemens Corporate Research
Internal Report.
Princeton, USA, Dec. 1992

[BEE93] Beer, A.
Bridging The Gap Between Specs And Tests: ATG - a test procedure for
application-oriented user interfaces
[PAR93] Parish, A.S. and R.B. Borie; D.W. Cordes
Automated Flow Graph-Based Testing Of Object-Oriented Software Modules.

[MIC93] Microsoft corporation
Microsoft corporation, USA, 1993

[PUG93] Pugh, T.
C User Interface Library
Wilmslow England: Sigma Press, 1993

[TAY93] Tay, Y.C.
What is a deadlock?
IFIP Transactions A
Palma de Mallorca, Spain, 13-17 September 1993
Netherlands: ISSN, 1993, P. 49-60

[JUT94] Jüttner, P. and S. Kolb; S. Sieber; P. Zimmerer
Testing Major Object-Oriented Software Systems
Siemens Review - R&D Special, Spring 1994, P. 25-29

[TIJ94] Tijms, H.C.
Stochastic Models: An algorithmic approach
Chichester England: Wiley, 1994

[VRA94] Vrancken, H.P.E. and M.P.J. Stevens; M.T.M. Segers; J.H.M.M. van Rhee
System-Level Testability of Hardware/Software Systems
In: Proceedings 4th International Test Conference 1994,
Washington, D.C. USA, 2-6 October 1994
Philadelphia, USA, IEEE, 1994, P. 134-142

Automatic Test Generation V1.0 Prototype.
Siemens Corporate Research, Inc., Software engineering department
Company Confidential Report.
Princeton USA, Jul. 1994

[LAN94] Lansdale, M.W. and T.C. Ormerod
Understanding Interfaces
A Handbook of Human-Computer Dialogue
San Diego, USA: Academic Press, 1994

[HÜS95] Hüskes, R.
Umtauschrecht für Pentium-Prozessoren.
CT Magazin für Computer Technik, Germany, 1995, No. 2, P. 17
Appendix I: Project Description and graduation assignment

Before the OOTG project was started, a description of it was made by Mr W. Römer. Based on this, a graduation assignment was formed. This appendix contains the original texts.

I-1 Project Description

**Project: Design and Development of a Object-Oriented Test Generator**

There is currently no known technique which allows a flexible automatic generation of test cases for functional tests. The goal of this project is to design and develop such a test generator based on the object oriented capabilities of C++.

The final system will be a class library which provides a general class for objects to be tested. By methods of class inheritance, association and aggregation, other test cases can be defined by the test designer. The resulting test classes would be used to create real test objects in any degree of abstractness.

The test classes would provide methods to link test objects in order to implement a state transition graph. The state transitions could be class specific as well as object specific, allowing a great flexibility of the test object specification. Each state transition would contain the stimulus (procedure) which would be used to reach that state as well as a test (procedure) which would verify the correct state.

Additional functions in the class library then allow the generation of test cases: this would be implemented by using path coverage algorithms on the state transition graph.

Further functionality could be made available:

- analyses of the state transition graph in order to detect unreachable states, undesired loops, etc.
- creation of test lists
- a graphical representation of the state transition graph
- optimization on the number of test cases by using minimal coverage strategies or sampling
- a test case management system which would report differences before and after changes in the test objects in order to selectively run test cases

This system would enable the test designer to specify test cases with any detail, from high level test plans down to ready-to-run test scripts for automatic test machines.
I-2 Graduate Assignment

Description of the graduate assignment

The subject of the graduation work will be the project „Design and Development of a Object-Oriented Test Generator“. Since the whole project is too big to be finished in seven months, the assignment exists only in the first half of it.

This half will contain the following parts:

- Research to what there is to be tested in an arbitrarily object.
- Research to how the tests can be defined so that an optimal use of the object-oriented capabilities of C++ is reached.
- Development of a standard which prescribes in accordance to what rules tests have to be defined. The development of this standard will be on the basis of the earlier research.
- Development of a prototype tester which can execute the tests that are defined in accordance with the rules of the standard.

The goal of the graduation work will be: „The development of a prototype generic object oriented test interface standard“.
Appendix II: OOTG applied on the program ‘Clock’

To prove the theory of OOTG, the method was applied in practice on a small program named ‘Clock’. The user interface of Clock is based on the well known MS Windows. User interfaces under Windows have a very standard form. To reach an optimal reuse of code it seems logical to define the standard qualities in a separate library, that can be used in describing any Windows application. Before we take a closer look at Clock, we first have to treat Windows in general.

II-1 The standard classes for MS-Windows

MS Windows itself is programmed in an Object-Oriented way. A lot of standard objects of Windows can easily be defined as item classes for OOTG. An item class can contain

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Attributes</th>
<th>Stimuli</th>
<th>Child Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckButton</td>
<td>const char· Caption int CheckedFlag int EnabledFlag int FocusFlag</td>
<td>ClickButton</td>
<td></td>
</tr>
<tr>
<td>ComboBox</td>
<td>int EnabledFlag</td>
<td></td>
<td>ComboEditBox ComboListItems</td>
</tr>
<tr>
<td>ComboEditBox</td>
<td>const char· Caption const char· Text int EnabledFlag</td>
<td>ComboEditEnter</td>
<td></td>
</tr>
<tr>
<td>ComboListItem</td>
<td>const char· ListName const int IdNr const char· Text int FocusFlag</td>
<td>ComboItemSelect</td>
<td></td>
</tr>
<tr>
<td>EditBox</td>
<td>const int Idnr char· Text int EnabledFlag</td>
<td>EditEnter</td>
<td></td>
</tr>
<tr>
<td>ListItem</td>
<td>const int ListNr const int IdNr const char· Text int FocusFlag</td>
<td>ListItemSelect</td>
<td></td>
</tr>
<tr>
<td>MenuItem</td>
<td>const int IdNr const char· Text int EnabledFlag int CheckedFlag int FocusFlag</td>
<td>MenuItemClick MenuItemDoubleClick</td>
<td></td>
</tr>
<tr>
<td>OptionButton</td>
<td>const int IdNr int EnabledFlag int SelectedFlag int FocusFlag</td>
<td>OptionSelect</td>
<td></td>
</tr>
<tr>
<td>PushButton</td>
<td>const int IdNr const char· Caption int EnabledFlag const int DefaultFlag int FocusFlag</td>
<td>ButtonClick ButtonDoubleClick</td>
<td></td>
</tr>
<tr>
<td>StaticControl</td>
<td>const int StaticCtrlNr const char· Text</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table II-1: The standard items for a Windows’ interface.
attributes, methods, Stimuli and Child Items. The qualities, that are defined in an Item class, can be handed over to any TestObject. In table II-1 the item classes for Windows are stated together with the associated attributes, stimuli and child items. In the table we can see what item classes there are under Windows and what attributes, stimuli and child items belong to them. The next step is to see how these can be defined.

**Defining item classes:**
First of all, every item class is a subclass of the basic class ‘Item’ under OOTG. A subclass is an extension of a basic class. Just by adding a few things to this basic class a subclass is formed.

The basic class ‘Item’ contains a selftest, an empty list of stimuli and an empty list of child items. In every subclass attributes, methods and a selftest can be defined directly in the class definition. Stimuli and child items are created separately, they are put into the linked lists of the item during run time. When an instance of a certain item class is constructed, the related stimuli and child items have to be constructed as well. Under C++ the so called constructor function can be used, to do this automatically.

**Defining stimulus classes:**
Every stimulus class is a subclass of the basic class ‘TestStimulus’. This basic class contains an empty list of MethodCalls. A simulation of user actions can be defined directly in a subclass. MethodCalls are constructed during run time. It is impossible to predefine these in a standard library, since in every different application other methods have to be called. A MethodCall contains a pointer to a method. This pointer can only be created as the method already exists. It is most practical not to define MethodCalls before all objects, items and stimuli are created.

The development of an Object-Oriented prototype for any Windows application will exist now of the following steps:

1) **Distinguish the TestObjects in the user interface**
2) **Look what standard (child-)items every TestObject contains**
3) **Look if there are extra stimuli or attributes that are of importance and were not defined yet in any item**
4) Create the TestObjects: If there were no extra attributes found under step 3, a TestObject can be created as a direct instance of the basic class ‘TestObject’. In the other case a subclass has to be defined, that contains the extra attributes and a new selftest.

5) Define extra stimuli classes: If there were extra stimuli found under step 3 these have to be defined as subclasses of the basic ‘TestStimulus’.

6) Create all items and extra stimuli and put them in the linked lists of the TestObjects

7) Create all MethodCalls and put them in the linked lists of the stimuli: Every MethodCall is a direct instance of the basic class ‘MethodCall’.

II-2 The prototype of the interface of Clock

Using the steps that were defined in chapter II-1 a prototype of the user interface of the program clock was defined. The first three steps will be treated in this chapter. In the last four steps the actual implementation is done. For those interested, the programming code is listed in appendix III.

Step 1: Distinguish the TestObjects in the user interface

The user interface of the program Clock is built of five different Windows. These can be seen as the TestObjects for our prototype. In figure II-3 all these objects are drawn.
Step 2: Look what standard (child-)items every TestObject contains

Now we know what TestObjects we have we can start to define them in detail. As we will see, the item classes will play a mayor role in doing this.

In the TestObjects of Clock the following standard items were found:

Object 1:  
- MenuItem 'Settings'

Object 2:  
- MenuItem 'Analog'
- MenuItem 'Digital'
- MenuItem 'Set Font...'
- MenuItem 'No Title'
- MenuItem 'Seconds'
- MenuItem 'Date'
- MenuItem 'About Clock...'

Object 3:  
- Button ‘OK’
- Button ‘Cancel’
- ComboBox ‘Font’
- ComboBoxItem ‘Algerian’
- ComboBoxItem ‘Arial’ (child items)
- ComboEditBox
- StaticControl ‘Font:

Object 4:  
- Button ‘OK’
- StaticControl ‘’
- StaticControl ‘Clock’
- StaticControl ‘Microsoft Windows for Workgroups’
- StaticControl ‘Version 3.11’

Object 5:  
This object contains no standard items.

Step 3: Look if there are extra stimuli or attributes that are of importance and were not defined yet in any item

There are no extra attributes that have to be tested. All qualities of Clock we were interested in are defined in the items. Extra stimuli were found in the first and fifth TestObject. For both the keystroke ‘ESC’ was defined as stimulus and put in the linked lists of the TestObject.

In the steps 4-7 the actual implementation under C++ is done. The programming code, that was used for this, is listed in appendix III.
II-3 The Dummy Test Driver applied on Clock

As was described in chapter 6 of the report, the Dummy Test Driver does not test directly. In a first run the prototype of the interface is executed and a WTT test script is created. In a second run this script is used to test the real interface. To make this all possible, all SelfTest functions in the prototype had to be changed to WriteSelfTest functions. By calling one of these, a test is written in a test script instead of directly executed.

The Dummy Test Driver expects a seed for the pseudo random generator and a number of test steps as input. During the run a test script is written. At the end, there is an output in which the reached stimulus coverage is stated. As said in chapter 5.7.1, the minimal demand for a test is that all stimuli are used at least once. This means that the coverage must be hundred percent. Experience shows, that, depending on the seed that is used, 130 to 1000 test steps are necessary to reach this for clock.

In a test it is the best, if all stimuli are called under as many different states as possible. This goal is optimally reached, if there is a uniform distribution of the calls over the different stimuli. As we will see in the next paragraph, the Dummy Test Driver shows very poor results on this aspect.

II-3.1 A Markov model of the prototype interface of Clock

To calculate the distribution of calls over the different stimuli, we have to calculate what the chance of all stimuli is to be called at a certain moment. If we can determine the chances for the objects to be active, the chances for the stimuli to be called easily follow. We first have to take a look at what different stimuli there are in what object. Another important aspect is how the activity of an object can be handed over to others. In figure II-4 a graph is drawn of the interface of Clock. In this graph the TestObjects are the nodes and the transitions of the activity, caused by the stimuli, form the branches.

To calculate the chances for the TestObjects to be active, a Markov model [TIJ94] can be used. The Markov model of clock is in fact a graph in which the nodes are the TestObjects, and the branches are the chances that the activity from one object is handed over to another. The definition of a Markov model for Clock would be very easy to do, if there were no conditional stimuli. The third stimulus of object 2 however cannot be used at any time.

Clock has two modes. In one mode the clock is analog (selected by stimulus 2.1) and in the other mode it is digital (selected by stimulus 2.2). Only in the digital mode Stimulus 2.3 ('Set font...') can be called. Without this stimulus the third object becomes unreachable. The problem, that arises in the Markov model, was solved by defining two nodes for every TestObject. One node stands for the object in the digital mode and the other for the same object in the analog mode. The resulting Markov graph is drawn in figure II-5.
From the graph we can derive the chances for the objects to be active in the two modes. If we define that \( P(j) \) is the chance that object \( j \) is active, then follows:

\[
\begin{align*}
P(2) &= \frac{1}{2} P(1) \\
P(3) &= \frac{1}{7} P(2) + \frac{2}{3} P(3) \\
P(4) &= \frac{1}{7} P(2) \\
P(5) &= \frac{1}{2} P(1) \\
P(1') &= \frac{1}{7} P(2) + \frac{2}{3} P(2') + P(4') + P(5') \\
P(2') &= \frac{1}{2} P(1') \\
P(3') &= 0 \\
P(4') &= \frac{1}{6} P(2') \\
P(5') &= \frac{1}{2} P(1') \\
P(1) + P(2) + P(3) + P(4) + P(5) + P(1') + P(2') + P(3') + P(4') + P(5') &= 1
\end{align*}
\]
Further derivation leads for the objects to the following chances of being active:

\[
\begin{align*}
P(1) &= \frac{14}{57} & P(1') &= \frac{12}{57} \\
P(2) &= \frac{7}{57} & P(2') &= \frac{6}{57} \\
P(3) &= \frac{3}{57} & P(3') &= 0 \\
P(4) &= \frac{1}{57} & P(4') &= \frac{1}{57} \\
P(5) &= \frac{7}{57} & P(5') &= \frac{6}{57}
\end{align*}
\]

Within one TestObject the chances for the enabled stimuli to be called are equal. So, if we divide the chance for a TestObject to be active through its number of stimuli, we find the chance per stimulus to be called. The results of the derivations are listed in table II-2.

As can be concluded from the table, the distribution of the chances over the different stimuli, is far from uniform. On the average the stimuli of object 1 are called twenty-six times as often as the ones from object 3! With systematical methods instead of a pseudo random generator, better distributions must be reachable.
## Tabel II-2: The chances for the stimuli to be called.

<table>
<thead>
<tr>
<th>Stimulus Nr.</th>
<th>Analog Mode</th>
<th>Digital Mode</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>7/57</td>
<td>6/57</td>
<td>13/57</td>
</tr>
<tr>
<td>1.2</td>
<td>7/57</td>
<td>6/57</td>
<td>13/57</td>
</tr>
<tr>
<td>2.1</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>2.2</td>
<td>1/57</td>
<td>0</td>
<td>1/57</td>
</tr>
<tr>
<td>2.3</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>2.4</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>2.5</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>2.6</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>2.7</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>3.1</td>
<td>1/114</td>
<td>0</td>
<td>1/114</td>
</tr>
<tr>
<td>3.2</td>
<td>1/114</td>
<td>0</td>
<td>1/114</td>
</tr>
<tr>
<td>3.3</td>
<td>1/114</td>
<td>0</td>
<td>1/114</td>
</tr>
<tr>
<td>3.4</td>
<td>1/114</td>
<td>0</td>
<td>1/114</td>
</tr>
<tr>
<td>3.5</td>
<td>1/114</td>
<td>0</td>
<td>1/114</td>
</tr>
<tr>
<td>3.6</td>
<td>1/114</td>
<td>0</td>
<td>1/114</td>
</tr>
<tr>
<td>4.1</td>
<td>1/57</td>
<td>1/57</td>
<td>2/57</td>
</tr>
<tr>
<td>5.1</td>
<td>7/57</td>
<td>6/57</td>
<td>13/57</td>
</tr>
</tbody>
</table>

**Remark:** The Markov analyses, that were used in this appendix, are only correct for a number of test steps that is large enough to cover the initial start conditions. For smaller numbers of test steps the chances in table II-2 are just an approach.

### II-4 Evaluation

The definition of an Object-Oriented prototype of the user interface of clock was not difficult to do. The use of a library of standard items for Windows was very effective. The most time consuming job was the definition of MethodCalls. The problem in doing this is that multiple pointers are needed to specify a method that has to be called. In some cases more than one hundred characters were needed for the definition of just one MethodCall. An improvement might be found in the use of a kind of matrix. In the matrix, the stimuli of the interface are used as rows and the MethodCalls as columns. In a matrix like this we can mark what methods have to be called for the execution of what stimuli.

By the use of a pseudo random test routine a hundred per cent stimulus coverage was reached after a relative large number of test steps. As can be concluded from the Markov analysis, the calls are not uniformly spread over the different stimuli. Better results must be reachable by the use of a systematic test routine.
Appendix III: Program code of the Dummy Test Driver

This appendix contains the program code that was used to prove the basic methods of OOTG. The code is spread over eighteen files. These are structured in three groups of files with similar functions.

The groups are:
1) OOTG Basic Classes
2) Standard Classes for MS Windows
3) Remaining Files

1) OOTG Basic Classes: This group contains files in which the four basic OOTG classes are defined and implemented.

There are seven files:
- OOTG.h: Definition of the classes TestObject, Item, Stimulus and MethodCall.
- GLOBAL.h: Definition of a global pointer to the first TestObject.
- TESTOBJE.cpp: Implementation of the class ‘TestObject’.
- ITEM.cpp: Implementation of the class ‘Item’.
- STIMULUS.cpp: Implementation of the class ‘Stimulus’.
- METHODCA.cpp: Implementation of the class ‘MethodCall’.
- GLOBAL.cpp: Definition of some functions that are used in the other files of this group.

2) Standard Classes for MS Windows: this group is a library that can be used for any Windows application. It contains definitions for the item classes that were used in the program ‘Clock‘. All stimuli in these classes are defined as well. Since in clock not all standard Windows' items were used, this library is not complete. For other applications an extension might be necessary.

There are nine files:
- WINITEM.h: Definition of all item and stimulus classes for Windows.
- BUTTON.cpp: Implementation of the class ‘Button’.
- COMBOBOX.cpp: Implementation of the class ‘ComboBox’.
- COMBOEDT.cpp: Implementation of the class ‘ComboEdit’.
- COMBOITM.cpp: Implementation of the class ‘ComboItem’.
- EDITBOX.cpp: Implementation of the class ‘EditBox’.
- MENUITEM.cpp: Implementation of the class ‘MenuItem’.
- STATIC.cpp: Implementation of the class ‘StaticControl’.
- WINSTIM.cpp: Implementation of the used stimulus classes.

3) Remaining Files: There are two files left that do not belong to the first two groups:
- CLOCK.cpp: Definition of the prototype of the interface of clock.
- DUMMY.cpp: Definition of functions that are necessary for the Dummy Test Driver. This file also contains the function ‘Main’.

In the rest of this appendix the programming code is listed.
1) OOTG Basic Classes

OOTG.h

/////////////////////////////////////////////////////////////////////

// This file is the main header of the software of OOTG. It contains the definition of the
// classes: TestObject, Stimulus, TestAttribute and MethodCall. The implementation of these classes
// is done in the files: TestObject.cpp, Stimulus.cpp, TestAttribute.cpp and MethodCall.cpp.

#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Forward declaration of classes
class TestClass; // Does not belong to ootg. Is just for testing.
class TestCall; // Does not belong to ootg. Is just for testing.
class TestObject;
class Stimulus;
class MethodCall;
class Item;
extern TestObject* FirstTestObject;

// Some global functions defined in global.cpp
extern TestObject* GetLastTestObject();
extern Item* GetLastItem();
extern Stimulus* GetLastStimulus();

/////////////////////////////////////////////////////////////////////

// The class MethodCall: see methodca.cpp for the implementation.

class MethodCall {

public:

    MethodCall( Stimulus* parent, TestObject* testobjectptr, void (*methodptr) ());
    MethodCall( Stimulus* parent, TestObject* testobjectptr, void (*methodptr) (int), int nr);
    MethodCall( Stimulus* parent, TestObject* testobjectptr, void (*methodptr) (char*), char* text);
    MethodCall( Stimulus* parent, Item* itemptr, void (*methodptr) ());
    MethodCall( Stimulus* parent, Item* itemptr, void (*methodptr) (int), int nr);
    MethodCall( Stimulus* parent, Item* itemptr, void (*methodptr) (char*), char* text);

    ~MethodCall(); // Destructor

// A pointer to the Item or TestObject to which the Method belongs. Only one of the pointers
// may be not NULL.
    Item* itemPtr;
    TestObject* TestObjectPtr;

// An integer 'type' is used to define if the method belongs to an Item or to an TestObject and


// if the method uses no parameter, an integer, or a character string.
    int type;

// The optional parameters for the method.
    int Nr;
    char* Text;

    void (*MethodPtr) (...); // A pointer to a member function.
    void Execute(); // The actual calling of the method
    MethodCall* next; // Pointer to the next MethodCall

比亚迪

class Stimulus {

public:
    Stimulus::Stimulus(Item* parent);
    Stimulus::Stimulus(TestObject* parent);
    Stimulus::Stimulus(); // Constructors

    MethodCall* MethodCallList; // Pointer to the list of methods that have to be called

    void AddMethodCall(MethodCall* NewMethodCall); // Add a new MethodCall to the stimulus

    virtual int Enabled(); // Tests if all conditions are fullfilled. The return values are 0 if 1 or
        // more conditions are not fullfilled, 1 if all conditions are fullfilled
        // and 2 if no conditions are defined.

    int MethodCount(); // Returns the number of methods in the list of method calls.

    char* CallCommand; // A command that can be written in a test script.

    void Call(); // Calls the methods in the list & writes the CallCommand in a Script

    virtual void WriteCallCommand(); // In the first version of OOTG no commands are sent directly
        // to the interface.

    int HitNr; // This flag counts how often the stimulus was called.
        // It can be used for keeping statistics

    Stimulus* next; // Pointer to the next stimulus

private:
    virtual void UserSimulation(); // The simulation of user actions (not used in the first
        // version of OOTG)

};

比亚迪

class TestObject {

public:
    TestObject(char* name, int activeflag = 0); // Constructors
~TestObject(); // Destructor

int ActiveFlag; // Active Flag
TestObject *next; // Pointer to the next TestObject
Stimulus* StimulusList; // Pointer to the linked list of stimuli
Item* ItemList; // Pointer to the linked list of items.
char* Name;

void __near __cdecl SetActiveFlag (int activeflag);

int StimuliCount(); // Returns the number of Stimuli in the TestObject.
int StimuliHitCount(); // Returns the number of Stimuli in the TestObject that
                        // were called at least once.
int StimuliEnabledCount(); // Returns the number of stimuli in the TestObject that
                            // are enabled

void DisplayStimuli(); // Display info about all stimuli
void AddStimulus(Stimulus* NewStimulus); // Add a Stimulus to the linked list of stimuli
void AddItem(Item*NewItem); // Add an item to the linked list of Items.

void SelfTest(); // Let the TestObject check itself and all its items and their children.
virtual void WriteSelfTest(); // Write a command in a test script to test the TestObject.

Item* GetItem(int ItemNr);
Stimulus* GetStimulus(int StimulusNr); // Returns a pointer to the stimulus that is defined
                                        // by number. All stimuli are taken into account.
Stimulus* GetStimulusEnabled(int Nr); // Returns a pointer to the stimulus that is defined
                                       // by number. Only stimuli that are enabled are taken
                                       // into account.

class Item 
{
public:

    Item(TestObject* parent); // Constructors
    Item(Item* parent); 
    Item(); 
    ~Item(); // Destructor

    Item* next; // Pointer to the next Item
    Stimulus* StimulusList; // Pointer to the linked list of stimuli
    Item* ChildList; // Pointer to the linked list of child items

    void AddStimulus(Stimulus* NewStimulus); // Add a Stimulus to the linked list of stimuli
    void AddItem(Item* NewItem); // Add an item to the linked list of Items.
    Item* GetChild(int ChildNr);

    char* Type;
    char* Name;

};
// TestCommands: In the first version of OOTG, tests are not executed directly, but they are
// written in a test script. (The function SelfTest is for future use. At this moment WriteSelfTest
// is used.)

virtual void SelfTestO(); // Let the Item check itself.
    virtual void WriteSelfTestO(); // Write a command in a test script to test the Item.

private:

};

GLOBAL.h

/******
// Global pointer to the first TestObject

TestObject* FirstTestObject = NULL;

TESTOBJE.cpp

/******
// Implementation of the functions of the class TestObject.

#include "ootg.h"
extern FILE* script;

/******
// Global pointer to the first TestObject

TestObject* FirstTestObject = NULL;

/******
// The function SetlsActive(int ActiveFlag) passes a value to the flag IsActive

void TestObject::SetActiveFlag(int activeflag)
{
    if (activeflag == 0) {
        ActiveFlag = activeflag;
    } else {
        ActiveFlag = (ActiveFlag * activeflag);
    }
}

/******
// Return the pointer to a stimulus with a certain number. Return NULL if it doesn't exist.

Stimulus* TestObject::GetStimulus(int StimulusNr)
{
    int StimulusCount = 0;
    Stimulus* StimulusPtr;

    // Look for stimuli in the testobject itself.
    if (StimulusPtr = this->StimulusList) != NULL)
    {
        while(StimulusPtr)
{  
    StimulusCount++;  
    if( StimulusCount == StimulusNr )  
        return (StimulusPtr);  
    StimulusPtr=StimulusPtr->next;  
}

// Look for stimuli in the items.  
Item* ItemPtr;  
for (ItemPtr = this->ItemList; ItemPtr != NULL; )  
{
    if( (StimulusPtr = ItemPtr->StimulusList) != NULL)  
    {  
        while(StimulusPtr)  
        {  
            StimulusCount++;  
            if( StimulusCount == StimulusNr )  
                return (StimulusPtr);  
            StimulusPtr=StimulusPtr->next;  
        }
    }
    ItemPtr = ItemPtr->next;  
}

// Look for stimuli in the child items.  
Item* ChildPtr;  
for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )  
{
    if( (StimulusPtr = ChildPtr->StimulusList) != NULL)  
    {  
        for (; StimulusPtr != NULL;)
        {  
            StimulusCount++;  
            if( StimulusCount == StimulusNr )  
                return (StimulusPtr);  
            StimulusPtr=StimulusPtr->next;  
        }
    }
    ChildPtr = ChildPtr->next;
    ItemPtr = ItemPtr->next;  
}

cout << "ERROR: There is no stimulus with the number " << StimulusNr << " in TestObject " << this->Name << "\n";  
return(NULL);

Item* TestObject::GetItem(int ItemNr)  
{  
    Item* ItemPtr = ItemList;  
    for( int i = 1; i < ItemNr; i++ )  
    {  
        if( ItemPtr != NULL )  
            ItemPtr = (*ItemPtr).next;  
    }
    return( ItemPtr );
}

/*****************************************************************************  
// Output info about the stimuli.  
*****************************************************************************/
void TestObject::DisplayStimuli()
{
    fprintf(script, "%s%s", 
            "\nTestObject: ", Name);
    fprintf(script, "%s%i%s", "Number of stimuli enabled: ", 
            StimuliEnabledCount(), "\n");
    int StimuliCount = 0;
    Stimulus* StimulusPtr;

    // Look for stimuli in the testobject itself.
    if( (StimulusPtr = this->StimulusList) != NULL )
    {
        while(StimulusPtr)
        {
            StimuliCount++;
            fprintf(script, "%s%i%s", "Stimulus ", StimuliCount, ": ");
            StimulusPtr->WriteCallCommand();
            if( StimulusPtr->Enabled())
            {
                fprintf(script, "%s%i%s", "Enabled - Number of MethodCalls: ", 
                        StimulusPtr->MethodCount(), "\n");
            }
            else
            {
                fprintf(script, "%s%i%s", "Disabled - Number of MethodCalls: ", 
                        StimulusPtr->MethodCount(), "\n");
            }
            StimulusPtr=StimulusPtr->next;
        }
    }

    // Look for stimuli in the items.
    Item* ItemPtr;
    for (ItemPtr = this->ItemList; ItemPtr != NULL; )
    {
        if( (StimulusPtr = ItemPtr->StimulusList) != NULL )
        {
            while(StimulusPtr)
            {
                StimuliCount++;
                fprintf(script, "%s%i%s", "Stimulus ", StimuliCount, ": ");
                StimulusPtr->WriteCallCommand();
                if( StimulusPtr->Enabled())
                {
                    fprintf(script, "%s%i%s", "Enabled - Number of MethodCalls: ", 
                            StimulusPtr->MethodCount(), "\n");
                }
                else
                {
                    fprintf(script, "%s%i%s", "Disabled - Number of MethodCalls: ", 
                            StimulusPtr->MethodCount(), "\n");
                }
                StimulusPtr=StimulusPtr->next;
            }
        }
    }

    // Look for stimuli in the child items.
    Item* ChildPtr;
    for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
    {
        if( (StimulusPtr = ChildPtr->StimulusList) != NULL )
        {
            for (; StimulusPtr != NULL;)
            {
                StimuliCount++;
                fprintf(script, "%s%i%s", "Stimulus ", StimuliCount, ": ");
                StimulusPtr->WriteCallCommand();
                if( StimulusPtr->Enabled())
                {
                    fprintf(script, "%s%i%s", "Enabled - Number of MethodCalls: ", 
                            StimulusPtr->MethodCount(), "\n");
                }
            }
        }
    }
}
else
    fprintf(script, "%s\n", " Disabled - Number of MethodCalls: ",
    StimulusPtr->MethodCount(), "n");
    StimulusPtr=StimulusPtr->next;
}
ChildPtr = ChildPtr->next;
}
ItemPtr = ItemPtr->next;
}

/*---------------------------------------------------------------------------
// Count how many stimuli are Enabled.
///
int TestObject::StimuliCount()
{
    int StimulusCount = 0;
    Stimulus* StimulusPtr;

    // Look for stimuli in the testobject itself.
    if( (StimulusPtr = this->StimulusList) != NULL)
    {
        while(StimulusPtr)
        {
            StimulusCount++;
            StimulusPtr=StimulusPtr->next;
        }
    }
    // Look for stimuli in the items.
    Item* ItemPtr;
    for (ItemPtr = this->ItemList; ItemPtr != NULL; )
    {
        if( (StimulusPtr = ItemPtr->StimulusList) != NULL)
        {
            while(StimulusPtr)
            {
                StimulusCount++;
                StimulusPtr=StimulusPtr->next;
            }
        }
    }
    // Look for stimuli in the child items.
    Item* ChildPtr;
    for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
    {
        if( (StimulusPtr = ChildPtr->StimulusList) != NULL)
        {
            for (; StimulusPtr != NULL;)
            {
                StimulusCount++;
                StimulusPtr=StimulusPtr->next;
            }
        }
        ChildPtr = ChildPtr->next;
    }
    ItemPtr = ItemPtr->next;
}
return( StimulusCount );
}
// Count how many stimuli were used.
///
int TestObject::StimuliHitCount()
{
    int HitCount = 0;
    Stimulus* StimulusPtr;

    // Look for stimuli in the testobject itself.
    if( (StimulusPtr = this->StimulusList) != NULL)
    {
        while(StimulusPtr)
        {
            if(StimulusPtr->HitNr)
                HitCount++;
            StimulusPtr=StimulusPtr->next;
        }
    }

    // Look for stimuli in the items.
    Item* ItemPtr;
    for (ItemPtr = this->ItemList; ItemPtr != NULL; )
    {
        if( (StimulusPtr = ItemPtr->StimulusList) != NULL)
        {
            while(StimulusPtr)
            {
                if(StimulusPtr->HitNr)
                    HitCount++;
                StimulusPtr=StimulusPtr->next;
            }
        }
    }

    // Look for stimuli in the child items.
    Item* ChildPtr;
    for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
    {
        if( (StimulusPtr = ChildPtr->StimulusList) != NULL)
        {
            for (; StimulusPtr != NULL;)
            {
                if(StimulusPtr->HitNr)
                    HitCount++;
                StimulusPtr=StimulusPtr->next;
            }
            ChildPtr = ChildPtr->next;
        }
    }
    return( HitCount );
}

// Count how many stimuli are Enabled.
///
int TestObject::StimuliEnabledCount()
{
    int StimulusEnCount = 0;
    Stimulus* StimulusPtr;

    // Look for stimuli in the testobject itself.
    if( (StimulusPtr = this->StimulusList) != NULL)
    {
        while(StimulusPtr)
        {
            if(StimulusPtr->HitNr)
                StimulusEnCount++;
            StimulusPtr=StimulusPtr->next;
        }
    }

    // Look for stimuli in the items.
    Item* ItemPtr;
    for (ItemPtr = this->ItemList; ItemPtr != NULL; )
    {
        if( (StimulusPtr = ItemPtr->StimulusList) != NULL)
        {
            while(StimulusPtr)
            {
                if(StimulusPtr->HitNr)
                    StimulusEnCount++;
                StimulusPtr=StimulusPtr->next;
            }
        }
    }

    // Look for stimuli in the child items.
    Item* ChildPtr;
    for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
    {
        if( (StimulusPtr = ChildPtr->StimulusList) != NULL)
        {
            for (; StimulusPtr != NULL;)
            {
                if(StimulusPtr->HitNr)
                    StimulusEnCount++;
                StimulusPtr=StimulusPtr->next;
            }
            ChildPtr = ChildPtr->next;
        }
    }
    return( StimulusEnCount );
}
// Look for stimuli in the testobject itself.
if ( (StimulusPtr = this->StimulusList) != NULL)
{
    while(StimulusPtr)
    {
        if(StimulusPtr->Enabled() )
            StimulusEnCount++;
        StimulusPtr = StimulusPtr->next;
    }
}

// Look for stimuli in the items.
Item* ItemPtr;
for (ItemPtr = this->ItemList; ItemPtr != NULL; )
{
    if ( (StimulusPtr = ItemPtr->StimulusList) != NULL)
    {
        while(StimulusPtr)
        {
            if(StimulusPtr->Enabled() )
                StimulusEnCount++;
            StimulusPtr = StimulusPtr->next;
        }
    }
}

// Look for stimuli in the child items.
Item* ChildPtr;
for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
{
    if ( (StimulusPtr = ChildPtr->StimulusList) != NULL)
    {
        for (; StimulusPtr != NULL;)
        {
            if(StimulusPtr->Enabled() )
                StimulusEnCount++;
            StimulusPtr = StimulusPtr->next;
        }
    }
    ChildPtr = ChildPtr->next;
    ItemPtr = ItemPtr->next;
}
return( StimulusEnCount );


// Returns a pointer to the stimulus defined by Nr Only the stimuli that are enabled are
// taken into account.

/// Stimulus* TestObject::GetStimulusEnabled(int Nr)
{
    int StimulusEnCount = 0;
    Stimulus* StimulusPtr;

    // Look for stimuli in the testobject itself.
    if ( (StimulusPtr = this->StimulusList) != NULL)
    {
        while(StimulusPtr)
        {
            // Implementation code
        }
    }
}
if(StimulusPtr->Enabled())
    StimulusEnCount++;

if (StimulusEnCount == Nr)
    return(StimulusPtr);
    StimulusPtr=StimulusPtr->next;
}

// Look for stimuli in the items.
Item* ItemPtr;
for (ItemPtr = this->ItemList; ItemPtr != NULL; )
{
    if ( (StimulusPtr = ItemPtr->StimulusList) != NULL )
    {
        while(StimulusPtr)
        {
            if(StimulusPtr->Enabled())
                StimulusEnCount++;
            if (StimulusEnCount == Nr)
                return(StimulusPtr);
            StimulusPtr=StimulusPtr->next;
        }
    }
    ItemPtr = ItemPtr->next;
}

// Look for stimuli in the child items.
Item* ChildPtr;
for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
{
    if ( (StimulusPtr = ChildPtr->StimulusList) != NULL )
    {
        for (; StimulusPtr != NULL;)
        {
            if(StimulusPtr->Enabled())
                StimulusEnCount++;
            if (StimulusEnCount == Nr)
                return(StimulusPtr);
            StimulusPtr=StimulusPtr->next;
        }
    }
    ChildPtr = ChildPtr->next;
}

if (StimulusList == 0 )
    StimulusList = NewStimulus;
else
{
    Stimulus* StimulusPtr;
    for(StimulusPtr = StimulusList; StimulusPtr->next != NULL;
            StimulusPtr = StimulusPtr->next);
    StimulusPtr->next = NewStimulus;
}

================================================================================

// Add a Stimulus to the linked list of stimuli of the TestObject

void TestObject::AddStimulus(Stimulus* NewStimulus)
{
    if ( StimulusList == 0 )
        StimulusList = NewStimulus;
    else
    {
        Stimulus* StimulusPtr;
        for (StimulusPtr = StimulusList; StimulusPtr->next != NULL;
                StimulusPtr = StimulusPtr->next);
        StimulusPtr->next = NewStimulus;
    }
ADD AN ITEM TO THE LINKED LIST OF ITEMS OF THE TESTOBJECT

```cpp
void TestObject::AddItem(Item* NewItem)
{
    if (ItemList == NULL)
        ItemList = NewItem;
    else
    {
        Item* ItemPtr;
        for (ItemPtr = ItemList; (ItemPtr->next != NULL);)
            ItemPtr = ItemPtr->next;
        ItemPtr->next = NewItem;
    }
}
```

LET THE OBJECT CHECK ITSELF AND ALL ITS ITEMS AND THEIR CHILDREN.

```cpp
void TestObject::SelfTest()
{
    // Test the Object Itself.
    WriteSelfTest();

    Item* ItemPtr;
    for (ItemPtr = this->ItemList; ItemPtr != NULL; )
    {
        // Test the items.
        ItemPtr->WriteSelfTest();

        Item* ChildPtr;
        for (ChildPtr = ItemPtr->ChildList; ChildPtr != NULL; )
        {
            ChildPtr->WriteSelfTest();
            ChildPtr = ChildPtr->next;
        }
        ItemPtr = ItemPtr->next;
    }
}
```

THE FUNCTION WRITESELFTEST IS TO BE USED ONLY IF SELFTEST IS NOT IMPLEMENTED.

```cpp
void TestObject::WriteSelfTest()
{
    // Tests have to be defined at lower level.
}
```

CONSTRUCTOR FOR TESTOBJECTS.

```cpp
TestObject::TestObject(char* name, int activeflag)
{
    if (FirstTestObject == NULL)
```
FirstTestObject = this;
else
{
    TestObject* LastTestObject = GetLastTestObject();
    LastTestObject->next = this;
}

Name = name;
ActiveFlag = activeflag;
StimulusList = NULL;
ItemList = NULL;
next = NULL;


// Destructor for TestObjects.
#endif
TestObject::~TestObject()
{
    if( FirstTestObject == this )
        FirstTestObject->next = this->next;
    else
    {
        TestObject* TestObjectptr;
        for( TestObjectptr = FirstTestObject; TestObjectptr->next != this;
            TestObjectptr = TestObjectptr->next) {

            if ( TestObjectptr == NULL)
                ; // ERROR: Object Not Found
        }
        TestObjectptr->next = (*this).next;
    }
}

ITEM.cpp

#include "ootg.h"

Add a Stimulus to the linked list of stimuli of the Item

void Item::AddStimulus(Stimulus* NewStimulus)
{
    if( StimulusList == NULL )
        StimulusList = NewStimulus;
    else
    {
        Stimulus* Stimulusptr;
        for( Stimulusptr = StimulusList; StimulusPtr->next != NULL;)
            StimulusPtr = StimulusPtr->next;
        StimulusPtr->next = NewStimulus;
    }
The function SelfTest is not implemented in this version.

```cpp
void Item::SelfTest() // Let the Item check itself.
{
    cout << "ERROR: The Function SelfTest is not implemented in this version of OOTG."
;
}
```

The function WriteSelfTest is to be used only if SelfTest is not implemented.

```cpp
void Item::WriteSelfTest() // Write a command in a test script to test the Item.
{
    // Has to be overloaded at lower level
}
```

Add an Item to the linked list of Items of the parent item.

```cpp
void Item::AddItem(Item*NewItem)
{
    if( ChildList == NULL )
        ChildList =NewItem;
    else
        {
            Item* ItemPtr;
            for( ItemPtr = ChildList; (ItemPtr->next != NULL);)
                ItemPtr = ItemPtr->next;
            ItemPtr->next = NewItem;
        }
}
```

```cpp
Item* Item::GetChild(int ChildNr)
{
    Item* ItemPtr = ChildList;
    for( int i = 1; i < ChildNr; i++ )
    {
        if( ItemPtr != NULL )
            ItemPtr = (*ItemPtr).next;
    }
    return( ItemPtr );
}
```

Constructors for Items

```cpp
Item::Item (TestObject* parent) // The Parent is a TestObject
{
    next = NULL;
    StimulusList = NULL;
    ChildList = NULL;
    parent->AddItem(this);
}
```

```cpp
Item::Item (Item* parent) // The Parent is an Item
```
{  
    next = NULL;
    StimulusList = NULL;
    ChildList = NULL;
    parent->AddItem(this);
}

// The Parent is a not given as a parameter. The item is added to the last testobject.
{
    next = NULL;
    StimulusList = NULL;
    ChildList = NULL;
    TestObject* parent = GetLastTestObject();
    if (parent)
        parent->AddItem(this);
    else
        cout "ERROR: An item was added, but there was no TestObject available.");
}

// Destructor for Items
Item::Item()
{
}

// Implementation of the functions of the class Stimulus.
#include "ootg.h"
extern FILE* script;

// The function WriteCallCommand writes a command to call the user simulation in a test script,
// the function is used in the first version of OOTG.
void Stimulus::WriteCallCommand()
{
    TO DO: Write this command to a script
    HitNr++;
    // HitFlag counts how often the stimulus was called.
    fprintf(script,"%s", CallCommand);
}

// The function UserSimulation has to be redefined, otherwise an error occurs.
void Stimulus::UserSimulation() // The simulation of user actions
{
    // Error: This function is not used in the first version of OOTG. Use: WriteCallCommand.
}

// If the function Enabled() is not redefined, no conditions are tested and 2 is returned.

61
int Stimulus::Enabled()
{
    return(2);
}

void Stimulus::Call()
{
    if (MethodCallList != NULL)
    {
        for( MethodCall* Method = MethodCallList; Method != NULL; )
        {
            Method->Execute();
            Method = Method->next;
        }
        WriteCallCommand(); // In later version this will be UserSimulation();
    }
}

int Stimulus::MethodCount()
{
    int i = 0;
    if (MethodCallList != NULL)
    {
        for( MethodCall* Method = MethodCallList; Method != NULL; )
        {
            i++;
            Method = Method->next;
        }
        return(i);
    }
}

void Stimulus::AddMethodCall(MethodCall* NewMethodCall)
{
    if( MethodCallList == NULL)
        MethodCallList = NewMethodCall;
    else
    {
        MethodCall* MethodCallPtr = MethodCallList;
        while(MethodCallPtr->next != NULL)
        {
            MethodCallPtr = MethodCallPtr->next;
        }
        MethodCallPtr->next = NewMethodCall;
    }
}
// Constructors for Stimuli

// The Parent is a TestObject
Stimulus::Stimulus(TestObject* parent) {  
    MethodCallList = NULL;
    next = NULL;
    parent->AddStimulus(this);
}

// The Parent is an Item
Stimulus::Stimulus(Item* parent) {  
    MethodCallList = NULL;
    next = NULL;
    parent->AddStimulus(this);
}

// The Parent is the last item of the last testobject, or, if there is
// no item in this object, it is directly added to the TestObject.
Stimulus::Stimulus() {  
    MethodCallList = NULL;
    next = NULL;

    if (GetLastItem()) {  
        Item* parent = GetLastItem();
        parent->AddStimulus(this);
    } else {  
        if (GetLastTestObject()) {  
            TestObject* parent = GetLastTestObject();
            parent->AddStimulus(this);
        } else {  
            cout << "ERROR: A stimulus was defined, but there was no TestObject ";
            cout << "or Item available to add it to";
        }
    }
}

METHODCA.cpp

// Implementation of the functions of the class MethodCall.

#include "ootg.h"

// The actual calling of the method

void MethodCall::Execute() {  
    if (type>2) {  
        switch(type%3) {  
        {  

63
case 1: // There is a parameter of the type integer
    MethodPtr(ItemPtr, Nr);
    break;
case 2:
    MethodPtr(ItemPtr, Text);
    break;
default:
    MethodPtr(ItemPtr);
    break;
}
}
else
{
    switch( type%3 )
    {
    case 1: // There is a parameter of the type integer
        MethodPtr(TestObjectPtr, Nr);
        break;
case 2:
        MethodPtr(TestObjectPtr, Text);
        break;
default:
        MethodPtr(TestObjectPtr);
        break;
    }
}

Constructors for MethodCall

MethodCall::MethodCall(Stimulus* parent, TestObject* testobjectptr, void (*methodptr) ( ))
{
    TestObjectPtr = testobjectptr;  // This pointer is necessary, while a pointer to a member
    // function doesn't define to what instance of a class it
    // belongs.
    type = 0;  // Type 0: Method belongs to TestObject, there is no parameter.
    next = NULL;
    MethodPtr = (void (*)( ...)) methodptr;  // By casting the function pointer MethodPtr can point to
    // the parameterless function methodptr.
    parent->AddMethodCall( this );
}
MethodCall::MethodCall(Stimulus* parent, TestObject* testobjectptr, void (*methodptr) (int), int nr)
{
    TestObjectPtr = testobjectptr;  // This pointer is necessary, while a pointer to a member function
    // doesn't define to what instance of a class it belongs.
    type = 1;  // Type 1: Method belongs to a TestObject, there is an integer as parameter.
    Nr = nr;
    next = NULL;
    MethodPtr = (void (*)( ...)) methodptr;  // By casting the function pointer MethodPtr can point to
    // the parameterless function methodptr.
MethodCall::MethodCall(Stimulus* parent, TestObject* testobjectptr, void (*methodptr) (char*), char* text)
{
    TestObjectPtr = testobjectptr; // This pointer is necessary, while a pointer to a member function
    // doesn't define to what instance of a class it belongs.

    type = 2;  // Type 2: Method belongs to a TestObject, there is a string as parameter.
    next = NULL;

    Text = text;

    MethodPtr = (void (*)(...)) methodptr; // By casting the function pointer MethodPtr can point to
    // the parameterless function methodptr.

    parent->AddMethodCall( this );
}

MethodCall::MethodCall(Stimulus* parent, Item* itemptr, void (*methodptr) ()
{
    ItemPtr = itemptr;

    type = 3;  // Type 3: Method belongs to an Item, there is no parameter.
    next = NULL;

    MethodPtr = (void (*)(...)) methodptr; // By casting the function pointer MethodPtr can point to
    // the parameterless function methodptr.

    parent->AddMethodCall( this );
}

MethodCall::MethodCall(Stimulus* parent, Item* itemptr, void (*methodptr) (int), int nr)
{
    ItemPtr = itemptr;

    type = 4;  // Type 4: Method belongs to an Item, there is an integer as parameter.
    next = NULL;

    Nr = nr;

    MethodPtr = (void (*)(...)) methodptr; // By casting the function pointer MethodPtr can point to
    // the parameterless function methodptr.

    parent->AddMethodCall( this );
}

MethodCall::MethodCall(Stimulus* parent, Item* itemptr, void (*methodptr) (char*), char* text)
{
    ItemPtr = itemptr;

    type = 5;  // Type 5: Method belongs to an Item, there is a string as parameter.
    next = NULL;

    Text = text;

    MethodPtr = (void (*)(...)) methodptr; // By casting the function pointer MethodPtr can point to
    // the parameterless function methodptr.

    parent->AddMethodCall( this );
}

////////////////////////////////////////////////////////////////////////////////////
// Destructor for MethodCalls
////////////////////////////////////////////////////////////////////////////////////
MethodCall::MethodCall()
{
}

GLOBAL.cpp

#include "ootg.h"

TestObject* GetLastTestObject()
{
    TestObject* TestObjectPtr = FirstTestObject;
    if( TestObjectPtr->next )
        for(; TestObjectPtr->next; TestObjectPtr = TestObjectPtr->next);
    return( TestObjectPtr );
}

Item* GetLastItem()
{
    TestObject* LastTestObject = GetLastTestObject();
    if ( LastTestObject )
    {
        Item* ItemPtr = LastTestObject->ItemList;
        while( ItemPtr )
        {
            if (ItemPtr->next)
                for(; ItemPtr->next; ItemPtr = ItemPtr->next);
            if (ItemPtr->ChildList)
                ItemPtr = ItemPtr->ChildList;
            else
                return( ItemPtr );
        }
        return ( ItemPtr );
    }
    else
        return( NULL );
}

// This function returns the last stimulus of the last item.
Stimulus* GetLastStimulus()
{
    Item* LastItem = GetLastItem();
    if ( LastItem ) // There is an item in the last testobject.
    {
        Stimulus* StimulusPtr = LastItem->StimulusList;
        if (StimulusPtr->next)
            for(; StimulusPtr->next; StimulusPtr = StimulusPtr->next);
        return( StimulusPtr );
    }
    else // There is no item in the last TestObject, but there still might be a stimulus!
    {
        TestObject* LastTestObject = GetLastTestObject();
        if ( LastTestObject ) // There is a TestObject.
        {
            Stimulus* StimulusPtr = LastTestObject->StimulusList;
            if (StimulusPtr->next)
                for(; StimulusPtr->next; StimulusPtr = StimulusPtr->next);
            return( StimulusPtr );
        }
    }
2) Standard Classes for MS Windows

**WINITEM.h**

```c
#include "ootg.h"

// In this file the definitions of the item classes for Windows application are defined.
// The definition of stimuli classes for Windows is also done down here.

// Item Classes:
class Button;
class ComboBox;
class ComboEdit;
class ComboItem;
class EditText;
class MenuItem;
class StaticControl;

// Stimuli Classes:
class ButtonClick;
class ComboBoxEnter;
class ComboBoxSelect;
class EditTextEnter;
class MenuItemSelect;

// Item Classes

class Button : public Item
{
    public:
        Button ( TestObject* parent, int idnr, char* caption, int enabledflag = 1, int focusflag = 0,
                int defaultflag = 0);
        Button ( int idnr, char* caption, int enabledflag = 1, int focusflag = 0, int defaultflag = 0);

    void __near __cdecl SetEnabledFlag(int enabledflag);
    void __near __cdecl SetFocusFlag(int checkedflag);
    void __near __cdecl SetDefaultFlag(int defaultflag);

    void WriteSelfTest();

    friend class ButtonClick;

    private:

        int FlagsValue();

    // Attributes

```
int IdNr;
char* Caption;
int EnabledFlag;
int FocusFlag;
int DefaultFlag;
}

class ComboBox : public Item
{
    public:
    ComboBox ( TestObject* parent, int IdNr, char* caption, int enabledflag = 1, int focusitemnr = 1);
    ComboBox ( int IdNr, char* caption, int enabledflag = 1, int focusitemnr = 1);
    void WriteSelfTest();
    void __near __cdecl SetEnabledFlag(int enabledflag);
    void __near __cdecl SetFocusFlag(int ItemNr);

    friend class ComboItem;
    friend class ComboEdit;
    friend class ComboSelect;
    friend class ComboEnter;

    int FocusItemNr; // The number of the item that has the focus.

    private:
    int FlagsValue();

    // Attributes
    int IdNr;
    char* Caption;
    int EnabledFlag;
};

class ComboEdit : public Item
{
    public:
    ComboEdit ( ComboBox* parent, char* text );
    void WriteSelfTest();
    void __near __cdecl ChangeText(char* NewText);
    void AddEditText(char* text);
    char* Text;

    friend class ComboEnter;

    private:

    // Attributes
    ComboBox* Parent;
};
class ComboItem : public Item
{
    public:
    ComboItem ( ComboBox* parent, int IdNr, char* text, int focusflag = 0);
void WriteSelfTest();
void __near __cdecl SetFocusFlag(int focusflag);

friend class ComboSelect;
private:

    int FlagsValue();

// Attributes
    ComboBox* Parent;
    int IdNr;
    char* Text;
    int FocusFlag;
};

class EditBox : public Item
{
    public:
        EditBox ( TestObject* parent, int idnr, char* text, int enabledflag = 1 );
        EditBox ( int idnr, char* text, int enabledflag = 1 );

    void WriteSelfTestO;
    void __near __cdecl ChangeText(char* NewText);
    void __near __cdecl SetEnabledFlag(int enabledflag);

    void AddEditText(char* text);

    friend class EditEnter;
    private:

        int FlagsValue();

// Attributes
    int IdNr;
    char* Text;
    int EnabledFlag;
};

class MenuItem : public Item
{
    public:
        MenuItem ( TestObject* parent, int IdNr, char* Caption, int enabledflag = 1,
                   int checkedflag = 0 );

    void __near __cdecl SetEnabledFlag(int enabledflag);
    void __near __cdecl SetCheckedFlag(int checkedflag);
    void WriteSelfTest();

    friend class MenuItemSelect;
    private:

        int FlagsValue();

// Attributes
    int IdNr;
    char* Caption;
    int EnabledFlag;
    int CheckedFlag;
};
class StaticControl : public Item
{
    public:
        StaticControl ( TestObject* parent, int idnr, char* caption);
        StaticControl ( int idnr, char* caption);

        void WriteSelfTest();

    private:

    // Attributes
        int IdNr;
        char* Caption;
};

// Stimuli Classes, the implementation of these classes is done in WinStim.cpp

class DoKeys : public Stimulus
{
    public:
        DoKeys(TestObject* parent, char* keys);
        DoKeys( char* Keys);

    private:

    // redefinitions:

        void WriteCallCommand();
};

class ButtonClick : public Stimulus
{
    public:
        ButtonClick(Button* parent);
        Button* Parent;

        void WriteCallCommand();
        int Enabled();
};

class ComboEnter : public Stimulus
{
    public:
        ComboEnter(ComboEdit* parent, char* text);
        ComboEdit* Parent;
        char* Text;       // the text that has to be entered

    private:

    // redefinitions:

        void WriteCallCommand();
        int Enabled();
};

class ComboSelect : public Stimulus
{
    public:
        ComboSelect(ComboBox* parent);
        ComboBox* Parent;
 BUTTON.cpp

#include "witems.h"

extern FILE* script;

// Implementation of the functions of the class Button.

BUTTON::ButtonDown (TestObject* parent, int idnr, char* caption, int enabledflag, int focusflag, int defaultflag )
:Item(parent)
{

// Attributes
Type = "Button";
Name = caption;
IdNr = idnr;
Caption = caption;
SetEnabledFlag( enabledflag );
SetFocusFlag( focusflag );
SetDefaultFlag( defaultflag );

// Define a stimulus of the type ButtonClick.

new ButtonClick(this);
}

BUTTON::ButtonDown ( int idnr, char* caption, int enabledflag, int focusflag, int defaultflag )
:Item()
{

// Attributes

// Define a stimulus of the type ButtonClick.

    new ButtonClick(this);
}

void Button::WriteSelfTest()
{
    int flags = FlagsValue();
    fprintf(script, "%s%i%s%s%i%s", "WTIButton ", IdNr, ", ", Caption, ", ", flags, ",n");
}

int Button::FlagsValue()
{
    return( EnabledFlag + (8 * DefaultFlag) + (16 * FocusFlag));
}

void Button::SetEnabledFlag(int enabledflag)
{
    if (enabledflag == 0) else
        EnabledFlag = enabledflag;
    else
        EnabledFlag = (EnabledFlag&enabledflag);
}

void Button::SetFocusFlag(int focusflag)
{
    if (focusflag == 0) else
        FocusFlag = focusflag;
    else
        FocusFlag = (FocusFlag&focusflag);
}

void Button::SetDefaultFlag(int defaultflag)
{
    if (defaultflag == 0) else
        DefaultFlag = defaultflag;
    else
        DefaultFlag = (DefaultFlag&defaultflag);
}

COMBOBOX.cpp

#include "witems.h"

extern FILE* script;

// Implementation of the functions of the class ComboBox.

ComboBox::ComboBox ( TestObject* parent, int idnr, char* caption, int enabledflag, int focusitemnr )
    : Item(parent)
{
// Attributes
    Type = "ComboBox";
    Name = caption;
    IdNr = idnr;
    Caption = caption;
    EnabledFlag = enabledflag;
    FocusItemNr = focusitemnr;

ComboBox::ComboBox ( int idnr, char* caption, int enabledflag, int focusitemnr )
    : Item()
{

    Attributes
        Type = "ComboBox";
        Name = caption;
        IdNr = idnr;
        Caption = caption;
        EnabledFlag = enabledflag;
        FocusItemNr = focusitemnr;

void ComboBox::SetFocusFlag(int ItemNr)
{
    ComboItem* child = ((ComboBox*) ChildList);
    int count=0;
    FocusItemNr = ItemNr;
    while(child)
    {
        count++;
        if (count == ItemNr)
            child->SetFocusFlag(1);
        else
            child->SetFocusFlag(0);
        child = ((ComboBox*) child->next);
    }
}

void ComboBox::WriteSelfTest()
{
    int flags = FlagsValue();
    fprintf( script, "%s%i%s%s%s%i%s", "WTICombo ", IdNr, ", " , Caption, "\", flags, "\n");
}

int ComboBox::FlagsValue()
{
    return( EnabledFlag );
}

void ComboBox::SetEnabledFlag(int enabledflag)
{
    if ((enabledflag == 0)||(enabledflag == 1))
        EnabledFlag = enabledflag;
    else
        EnabledFlag = (EnabledFlag^enabledflag);
}
 ComboEdit::ComboEdit ( ComboBox* parent, char* text )
    : Item(parent)
{
    Parent = parent;   // Give the child its parents address.

    // Attribute
    Type = "ComboEdit";
    Text = text;
}

// Add a stimulus to the edit box. Every text that must be entered during a test
// is defined in a stimulus of the type EditEnter.
void ComboEdit::AddEditText(char* text)
{
    new ComboEnter(this, text);
}

void ComboEdit::ChangeText(char* NewText)
{
    Text = NewText;
}

void ComboEdit::WriteSelITestO()
{
    int flags = FlagsValue();
    fprintf( script, "%s%i%s%i%s%i%s", "WTIComboEdit ", Parent->ldNr, ", ", IdNr, ", ", Text, ", Text,", Text, "flags, "in");
}

COMBOITEM.cpp

#include "winitems.h"

extern FILE* script;

/// Implementation of the functions of the class ComboItem.

ComboItem::ComboItem ( ComboBox* parent, int idnr, char* text, int focusflag )
    : Item(parent)
{
    Parent = parent;   // Give the child its parents address.

    // Attributes
    Name = text;
    Type = "ComboItem";
    IdNr = idnr;
    Text = text;
    SetFocusFlag( focusflag );

    // Stimulus: ComboSelect.
    new ComboSelect(this);
}

void ComboItem::WriteSelITestO()
{
    int flags = FlagsValue();
    fprintf( script, "%s%i%s%i%s%i%s", "WTIComboItem ", Parent->ldNr, ", ", IdNr, ", ", Text, ", Text, "flags, "in");
}

int ComboItem::FlagsValue()
{  return( FocusFlag * 16); }

void ComboItm::SetFocusFlag(int focusflag)
{
  if (focusflag == 0) || (focusflag == 1)
    FocusFlag = focusflag;
  else
    FocusFlag = (FocusFlag * focusflag);
}

EDITBOX.cpp

#include "winitems.h"
extern FILE* script;

// Implementation of the functions of the class EditBox.

EditBox::EditBox ( TestObject* parent, int idnr, char* text, int enabledflag ) :
Item(parent)
{
  // Attributes
  Type = "EditBox";
  IdNr = idnr;
  Text = text;
  EnabledFlag = enabledflag;
}

EditBox::EditBox ( int idnr, char* text, int enabledflag ) :
Item()
{
  // Attributes
  Type = "EditBox";
  IdNr = idnr;
  Text = text;
  EnabledFlag = enabledflag;
}

// In an editbox the number of stimuli is not predefined. For every text that has to be entered
// during tests, a new stimulus has to be defined. The function AddEditText can be called for
// doing this.

void EditBox::AddEditText(char* text)
{
  new EditEnter(this, text);
}

void EditBox::ChangeText(char* NewText)
{
  Text = NewText;
}

void EditBox::WriteSelfTest()
{
#include "winitems.h"

extern FILE* script;

// Implementation of the functions of the class Menultem.
Menultem::Menultem ( TestObject* parent, int idnr, char * caption, int enabledflag, int checkedflag) :
    Item(parent)
{
    // Attributes
    Type = "Menultem";
    Name = caption;
    IdNr = idnr;
    Caption = caption;
    EnabledFlag = enabledflag;
    CheckedFlag = checkedflag;

    // Define a stimulus of the type MenultemSelect.
    new MenultemSelect(this);
}

Menultem::Menultem ( int idnr, char * caption, int enabledflag, int checkedflag)
{
    // Attributes
    Type = "Menultem";
    Name = caption;
    IdNr = idnr;
    Caption = caption;
    EnabledFlag = enabledflag;
    CheckedFlag = checkedflag;

    // Define a stimulus of the type MenultemSelect.
    new MenultemSelect(this);
}
void MenuItem::WriteSelfTest()
{
    int flags = FlagsValue();
    fprintf(script, "%s%i%s%s%i%s", "WTIMenultemTest", IdNr, ",", Caption, ",", flags, ",n");
}

int MenuItem::FlagsValue()
{
    if (CheckedFlag == 1)
        return( EnabledFlag + 2);
    else
        return( EnabledFlag + 32);
}

void MenuItem::SetEnabledFlag(int enabledflag)
{
    if (enabledflag == 0) || (enabledflag == 1)
        EnabledFlag = enabledflag;
    else
        EnabledFlag = (EnabledFlag^enabledflag);
}

void MenuItem::SetCheckedFlag(int checkedflag)
{
    if (checkedflag == 0) || (checkedflag == 1)
        CheckedFlag = checkedflag;
    else
        if(CheckedFlag)
            CheckedFlag = 0;
        else
            CheckedFlag = 1;
}

STATIC.cpp

#include "witems.h"

extern FILE* script;

// Implementation of the functions of the class Button.

StaticControl::StaticControl ( TestObject* parent, int idnr, char* caption)
        :Item(parent)
{
    // Attributes
    Type = "StaticControl";
    Name = caption;
    IdNr = idnr;
    Caption = caption;
}

StaticControl::StaticControl ( int idnr, char* caption)
        :Item()
{
    // Attributes
    Type = "StaticControl";
    Name = caption;
    IdNr = idnr;
    Caption = caption;
}
WINSTIM.cpp

#include "winitems.h"

extern FILE* script;

/***********************************************************************************/
// The implementation of the Stimuli Classes.
/***********************************************************************************/

// Methods belonging to the class DoKeys

DoKeys::DoKeys(TestObject* parent, char* keys)
{
    HitNr = 0;
    Parent = parent;
    Keys = keys;
    CaliCommand = "DoKeys ";
}

void DoKeys::WriteCaliCommand()
{
    HitNr++;
    // HitNr counts how often the stimulus was called.
    fprintf( script, "%s%i%s", CaliCommand, Keys, "\n" );
}

// Methods belonging to class ButtonClick

ButtonClick::ButtonClick(Button* parent)
:Stimulus(parent)
{
    HitNr = 0;
    Parent = parent;
    CaliCommand = "WTTButtonClick ";
}

void ButtonClick::WriteCaliCommand()
{
    HitNr++;
    // HitNr counts how often the stimulus was called.
    fprintf( script, "%s%i%s", CaliCommand, Parent->ldNr, "\n" );
}

int ButtonClick::Enabled()
{
    return( Parent->EnabledFlag);
}

// Methods belonging to the class ComboEnter

ComboEnter::ComboEnter(ComboEdit* parent, char* text)
:Stimulus(parent)
{
    HitNr = 0;
    Parent = parent;
    CaliCommand = "WTTComboEnter ";
}

void ComboEnter::WriteCaliCommand()
{
    HitNr++;
    // HitNr counts how often the stimulus was called.
    fprintf( script, "%s%i%s", CaliCommand, Parent->ldNr, "\n" );
}
CallCommand = "WTTComboEnter ";  
Text = text;  
}  

void ComboEnter::WriteCallCommand()  
{  
    HitNr++;  // HitNr counts how often the stimulus was called.  
    fprintf( script, "%s%i%s%s", CallCommand, Parent->Parent->ldNr, ", " , Text, "\n" );  
}  

int ComboEnter::Enabled()  
{  
    return( Parent->Parent->EnabledFlag );  
}  

// Methods belonging to class ComboSelect  
ComboSelect::ComboSelect(ComboItem* parent)  
:Stimulus(parent)  
{  
    HitNr = 0;  
    Parent = parent;  
    CallCommand = "WTTComboSelect ";  
}  

void ComboSelect::WriteCallCommand()  
{  
    HitNr++;  // HitNr counts how often the stimulus was called.  
    fprintf( script, "%s%i%s%i%s", CallCommand, Parent->Parent->ldNr, ", ", Parent->ldNr, "\n" );  
}  

int ComboSelect::Enabled()  
{  
    return( Parent->Parent->EnabledFlag);  
}  

// Methods belonging to the class EditEnter  
EditEnter::EditEnter(EditBox* parent, char* text)  
:Stimulus(parent)  
{  
    HitNr = 0;  
    Parent = parent;  
    CallCommand = "WTTEditEnter ";  
    Text = text;  
}  

void EditEnter::WriteCallCommand()  
{  
    HitNr++;  // HitNr counts how often the stimulus was called.  
    fprintf( script, "%s%i%s.%, Text, "\n" );  
}  

int EditEnter::Enabled()  
{  
    return( Parent->EnabledFlag );  
}  

// Methods belonging to class MenuItemSelect
MenultemSelect::MenultemSelect(Menultem* parent) :
  Stimulus(parent)
{
  HitNr = 0;
  Parent = parent;
  CallCommand = "WTMenultemSelect ";
}

void MenultemSelect::WriteCallCommand()
{
  HitNr++;
  // HitNr counts how often the stimulus was called.
  fprintf( script, "%s%04i%s" , CallCommand, Parent->ldNr, "n");
}

int MenultemSelect::Enabled()
{
  return( Parent->EnabledFlag);
}

3) Remaining Files

CLOCK.cpp

#include "witems.h"
#include <ctype.h>

void DefineClock()
{
  TestObject* Clock;
  TestObject* PureClock;
  TestObject* Menu;
  TestObject* Font;
  TestObject* FontBackUp;
  TestObject* Info;

  // TestObject Clock
  Clock = new TestObject("Clock",1);  // Initial Flag: ActiveFlag = 1

  // Stimulus
  new DoKeys(Clock, "\{ESC}\"m");
  // Item
  new Menultem(Clock, 1, "&Settings"); // Initial Default Flags: Enabled, UnChecked

  // TestObject PureClock
  // This Object is in fact the same as clock, but it doesn't have a Menultem
  PureClock = new TestObject("PureClock"); // Initial Flag: ActiveFlag = 0
  // Stimulus
  new DoKeys(PureClock, "\{ESC}\"m");

  // TestObject Menu
  Menu = new TestObject("Menu");
  // Items
  new Menultem(Menu, 1, "&Analog",1,1); // Initial Flags: Enabled, Checked
new Menultem(Menu, 2, "&Digital"); // Initial Default = Flags: Enabled, UnChecked
new Menultem(Menu, 3, "Set &Font...", 0); // Initial Flags: Disabled, Unchecked
new Menultem(Menu, 4, "&No Title"); // Initial Default
new Menultem(Menu, 5, "&Seconds"); // Initial Default
new Menultem(Menu, 6, "Da&te"); // Initial Default
new Menultem(Menu, 7, "A&bout Clock..."); // Initial Default

// TestObject Font
Font = new TestObject("Font");
// Items
new Button(Font, 1, "OK", 1, 0, 1 ); //Initial Flags: Enabled, Focus, Default
new Button(Font, 2, "Cancel", 1 ); //Initial Flags: Enabled, NotFocus, NotDefault
ComboBox* Fonts = new ComboBox(Font, 1, "",1,2);
// Child items
new Comboltem( Fonts, 1, "Algerian", 0);
new Comboltem( Fonts, 2, "Arial", 1); // There are more Comboltems, but for the demonstration these two satisfy
//
new ComboEdit(Fonts, "Arial");

new StaticControl(Font, 1, "&Font");

// Test Object FontBackUp is a copy of the instance Font without the stimuli.
// As the button OK is pushed in font, the changed attributes in font are copied
// to FontBackUp. As the button CANCEL is pushed, the attributes of FontBackUp are
// written into Font.
FontBackUp = new TestObject("FontBackUp");
// Items
new Button(FontBackUp, 1, "OK", 1, 0, 1 ); //Initial Flags: Enabled, Focus, Default
new Button(FontBackUp, 2, "Cancel", 1 ); //Initial Flags: Enabled, NotFocus, NotDefault
ComboBox* FontsBackUp = new ComboBox(FontBackUp, 1, "",1,2);
// Children
new Comboltem(FontsBackUp, 1, "Algerian", 0);
new Comboltem(FontsBackUp, 2, "Arial", 1);
//
new ComboEdit(FontsBackUp, "Arial");

// No Stimuli are needed in the backup object:
FontBackUp->GetItem(1)->StimulusList = NULL;
FontBackUp->GetItem(2)->StimulusList = NULL;
while( child )
{
    child->StimulusList = NULL;
    child = child->next;
}

// TestObject Info
Info = new TestObject("Info");
// Items
new Button(Info, 1, "OK", 1, 1, 1 ); //Initial Flags: Enabled, Focus, Default
// As example just a few static controls are tested. Not all of them are defined.
new StaticControl(Info, 1, "");
new StaticControl(Info, 2, "Clock");
new StaticControl(Info, 3, "Microsoft Windows for Workgroups");
new StaticControl(Info, 4, "Version 3.11");
// ......... etcetera


 boasted MethodCalls are added at the end as all Objects and their methods already exist.


new MethodCall(Clock->GetStimulus(1), Clock, Clock->SetActiveFlag, 0);
new MethodCall(Clock->GetStimulus(1), PureClock, PureClock->SetActiveFlag, 1);
//MenuItemSelect1
new MethodCall(Clock->GetStimulus(2), Clock, Clock->SetActiveFlag, 0);
new MethodCall(Clock->GetStimulus(2), Menu, Menu->SetActiveFlag, 1);


new MethodCall(PureClock->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(PureClock->GetStimulus(1), PureClock, PureClock->SetActiveFlag, 0);


new MethodCall(Menu->GetStimulus(1), Menu->GetItem(1), (MenuItem *)Menu->GetItem(1)->
SetCheckedFlag, 1);
new MethodCall(Menu->GetStimulus(1), Menu->GetItem(2), (MenuItem *)Menu->GetItem(2)->
SetCheckedFlag, 0);
new MethodCall(Menu->GetStimulus(1), Menu->GetItem(3), (MenuItem *)Menu->GetItem(3)->
SetEnabledFlag, 0);
new MethodCall(Menu->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(1), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect2
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(1), (MenuItem *)Menu->GetItem(1)->
SetCheckedFlag, 0);
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(2), (MenuItem *)Menu->GetItem(2)->
SetCheckedFlag, 1);
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(3), (MenuItem *)Menu->GetItem(3)->
SetEnabledFlag, 1);
new MethodCall(Menu->GetStimulus(2), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(2), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect3
new MethodCall(Menu->GetStimulus(3), Font, Font->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(3), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect4
new MethodCall(Menu->GetStimulus(4), Menu, Menu->SetActiveFlag, 0);
new MethodCall(Menu->GetStimulus(4), PureClock, PureClock->SetActiveFlag, 1);
//MenuItemSelect5
new MethodCall(Menu->GetStimulus(5), Menu->GetItem(5), (MenuItem *)Menu->GetItem(5)->
SetCheckedFlag, 2);
new MethodCall(Menu->GetStimulus(5), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(5), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect6
new MethodCall(Menu->GetStimulus(6), Menu->GetItem(6), (MenuItem *)Menu->
GetItem(6)->SetCheckedFlag, 2);
new MethodCall(Menu->GetStimulus(6), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(6), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect7
new MethodCall(Menu->GetStimulus(7), Menu, Menu->SetActiveFlag, 0);
new MethodCall(Menu->GetStimulus(7), Info, Info->SetActiveFlag, 1);


new MethodCall(Clock "[ESC]
new MethodCall(Clock->GetStimulus(1), Clock, Clock->SetActiveFlag, 0);
new MethodCall(Clock->GetStimulus(1), PureClock, PureClock->SetActiveFlag, 1);
//MenuItemSelect1
new MethodCall(Clock->GetStimulus(2), Clock, Clock->SetActiveFlag, 0);
new MethodCall(Clock->GetStimulus(2), Menu, Menu->SetActiveFlag, 1);


new MethodCall(PureClock "[ESC]
new MethodCall(PureClock->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(PureClock->GetStimulus(1), PureClock, PureClock->SetActiveFlag, 0);

new MethodCall(Menu->GetStimulus(1), Menu->GetItem(1), (MenuItem *)Menu->GetItem(1)->
SetCheckedFlag, 1);
new MethodCall(Menu->GetStimulus(1), Menu->GetItem(2), (MenuItem *)Menu->GetItem(2)->
SetCheckedFlag, 0);
new MethodCall(Menu->GetStimulus(1), Menu->GetItem(3), (MenuItem *)Menu->GetItem(3)->
SetEnabledFlag, 0);
new MethodCall(Menu->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(1), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect2
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(1), (MenuItem *)Menu->GetItem(1)->
SetCheckedFlag, 0);
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(2), (MenuItem *)Menu->GetItem(2)->
SetCheckedFlag, 1);
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(3), (MenuItem *)Menu->GetItem(3)->
SetEnabledFlag, 1);
new MethodCall(Menu->GetStimulus(2), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(2), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect3
new MethodCall(Menu->GetStimulus(3), Font, Font->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(3), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect4
new MethodCall(Menu->GetStimulus(4), Menu, Menu->SetActiveFlag, 0);
new MethodCall(Menu->GetStimulus(4), PureClock, PureClock->SetActiveFlag, 1);
//MenuItemSelect5
new MethodCall(Menu->GetStimulus(5), Menu->GetItem(5), (MenuItem *)Menu->GetItem(5)->
SetCheckedFlag, 2);
new MethodCall(Menu->GetStimulus(5), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(5), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect6
new MethodCall(Menu->GetStimulus(6), Menu->GetItem(6), (MenuItem *)Menu->
GetItem(6)->SetCheckedFlag, 2);
new MethodCall(Menu->GetStimulus(6), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(6), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect7
new MethodCall(Menu->GetStimulus(7), Menu, Menu->SetActiveFlag, 0);
new MethodCall(Menu->GetStimulus(7), Info, Info->SetActiveFlag, 1);


new MethodCall(Clock "[ESC]
new MethodCall(Clock->GetStimulus(1), Clock, Clock->SetActiveFlag, 0);
new MethodCall(Clock->GetStimulus(1), PureClock, PureClock->SetActiveFlag, 1);
//MenuItemSelect1
new MethodCall(Clock->GetStimulus(2), Clock, Clock->SetActiveFlag, 0);
new MethodCall(Clock->GetStimulus(2), Menu, Menu->SetActiveFlag, 1);


new MethodCall(PureClock "[ESC]
new MethodCall(PureClock->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(PureClock->GetStimulus(1), PureClock, PureClock->SetActiveFlag, 0);

new MethodCall(Menu->GetStimulus(1), Menu->GetItem(1), (MenuItem *)Menu->GetItem(1)->
SetCheckedFlag, 1);
new MethodCall(Menu->GetStimulus(1), Menu->GetItem(2), (MenuItem *)Menu->GetItem(2)->
SetCheckedFlag, 0);
new MethodCall(Menu->GetStimulus(1), Menu->GetItem(3), (MenuItem *)Menu->GetItem(3)->
SetEnabledFlag, 0);
new MethodCall(Menu->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(1), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect2
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(1), (MenuItem *)Menu->GetItem(1)->
SetCheckedFlag, 0);
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(2), (MenuItem *)Menu->GetItem(2)->
SetCheckedFlag, 1);
new MethodCall(Menu->GetStimulus(2), Menu->GetItem(3), (MenuItem *)Menu->GetItem(3)->
SetEnabledFlag, 1);
new MethodCall(Menu->GetStimulus(2), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(2), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect3
new MethodCall(Menu->GetStimulus(3), Font, Font->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(3), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect4
new MethodCall(Menu->GetStimulus(4), Menu, Menu->SetActiveFlag, 0);
new MethodCall(Menu->GetStimulus(4), PureClock, PureClock->SetActiveFlag, 1);
//MenuItemSelect5
new MethodCall(Menu->GetStimulus(5), Menu->GetItem(5), (MenuItem *)Menu->GetItem(5)->
SetCheckedFlag, 2);
new MethodCall(Menu->GetStimulus(5), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(5), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect6
new MethodCall(Menu->GetStimulus(6), Menu->GetItem(6), (MenuItem *)Menu->
GetItem(6)->SetCheckedFlag, 2);
new MethodCall(Menu->GetStimulus(6), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Menu->GetStimulus(6), Menu, Menu->SetActiveFlag, 0);
//MenuItemSelect7
new MethodCall(Menu->GetStimulus(7), Menu, Menu->SetActiveFlag, 0);
new MethodCall(Menu->GetStimulus(7), Info, Info->SetActiveFlag, 1);
new MethodCall(Font->GetStimulus(1), Font, Font->SetActiveFlag, 0);
new MethodCall(Font->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Font->GetStimulus(1), ((ComboBox *)FontBackUp->GetItem(3))->GetChild(3),
((ComboBox *)FontBackUp->GetItem(3))->GetChild(3)->ChangeText,
((ComboBox *)Font->GetItem(3))->GetChild(3)->SetText);
new MethodCall(Font->GetStimulus(1), FontBackUp->GetItem(3), ((ComboBox *)FontBackUp->
GetItem(3))[$_->SetFocusFlag, ((ComboBox *)Font->GetItem(3))->FocusItemNr];

// ButtonClick2
new MethodCall(Font->GetStimulus(2), Font, Font->SetActiveFlag, 0);
new MethodCall(Font->GetStimulus(2), Clock, Clock->SetActiveFlag, 1);
new MethodCall(Font->GetStimulus(2), ((ComboBox *)Font->GetItem(3))->GetChild(3),
((ComboBox *)Font->GetItem(3))->GetChild(3)->ChangeText,
((ComboBox *)FontBackUp->GetItem(3))->SetText);
new MethodCall(Font->GetStimulus(2), ((ComboBox *)Font->GetItem(3)), ((ComboBox *)Font->
GetItem(3))->SetFocusFlag, ((ComboBox *)FontBackUp->GetItem(3))->FocusItemNr);

// ComboBoxSelect1
new MethodCall(Font->GetStimulus(3), ((ComboBox *)Font->GetItem(3))->GetChild(3),
((ComboBox *)Font->GetItem(3))->GetChild(3)->ChangeText, "Algerian);
new MethodCall(Font->GetStimulus(3), ((ComboBox *)Font->GetItem(3)), ((ComboBox *)Font->
GetItem(3))->SetFocusFlag, (ComboBox *)FontBackUp->GetItem(3))->FocusItemNr;

// ComboBoxSelect2
new MethodCall(Font->GetStimulus(4), ((ComboBox *)Font->GetItem(3))->GetChild(3),
((ComboBox *)Font->GetItem(3))->GetChild(3)->ChangeText, "Arial");
new MethodCall(Font->GetStimulus(4), ((ComboBox *)Font->GetItem(3)), ((ComboBox *)Font->
GetItem(3))->SetFocusFlag, 2);

// Info
// ButtonClick1
new MethodCall(Info->GetStimulus(1), Info, Info->SetActiveFlag, 0);
new MethodCall(Info->GetStimulus(1), Clock, Clock->SetActiveFlag, 1);

DUMMY.cpp

#include "witems.h"
#include <ctype.h>

FILE* stream;
FILE* script;

extern void DefineClock();

// Some initialisations
void Init()
{
// Define an output file "OOTGOUT.dat" and a script file CLOCK.mst
remove("OOTGOUT.dat");
stream = fopen("OOTGOUT.dat", "a");
remove("CLOCK.mst");
script = fopen("CLOCK.mst", "a");
fprintf(script, "$Header use O:\WTT\WTT.hdr\nWTTinitRun\n\nWTRun "Clock.exe"
\n$Header$\n\n";
}

// The function GetActiveObject returns a pointer to the active TestObject
TestObject* GetActiveObject()
{
TestObject* TestObjectPtr = FirstTestObject;
while (TestObjectPtr)
{
if (TestObjectPtr->ActiveFlag )

83
return (TestObjectPtr);
    TestObjectPtr = TestObjectPtr->next;
}
cout << "ERROR: No active TestObject found!"
    return (NULL);

// The function DisplayAll can be used to display all stimuli in all testobjects

void DisplayAll()
{
    TestObject* TOPtr = FirstTestObject;
    while (TOPtr)
    {
        TOPtr->DisplayStimuli();
        TOPtr = TOPtr->next;
    }
}

// The function Statistics determines the stimulus coverage that was reached.
// The coverage is written to the standard output

void Statistics()
{
    TestObject* TOPtr = FirstTestObject;
    int StimuliNr = 0;
    int HitNr = 0;
    int Coverage;
    while (TOPtr)
    {
        StimuliNr += TOPtr->StimuliCount();
        HitNr += TOPtr->StimuliHitCount();
        TOPtr = TOPtr->next;
    }
    Coverage = ((int)((float(HitNr)/StimuliNr) * 100));
    cout << "The total number of stimuli is: " << StimuliNr << "\n";
    cout << "The number of used stimuli is : " << HitNr << "\n";
    cout << "StimulusCoverage: " << Coverage << "%\n\n";
}

void Run(int StepNr = 250, int RandomSet = 1)
{
    cout << "Creating WTT test script for the program Clock....\n"
    fprintf(script, "%s%i%s%i%s", "This script was created by OOTG.\n"The Number of steps is: ",
                   "StepNr, "\n\n"The Random Generator was initialised on: ", RandomSet, "\n\n"");
    int i;
    int RandomNr;
    int StimNr;
    int StimEnabledCount;
    TestObject* ActiveObject;
    Stimulus* SelectedStim;
    srand(RandomSet);
    for (i = 0; i != StepNr; ++i)
    {
        ActiveObject = GetActiveObject();
        fprintf(script, "%s%i%s", "Active Object: ", ActiveObject->Name, "\n");
        fprintf(script, "%s%i%s", "WTTTestStep ", i + 100, "\n"");
        ActiveObject->SelfTest();
StimEnabledCount = ActiveObject->StimuliEnabledCount();

RandomNr = rand();
StimNr = RandomNr % StimEnabledCount + 1;
SelectedStim = ActiveObject->GetStimulusEnabled(StimNr);

SelectedStim->Call();
fprintf(script, "\n");
}
cout << "The script that was written is named 'Clock.mst'.\n";
cout << "The Number of steps was: " << StepNr;
cout << "The Random Generator was initialised on: " << RandomSet << "\n\n";
}

// The functions GetSteps and GetRandomSeed are used to get user input

int GetSteps()
{
    int steps;
    printf("Input the number of test steps: ");
    scanf("%i", &steps);
    return(steps);
}

int GetRandomSeed()
{
    int seed;
    printf("Input an integer as seed for the random generator: ");
    scanf("%i", &seed);
    return(seed);
}

void main(int argc, char *argv[])
{
    Init();
    DefineClock();

    Run(GetSteps(), GetRandomSeed());
    Statistics();

    fclose(stream);
    fclose(script);
}