Simulating the ENIAC as a Java Applet

Diploma Thesis

by Till Zoppke

at Free University of Berlin
Department of Mathematics and Computer Science
Takustr. 9, D-14195 Berlin, Germany

Advisors:
Prof. Dr. Raúl Rojas
Prof. Dr. Peter Löhr

June 2004

Abstract

The aim of this paper is to introduce the ENIAC simulation. The ENIAC simulation is a Java applet developed in the context of the author's diploma thesis. It revives the historic ENIAC, the world's first electronic digital computer and ancestor of all contemporary modern computers. The ENIAC simulation can be used to introduce the learner to the functionality of the ENIAC, but also as advanced developing environment for designing and testing ENIAC programs. The applet is located at `http://kinnla.de/eniac/`.

Contents

1 The historic ENIAC **6**
1.1 Trajectory calculation 6
1.2 Speed up through vacuum tubes 7
1.3 Programming the ENIAC 8

2 A first tour **10**
2.1 Starting the ENIAC simulation 10
2.2 The Initiating unit 11
2.3 The Cycling unit 12
2.4 The Accumulator unit 14
2.5 Running the simple example 16
2.5.1 Press "go!" 17
2.5.2 Triggering operations 17
2.5.3 Number transmission 17
2.5.4 Carry over 18
2.5.5 One transmission cycle 18
2.5.6 Operation completed 19
2.6 How to learn more? 19

3 Fibonacci number computation **20**
3.1 The basic setup 20
3.2 Interconnection of Accumulators 21
3.3 The Constant Transmitter 22
3.4 Running the program 23
3.4.1 The clear correct switch 23
3.4.2 Reading a constant 24
3.5 Two main loops 25
3.5.1 The Fibonacci computation loop 25
3.5.2 The for-loop 27

3.5.2.1 converting digits to program pulses 27
3.6 Program termination 28
3.7 Reading the result 28

4 The ENIAC simulation Architecture 29
4.1 Namespaces 29
4.1.1 Singleton classes 29
4.1.2 Class `eniac.util.Status` 30
4.1.3 Class `eniac.io.Tags` 31
4.1.4 Class `java.lang.Words` 31
4.1.5 The classloader namespace 32
4.1.6 Parameters 33
4.1.7 Model layer types 34
4.1.7.1 XML input and output 34
4.1.7.2 Creating graphical objects from model layer objects 34
4.1.7.3 Identifying nodes 34
4.2 ENIAC components 35
4.2.1 The model layer 35
4.2.2 The kindergarten 36
4.2.3 The graphical layer 36
4.2.4 The descriptor 37
4.2.5 The control layer 38
4.3 XML files 38
4.3.1 Parsing XML using the `org.xml.sax` API 38
4.3.2 The proxy concept 40
4.3.3 XML file types 40
4.3.3.1 eniac.xml 41
4.3.3.2 skin.xml 41
4.3.3.3 lang.xml 42
4.3.3.4 menu.xml 42
4.3.3.5 types.xml 42
4.4 Actions 43
4.4.1 EAction and ToggleAction 43
4.4.2 MenuManager and MenuHandler 44

5 The software development process 45
5.1 General aspects 45
5.1.1 Inner and outer software quality 45
5.1.2 Top-down versus Bottom-up 45

5.1.3 Maintaining a Prototype 46
5.1.4 The project manager and the customer 47
5.2 Developing the ENIAC simulation 48
5.2.1 A time-line for the ENIAC simulation 49
5.2.2 The to do list 50
5.2.3 The diary 51
5.2.4 The factual time-line for the ENIAC simulation 52

6 Summary and Outlook 53

Introduction

In 1995, the ENIAC's 50th birthday has been celebrated. For this reason a team around Jan v. d. Spiegel at University of Pennsylvania reconstructed the ENIAC on a silicium chip.[3] Extensive research on the ENIAC's functioning have been done, before it could be designed in VLSI. The original ENIAC was arranged along the walls of a 10x17 m room and weighted 30 tons. After 50 years the reconstruction took place on a 7.4 x 5.3 square mm chip. This is an effective demonstration of the power of Moore's law. In 2005, the ENIAC's 60th birthday will be celebrated. For this reason a team around Jan v. d. Spiegel and Raúl Rojas is rebuilding the ENIAC in the ratio of 1:4 by using historical technology, in particular vacuum tubes.

Together with the hardware reconstruction, it would be nice to have a software simulation. Programs for the ENIAC could be tested within the simulation before they are wired at the hardware. On the other hand, the double tracking can be used to debug the reconstruction and the simulation vice versa: Wire any program at the simulation and the same program at the reconstruction, then compare the results. The probability that simulation and reconstruction both have the same buggy result is supposed to be very small. So if the results are the same, their computation is probably OK, and if they are different, at least one of both has a bug. The ENIAC simulation has its origin in this idea.

After a short introduction to the historical ENIAC in chapter 1, a tutorial containing two examples of ENIAC programs will follow in chaps. 2 and 3. Switching to the programmer's perspective, I will introduce the software architecture in chap. 4, and report about the development process in chap. 5. Finally in chap. 6 the ENIAC simulation is compared with a prior work.[8] Also suggestions of new features are given.

Chapter 1

The historic ENIAC

1.1 Trajectory calculation

At the time of World War II intelligent bombs were not yet developed, so ground based artillery was used to attack the enemy. Depending on the distance of the target and the type of artillery, the bullet has to be shoot in a certain angle. This angle also is related to the weather, especially to the wind. To know the correct angle in the specific situation, the artillery men used so-called firing tables. But those firing tables had to be computed first.

Those ballistic computations were done at the Moore School of Electrical Engineering, part of the University of Pennsylvania, too.

> Calculating a trajectory could take up to 40 hours using a desktop calculator. The same problem took 30 minutes or so on the Moore School's differential analyzer. But the School had only one such machine, and since each firing table involved hundreds of trajectories it might still take the better part of a month to complete just one table.[5]

The speed up in developing new artillery designs caused an increased need of computation power.[1] Under these circumstances John Mauchly, a member of Moore School's *Engineering, Science, and Management War Training* (ESMWT) program, wrote a first five-page memo called *The Use of Vacuum Tube Devices in Calculating.* In this paper he suggested a machine that

[1] In November 1942 US forces landed in French North Africa, and entered a terrain, which was entirely different from what they had met before. The existing firing tables turned out as completely useless. That made the computation power totally to the bottleneck of the war machinery.

Figure 1.1: View of the ENIAC in its U-shaped room (from [15])

would add 5,000 10-digit numbers per second and would be more than 100 times faster than the fastest computer at that time.[2]

1.2 Speed up through vacuum tubes

The memo was followed by two proposals, which Mauchly wrote together with J. Presper Eckert, Jr., an instructor at Moore School. In June, 1942, a contract was signed and the project became funded by the United States Army. The machine to be developed was named *Electronic Numerical Integrator And Computer*, short *ENIAC*.[3]

The ENIAC's great innovation was the usage of vacuum tubes for number representation in contrast to mechanical relays, which were commonly used at that time. The machine consists out of forty independent panels, each 0.6 m wide, 2.7 m high and 0.7 m deep plus three movable function tables. Those units were *arranged in U shape occupying an area of about 10m by 17m* (see [3], p.124). The total number of 17,468 vacuum tubes were used.

[2]The fastest computer in 1942 was a mechanical relay computer operating at Harvard, Bell Laboratories with 15-50 additions per second.[5]

[3]On the web you also can read *Calculator* instead of *Computer*. I use the name according to the report from 1945.[1]

The first demonstration of the ENIAC's computing power could be given two years later:

> In May of 1944, the ENIAC team was able to demonstrate ENIAC's workability in what has come to be known as the two accumulator test. In this, one accumulator was made to increment its value from one to five. The number was then transferred into the second unit one thousand times using the limited control circuitry housed in each accumulator, all in just over one fifth of a second, or about the blink of the eye. At the end of the test, the second accumulator sat idle, displaying the number 5,000 – hardly the most impressive of mathematical feats. [5]

One problem to deal with was the reliability of the vacuum tubes. For frequently used circuits the tubes were selected by hands, and special test procedures were implemented to identify a broken tube within minutes in case of a failure. This happened about two or three times a week causing a weekly downtime of only a few hours. (cf. [3], p.126)

1.3 Programming the ENIAC

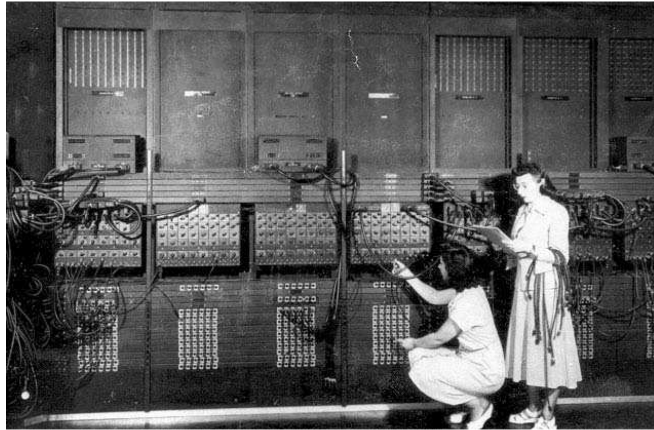
The ENIAC didn't have a memory to store programs. So programming the ENIAC means physically connecting the units by cables and turning switches to the appropriate settings. This work was done by six women whose jobs were called *computers*. They were chosen from a group of over eighty women that were calculating ballistic trajectories by hand. (cf. [6])

20 of the ENIAC's 40 units were *Accumulators*. An Accumulator was capable to store a 10-digit number, to add and to subtract. Other arithmetic units were the *Multiplier*, supported by the *Function Table*, and the *Divider/Square-rooter*. For fast number input, the *Constant Transmitter* was used, whereas for controlling the program flow a *Master Programmer* was helpful. The IO connection to stored data was done by an IBM card reader and an IBM card punch controlled by a printer panel. Finally there were the *Cycling unit* as pulse machine and the *Initiating unit* to boot the whole machine or to start a program.[4]

In May 1945, yet before the ENIAC was formally completed, the World War II came to its end. That meant, the initial aim of computing firing tables

[4]It doesn't fit into the context of this paper, to give a detailed description of the ENIAC's functionality. If you want to learn more about that, refer to J. v. d. Spiegel [3].

Figure 1.2: The first professional programmers (from [16])



for the artillery became obsolete. But the upcoming cold war required even more complex computations. So *the first real task assigned to ENIAC during its test runs in 1945 involved millions of discrete calculations associated with top-secret studies of thermonuclear chain reactions – the hydrogen bomb.* [5]

Chapter 2

A first tour

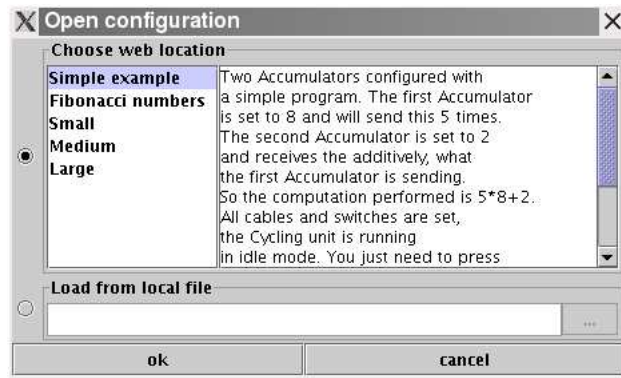
In this chapter the features of the ENIAC simulation are explained. Even people with little computing background will find a way to use the ENIAC simulation and to understand the basics of how the ENIAC works.

2.1 Starting the ENIAC simulation

The ENIAC simulation needs less than 5 seconds to start on a GHz PC plus the time you need to download the program.

At first you are prompted to accept the certificate of the ENIAC simulation. If you refuse to do so, you won't be able to save ENIAC configuration files to your hard-disk. But anyway, this might not be your plan, even if you start the ENIAC simulation the first time. So you probably won't miss that feature. If you change your mind later, at any time you can restart your browser and accept the certificate.

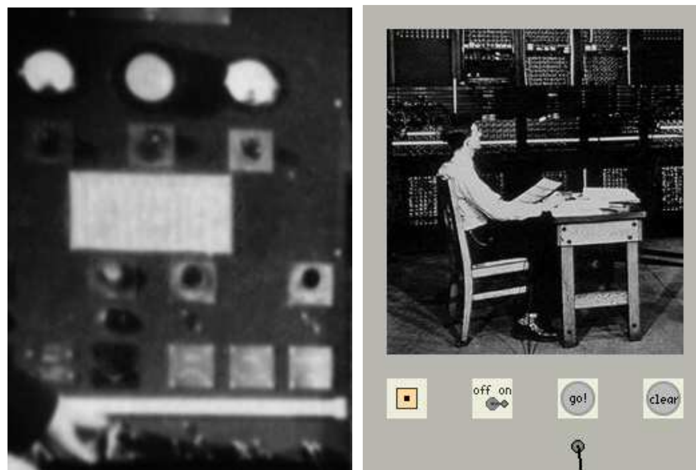
When the Java class files are loaded, the applet starts and announces that it is busy. External resources as images and data files must be loaded. You stay informed about the progress.



At next you are prompted to select an ENIAC configuration from a list of currently 5 alternatives. An ENIAC configuration includes more or less of the units that originally could be found at the ENIAC. Also some configurations have a program wired. The features of the configurations for your choice are printed in the right hand text area when you select a configuration. When you are using the ENIAC simulation the first time, choose the "Simple Example" that is already preselected and confirm your selection by pressing OK.

Now the main window opens. As you probably know it from other software, you find a menu bar and a tool bar to control the program. Inside the main window you can see the units of the ENIAC.

2.2 The Initiating unit



The first unit is called "Initiating unit". Above, you see the original front panel at the left hand, and the simulated one on the right hand. Don't be irritated by the fact that a photo of the original ENIAC is displayed at the simulated one.

The main purpose of the Initiating unit was to perform the boot process of the ENIAC. When a unit was switched to power, its tubes were in an inconsistent state and had to be initially cleared. This complex process is not simulated, so the simulated Initiating unit has less instruments than the original one.

Like all units of the ENIAC simulation, the Initiating unit has a colored icon. If you later load a bigger configuration you will use the overview window to navigate. In this window you find any unit represented by its icon. The icon cannot be clicked.

The second square is the power switch, the so called "heaters". Every unit has such a power switch. In the simple example, all units are initially switched on.

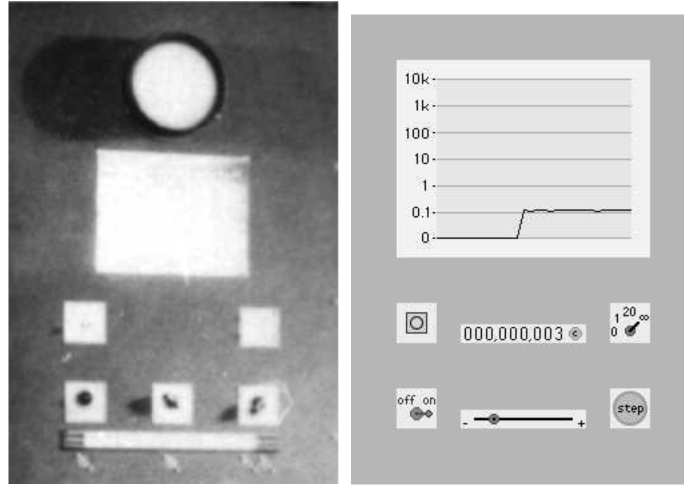
Next is the "go!" button. Its original name is unknown, but its meaning is suitable described by "go!". When this button was pressed, an initial program pulse is sent to the other units. Or rather: The next time when the pulse cycle of the ENIAC reaches the CPP[1] phase it will be sent. One addition cycle of the original ENIAC was 200 milliseconds, the default setting in the ENIAC simulation is about 5 seconds.

The last button of the Initiating unit is the clear button. It is used to send a clearing pulse to the other units.

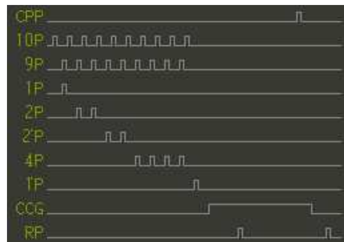
2.3 The Cycling unit

The task of the Cycling unit is to provide the other units with clock pulses. It is the heart of the ENIAC.

[1]Central Programming Pulse



The original Cycling unit had an oscilloscope (see the circle at the photo), where you could observe a selected pulse and a few controls for using two debug modes. In the first debug mode, the pulses for one addition cycle were sent, in the second one just the pulse for the next 10 milliseconds was sent.



Though you cannot see the pulse in the simulated Cycling unit, it provides more features than the original one. The rectangle underneath the oscilloscope of the original Cycling unit is a pulse table. The same table you can find at the head of the simulated cycling unit, but improved by a vertical line indicating the current state of the pulse cycle. When you loaded the simple example, you should see this line moving to the right, then disappearing at the end and restarting from the left. That means, the simulated ENIAC is already running in idle mode.

Below the animated pulse table, you find a slightly smaller rectangle. Over there the time-line of the simulation speed is displayed. On a state of the art GHz computer, you should have a simulation high-speed of about 2000 addition cycles per second. Of course the high-speed also depends on

the complexity of the program executed on the ENIAC simulation. The original speed of the ENIAC was 5000 addition cycles per second. So according to Moore's law the simulated ENIAC will be as fast as the original one within about 3 years.

Below the simulation speed screen, the number of performed addition cycles is counted. On the right you find the iteration switch, a rotary switch with a range from zero to infinity. There you can select one of three debug modes or the continuous operation mode. Below the iteration switch is a button labeled as "step". If you are in the debug mode, you can step through the cycle by pushing this button. The step width depends on the iteration switch setting.

To the left of the step button, you find the frequency slider, with that you can set the speed of the simulated ENIAC clock. This tool is also not know at the original ENIAC, because the clock was running at a fixed speed. Finally there are a power switch and an icon.

2.4 The Accumulator unit

The ENIAC is called an Accumulator based computer. The aim of the accumulator is to store a ten-digit signed number and to receive or to send it positively or negatively.

The Accumulator cannot be explained without a general introduction into the concept how the distributed units of the ENIAC work together. The operations of the Accumulator are triggered by program pulses. Program pulses usually are send through the lower trays.

The screen-shot above shows 5 trays. Every tray contains 11 parallel wires. Each wire can be accessed by several connectors. In our case, we have 3 connectors per wire (only 2 are included in the screen-shot). A program connector is represented by a small circle with a dot in its middle. There are 3 series of those circles. The first connector of every series leads to the first wire, the second connector to the second wire, and so on.

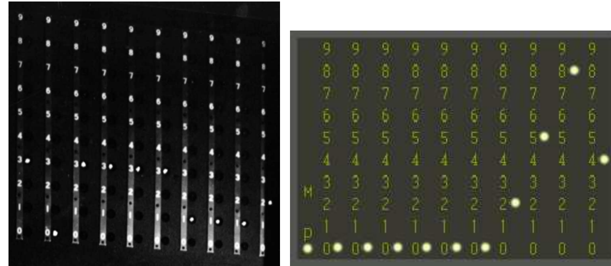
A connector can be plugged by a cable. In the simple example, there are 3 cables between program connectors. So the pulse can go from a unit to a tray and from the tray to another unit. Both Accumulators have a program connector plugged to the same program tray, both to the 9th wire.

An accumulator has 4 single program connectors and 8 pairs. The left one of a pair is an input and the right one is an output. The 4 single connectors are all inputs. If a program pulse reaches an input connector, an operation will start. The type of operation is given by the corresponding operation switch. In the case of a paired program connector, the corresponding operation switch is the one two rows above, in case of a single program connector it is even one row higher. The Operation switch lets you select values from alpha to ε, 0, A, AS and S.

Now the digit pulses join the game. The Accumulator has 7 digit connectors close to its upper boarder. A digit connector is represented by a rectangle with 15 dots. The digit connectors from left to right are named α .. ε, A, S (please note that these names are displayed as tool-tips). The first 5 are input connectors, the last 2 are outputs.

Depending on the Operation switch setting, a number is received by one of the inputs, no operation is performed or a digit is sent through one or both outputs. Receiving a number means adding it to the currently stored number. Sending a number through the A output means sending it as it is, sending it through the S output means sending its negated value.

An operation can be repeated several times, if the Repeat switch is set accordingly. The corresponding Repeat switch to a paired program connector is the one just above. After executing the operation the last time, a program pulse is sent through the output Program connector.



On top of the Accumulator you can find the blinkenlights.[2] During program execution their flashing indicates that the Accumulator is performing a computation. When the program is finished you can see the result. The original blinkenlights above show the number 3,033,331,112, while the screenshot shows 2,584. In the simulation you can use the blinkenlights as well for number-input – just click them.

There are a few details of the Accumulator that haven't been explained yet. But for our simple example they can be ignored.

2.5 Running the simple example

If you just skipped the last sections, because you wanted to know what happens after pressing the "go!" Button, please calm down. It is nothing spectacular, just a simple computation. Anyway, if you don't get it, why the computation is performed like it is, you can repeat the whole procedure. And it might be a good idea to set the Iteration switch to the debug mode and watch the computation step by step. OK, here we go!

[2] The term *blinkenlights* is a German-English crossover and origins from a sign in Stanford University, 1959. [14] But in fact the design was introduced by the ENIAC: *In planning their public demonstration in 1946, it occurred to Pres Eckert and the rest of the ENIAC team to place translucent spheres– ping-pong balls cut in half–over the neon bulbs that displayed the values of each of ENIAC's twenty accumulators. Ever since, the flashing lights of computers, often called electronic or giant "Brains" in the early years, have been part of the scene involving computers and science fiction. [5]*

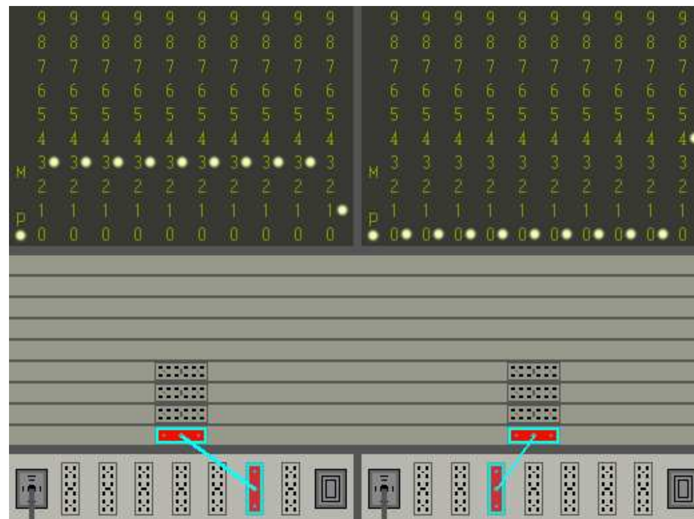
2.5.1 Press "go!"

As mentioned above, we have to wait until the cycle reaches the CPP state. When this is the case, a pulse will go into the program tray. Time-line: less than 170 ms.

2.5.2 Triggering operations

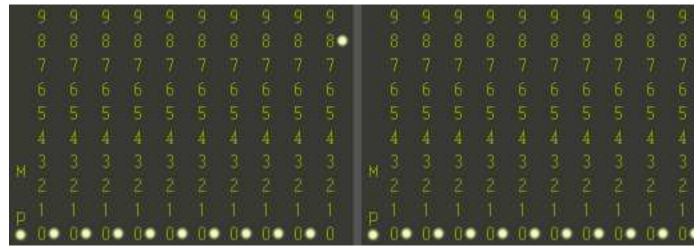
So the operations of the Accumulators are triggered synchronously. The first one is configured to send its number 5 times through the A output. The second Accumulator will listen for 5 addition cycles at its γ input. By the way – the first Accumulator initially stores the number 8, the second one stores number 2. Time-line: 170 ms.

2.5.3 Number transmission



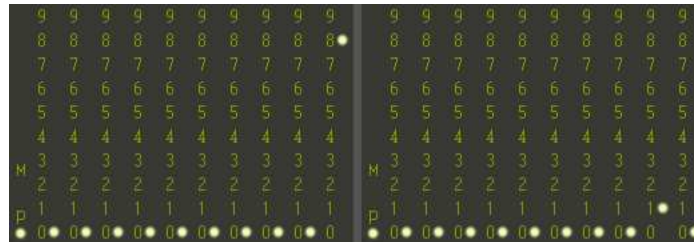
The first accumulator transmits its number by rotating all of its ten decades. The aim is not the flashing effect, it is conditioned by the electronic circuits. The second accumulator receives the digit pulses causing its number to increment. Time-line: 230 ms.

2.5.4 Carry over



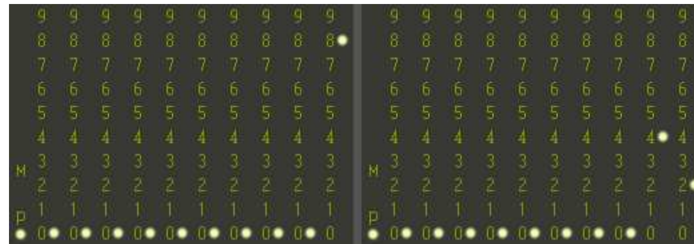
After finishing one rotation cycle, the first Accumulator came back to its initial value 8. But the second one displays 0, though it should display 2+8. The reason for this is the fact, that a carryover cannot be performed immediately. Just an internal flag is set, waiting for the reset pulse to come. This pulse finally triggers the carryover. Time-line: 320 ms.

2.5.5 One transmission cycle

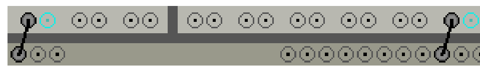


After completing one transmission cycle, the carryover is performed as what we expected. In the next 4 cycles the computation will go on the analogous way. Time-line: 400 ms.

2.5.6 Operation completed



After the 5th number transmission and the carryover performance, the computation is finished and the result is shown on the right Accumulator.



At the CPP cycle-time, both Accumulators send a program pulse in order to confirm that their job is finished. Time-line: 1170.

2.6 How to learn more?

Hopefully you enjoyed our little tour through the ENIAC simulation. If you want to go on learning about the ENIAC, you should start to play around with the simple example. To change numbers you simply can click the blinkenlights of the Accumulators. What will happen, if you change the output from A to S? What will happen, if you transmit a negative number? Try to remove all cables and reset all switches, then try to reconstruct the simple example from your memory. Create your own programs. Can You find one, that runs in an eternal cycle?

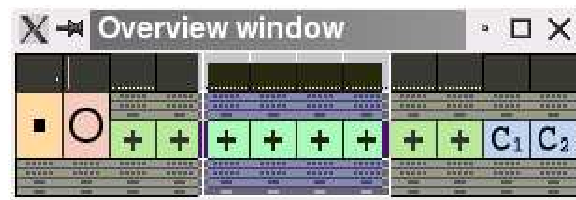
When you feel familiar with those cables and switches, load the advanced example of computing the Fibonacci numbers.

Chapter 3

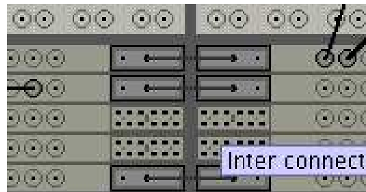
Fibonacci number computation

3.1 The basic setup

After loading the Fibonacci number example, we find the ENIAC running in idle mode like in the previous example. We can know this by the fact, that in the overview window a vertical line is moving above the Cycling units. This configuration is bigger than the previous one. It contains 12 units. We can use the Overview window to scroll around the units by just clicking with the mouse on it.

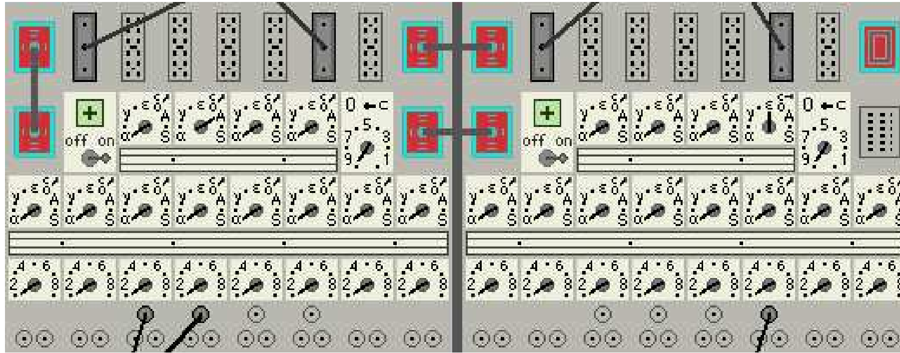


To the right we see the four units we already know from the previous example. Please note, that the trunks containing the wires for pulse transmission, don't lead across the whole length of the configuration. They are as width as 4 units. The upper trunks to the left are just as width as 2 units. So if a pulse should be passed from one trunk to its neighbor, they must be interconnected by a cable.



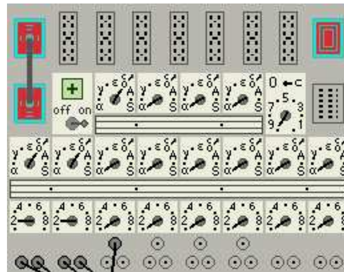
Because of its length, the trunks divide the units into groups of 4 members. Now we scroll to the second group, to the 4 Accumulators, which we saw immediately after the configuration was loaded. We can notice that their wiring is different from that of the 2 accumulators in the first group.

3.2 Interconnection of Accumulators



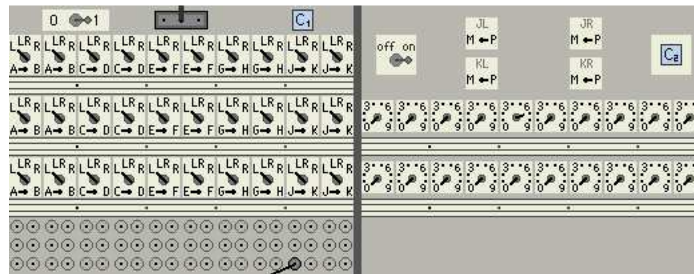
The snapshot is taken at that moment, when the first reset pulse is send. We see 7 connectors (or rather: plugs) highlighted. These connectors are called the *Interconnectors* of the Accumulator units.

An Accumulator can be configured in two ways. In this case, two Accumulators are switched together as a pair. Each partner keeps its own digit inputs and outputs, but shares the program input and output connectors. Whenever a program pulse reaches one partner, the same program is started for both, and they will act synchronously. Because of this feature, the ENIAC is capable to deal with 20-digit numbers. As the sign of the combined number, the sign of the left hand Accumulator is taken and the one of the right hand one is neglected. So when a carryover occurs at the highest digit of the right partner, its sign won't be toggled. The pulse is immediately passed to the lowest decade of the left partner. More than two accumulators cannot be interconnected.

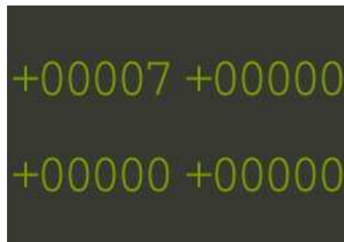


This is how an Accumulator must be wired to act as a single player. Without these plugs, the Accumulator doesn't work (in our example this is the case for Accumulator No. 2, which is out of service). The upper right connector seems to be plugged to itself – it has sticked an item called *load-box*. This load-box contains resistors, and closes a few circuits at the Accumulator, as the cable on the left side, too. The same circuits are closed in a different way, when two Accumulators are interconnected as above.

3.3 The Constant Transmitter



Back to our example. The last group of units contains another two (single) Accumulators and two units to be introduced now. Correctly speaking, these are two panels of a single unit, the Constant Transmitter. The left hand panel has an irritating high number of switches. I suggest you don't touch it this time. The only thing you have to know so far, is that an incoming program pulse triggers this panel to read a constant from its right hand neighbor. The switch according to the active program connector, in our case points to the upper left 5 digits, that are set to 7 in our example.



Please turn the switches and change the number to your favorite one. When the computation starts, this number will be read and taken as the index of the Fibonacci number to be computed. But note, that $fib_{97} =$

83621143489848422977 is the highest number to be displayed.[1] Note also, that the computation of fib_n will need $2 \cdot n + 1$ addition cycles.

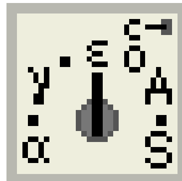
3.4 Running the program

Now let's get ready to start the program. Before pressing the go-button, I suggest, you bring the pulse line at the start position to the left. To do this, you turn the iteration switch to position 0, then you repeatedly press the step button until the pulse line reaches the pole position. Because this program is too complex to be understood at the first view, you better turn the iteration switch to 20 and watch the program addition cycle by addition cycle.

Now press the go-button and the step button. When the pulse line comes to the CPP phase, the program pulse will be sent to the program tray on channel 9. Because the three program trays are interconnected, the pulse also reaches the second and the third group of units. To channel 9 there are four units listening.

3.4.1 The clear correct switch

Accumulator No. 4 is triggered to receive digits at digit input connector ε. Because this Accumulator is coupled with its left neighbor, both share this operation. But neither No. 3 nor No. 4 has any cable plugged to digit-connector ε. So what is the purpose of this command?



[1] This is the largest 20-digit Fibonacci number. If you take the negative sign as a positive digit 1, also fib_{98} can be read computed.

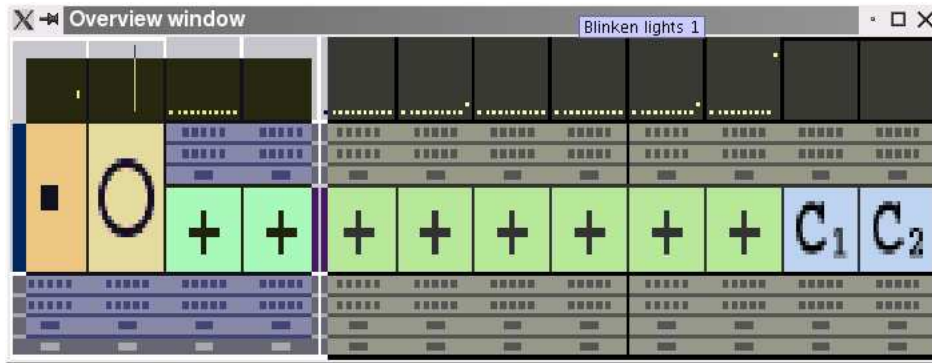
In the default size you hardly can see it, so have a look at the zoomed image above. The *clear correct switch*, a small pointer in the upper right corner of the operation switch, points to c instead of 0. In combination with a receiving operation it means, that the 1'P-pulse will be picked up by the lowest circuit. This is an easy way to increment the value of an Accumulator. So our pair of Accumulators will be set to 1 during the next addition cycle. The same operation is triggered at Accumulator 7, too.

3.4.2 Reading a constant

Accumulator No. 8 is triggered to receive a number through its α digit connector. The Constant Transmitter Panel 1 is triggered to read a number from Panel 2, and send it to its digit output connector. Because both connectors are plugged to digit tray 1, this will effect that the constant n will transferred to Accumulator No. 8.

All events of the following addition cycle are predicted now. You can go back to the Initiating unit and press the step button. Please note, that you probably cannot observe everything live, but you can check the results afterwards. A good tool to view the current values of the Accumulators is the overview window. All blinkenlights are represented as small dots over there. You can adjust the size of the overview panel to your personal preference.

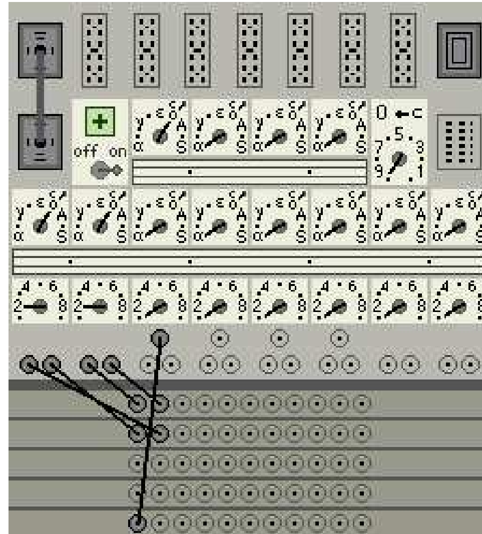
3.5 Two main loops



After the first cycle, the program initialization is finished, and two loops are started. The first loop has a length of 4 addition cycles and includes 5 Accumulators. In this loop, the Fibonacci number is computed. The second loop lasts 2 addition cycles and includes 2 Accumulators. This loop counts n down to 0 and will then cause the stopping procedure. We will examine both loops independently.

3.5.1 The Fibonacci computation loop

When the constant number was received at Accumulator No. 8, a program pulse is send to the channels 1 and 10 of program trays 1 and 2. This triggers the right Accumulator pair to send its number and the left Accumulator pair to receive. When this operation is performed the first time, a 0 is transmitted, so you won't see any pulse highlighting the cables. These operations will be executed once, because they are related to single input connectors.

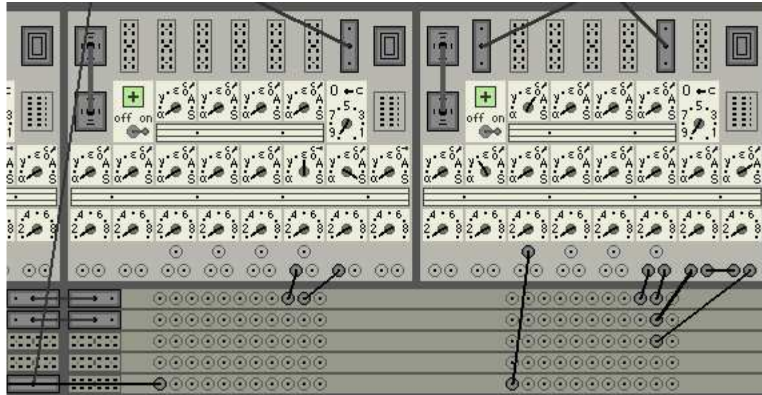


Synchronously to that, the pulse on channel 1 triggers Accumulator No. 1 to do nothing. The nothing will be repeated for two addition cycles, then a program pulse will be sent to channel 2. Accumulator No. 1 itself listens to this channel, and will start another no-operation for two addition cycles. When this operation finished, a program pulse will be sent to channel 1, and the loop starts again.

Effectively, Accumulator No. 1 plays the role of a timer. Every two cycles alternately a pulse to channel 1 and to channel 2 is emitted. Without interruption, this eternal loop will run forever.

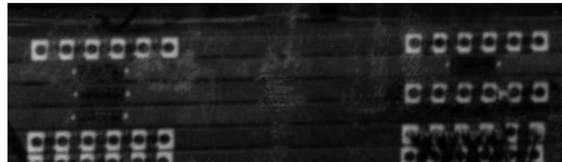
The two pairs of Accumulators, which compute the Fibonacci numbers, are controlled by this timer. If there is a pulse on channel 1, the right pair is sending while the left one is receiving. If there is a pulse on channel 2, the transmission is performed in the opposite direction. In mathematical terms: having the start values $a = 1$,$b = 0$, the iterated execution of the operations $a := a + b$, $b := b + a$ the Fibonacci numbers lets arise.

3.5.2 The for-loop



The other loop also alternates between two states. Over here the role of the timer is played by Accumulator No. 8. During the first cycle of the loop, Accumulator No. 7 sends the negation of its digits, so -1. Accumulator No. 8 receives, and its value is decremented. At the second cycle, No. 7 does nothing, while No. 8 is sending its number to digit tray 2. From there the digit is passed to the program tray H.

3.5.2.1 converting digits to program pulses



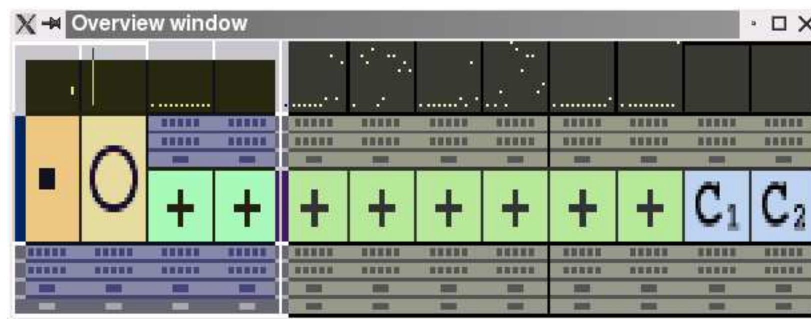
The transition from the digit-sphere to the program-sphere is possible, because all pulses occurring at the ENIAC have the same length and voltage. As well, program and digit trays internally are completely the same components, but with different connectors. Conversions of digit pulses to program pulses were a common praxis in programming the ENIAC. It was even supported by the layout of specialized trays, as you can see on the original photography above: The third tray from the top has a digit connector on its left hand, and 11 program connectors (6 whith a white background, between them another 5 with a dark backround) on its right hand.

3.6 Program termination

Back to our program. Only channel 1 of tray H is read. Pulses in other channels won't cause anything, because no one listens. Channel 1 of a program tray corresponds to the channel of a digit tray, which transmits the sign. For our example this means, there will be a pulse in channel 1 of tray H, when a negative number is transmitted by Accumulator No. 8. This will be the case when the loop is executed the $(n+1)$-th time.

Both timers, Accumulator No. 1 and No. 8, listen to the specified channel. If they receive a pulse from there, they will stop their current operation and start a no-operation. This is how the program is terminated. Please note, that the stopping pulse will occur at the time of the first 9P pulse. So this pulse is asynchronous with regular program pulses, that occur at CPP time. This asserts, that there won't be two program pulses reaching an Accumulator at the same time. That would cause an unpredictable behavior, in the original ENIAC as well as in the ENIAC simulation.

3.7 Reading the result



Because n is tested to be negative, the for-loop will be executed $n+1$ times. For that reason, the correct Fibonacci number is the lower one of the two numbers computed by the Accumulator pairs. The screen-shot above shows the result for $n = 64$ on the overview window. The lower number is 10610209857723.

Chapter 4

The ENIAC simulation Architecture

A complex software as the ENIAC simulation is composed of several modules sharing the responsibility for the whole system. The architecture of this system follows so-called *design patterns*. The main design concepts of the ENIAC simulation are described from the programmers point of view.

4.1 Namespaces

The interaction between the different parts of a software is supported by namespaces. A *namespace*[1] provides its members with a unique name, so that they can be identified.

4.1.1 Singleton classes

Java classes are organized in their own namespace, the package structure. This structure can be used to address objects as well, especially if the class has only one instance. Such a class is called a *singleton*. A singleton usually has a private constructor, a private static instance and a public static access method returning this instance. The use of singleton classes can make software self-instantiating – whenever the access method is called, it checks

[1] In general, a namespace is an abstract zone which is or could be populated by names, or technical terms, or words. A namespace uniquely identifies a set of names so that there is no ambiguity when objects having different origins but the same names are mixed together. In a namespace, each name must be unique. The namespace is the context, and in the namespace each word can uniquely represent (map to) a real-world concept.[13]

whether the static instance is set and initializes it on demand. So the typical java code implementing the singleton pattern looks like this:

```
public static SingletonClass getInstance() {
    if (instance == null) {
        instance = new SingletonClass();
        instance.init();
    }
    return instance;
}
```

In the ENIAC simulation, all singleton classes implement the interface `eniac.MainListener`. This works around a problem caused by the fact that the ENIAC simulation is an applet.[2] Interface `MainListener` defines a method to dispose the singleton instances. All implementing classes should register themselves at the Main class, so that they can be disposed when the applet's life cycle comes to its end. Typical singletons in the ENIAC simulation are window components like `eniac.window.EFrame` and `eniac.window.LogWindow`. Another singleton to be mentioned is discussed in the following subsection.

4.1.2 Class `eniac.util.Status`

The Status class provides static access to a bunch of properties. To each property a `java.beans.PropertyChangeListener` can be registered in order to be informed when the property changes. All properties are initialized with hardcoded default values (e.g. 0, null) at the first call of Status.getInstance(). For the property names, the status class provides its own namespace consisting of public static final strings. By convention, these strings are just

[2]During the execution of a Java-program its classes are loaded into the Java Virtual Machine. This is the same for an application as for an applet. When you exit an application, the virtual machine terminates. When you exit an applet, the behaviour depends on the browser. But it is quite common, that the virtual machine stays alive until you close the browser window. Also the classes stay loaded.

This behaviour effects a problem, that I called the *reload-bug*: The applet is running. You press the browser's refresh button. That causes the browser to call your applets `stop()` and `destroy()` methods and to terminate all threads. Concurrently the html-Page is reloaded causing your applet to start again. But the 2nd start is performed under different circumstances than the first one – the classes are already loaded, and the class variables are already set, especially the singleton instances. The result is an inconsistent mixture of old and new data that crashes the applet.

reference copies of strings in class `eniac.io.Tags`. So the typical access (setting and getting) to a property is done by following calls:

```
Status.getInstance().
    set(Status.MY_PROPERTY_NAME, myProperty);
PropertyObject myProperty =
    (PropertyObject) Status.getInstance().
    get(Status.MY_PROPERTY_NAME);
```

4.1.3 Class `eniac.io.Tags`

In this class all XML-tags[3] and attribute names are collected as static fields, and also the property names referenced by class `eniac.util.Status`. The alternative of using this class is to hardcode all strings in the XML-handlers. But it is preferable to have them at one point and to assert a naming convention in order to avoid typing mistakes.

By convention and for readability XML-tags are lowercase, whereas java constants should be named uppercase. The constants in `eniac.io.Tags` formally are not constants, their initial values are null. When the class is loaded they are initialized by their lowercase names using the java reflection framework:

```
Tags.class.getFields();
for (int i = 0; i < fields.length; ++i) {
    // init field by its lowercased name
    String value = fields[i].getName().toLowerCase();
    fields[i].set(null, value);
}
```

This static initialization method and the static strings are the only members of class Tags, so its purpose is purely namespace providing.

4.1.4 Class `java.lang.Words`

The same way of initialization is used for the static strings contained in class `java.lang.words`. This class collects all strings which are displayed to the user, with an exception of error messages (hardcoded) and technical ENIAC names (loaded from configuration files). These strings can be accessed in public, too.

[3] An exception: ENIAC model layer types are not contained. Please refer to 4.1.7

4.1.5 The classloader namespace

The opposite of collecting XML-tags in a java class is referencing classes from an XML-file. So it is done in the *menu.xml* and the *types.xml* files. In the first one, the actions to be included in the menu are listed, the second file maps graphical GUI classes to their data model compounds. In both cases instances are created the following way, as it is commonly known from instantiating a JDBC driver:

```
String myClassName = ...;
Object o = Class.forName(myClassName).newInstance();
MyObject myo = (MyObject) o;
```

The classloader not only is useful to load classes, it also provides access to other kinds of resources located at the classpath. In the ENIAC simulation, all resources are contained in several jar files. These jar files are passed to the applet classloader by HTML-tags. So their content is in the classpath. Now image data can be accessed like this:

```
String myImagePath = ...;
java.net.URL url;
url = getClass().getClassLoader().getResource(myImagePath);
java.awt.Image img = javax.imageio.ImageIO.read(url);
```

and an input stream for parsing XML files can be opened as well:

```
String xmlFileName = ...;
java.io.InputStream in;
in = getClass().getClassLoader().getResourceAsStream(name);
```

The disadvantage is that if a resource cannot be found in the jar-file, an HTTP-request is send to the web server, because the applet's codebase inherently is contained in the classpath. Another way of accessing those resources would to load a jar file by an `java.net.JarURLConnection` and parse its entries by using the `java.util.jar` API. This way has three inconveniences:

- When running the program as application, all resources must be contained in jar files the same, as it were started as applet,
- the names of the jar files must be passed to the program,
- the usage of the `java.util.jar` API is more complicated than the examples above.

Because I don't expect resources to be missed in a software that is under such complete control of the webmaster as an applet, I chose the first way of loading external resources.

4.1.6 Parameters

The common way of parameterizing an applet is by using the HTML `<param>` tag. The common way of parameterizing a Java application is by passing arguments to the string-array, with which the main method is invoked. Because the ENIAC simulation is designed to be runnable both as applet and as application, both ways are supported. Also there is a third way implemented: A hidden properties file named ".eniac_settings" can be written and read from the users home directory.

A parameter representation independent from the input methods is needed. This is given by class `eniac.util.Parameters`. Similar to class `eniac.util.Tags` described above, a parameter is represented as a public static String member and can be accessed in the same way. Reasonable default values are hardcoded, so the applet runs fine without passing any parameter.

The reading of applet tags differs from parsing an arguments array – there is no way to get a collection of all parameters, you have to request them by their names. Because the parameter names are given as class members, the java reflection framework is used to iterate on them and to eventually set them.

```
java.applet.Applet myApplet = ...;
java.lang.reflect.Field[] fields;
fields = Parameters.class.getFields();
    for (int i = 0; i < fields.length; ++i) {
    String key = fields[i].getName();
    String value = applet.getParameter(key);
    if (value != null)
        Parameters.set(fields[i], value);
}
```

In contrast to class `eniac.util.Status`, class `eniac.util.Parameters` only contains properties, which won't be changed dynamically. So no event notification mechanism is necessary. Another distinction lies in the fact that all

parameters are strings. So class `eniac.util.StringConverter`, that provides methods to parse primitives and objects from strings, is a suitable co-worker.

4.1.7 Model layer types

The most advanced issue was to provide a namespace for the ENIAC model layer. Similar to the ENIAC graphical layer, the objects are organized in a tree. So any object has immediate access to its parent and its children. For graphical objects this relation is sufficient, because their task is to paint themselves or a subtree, triggered by the window containing them or their model layer counterparts whom they are observing. For model layer objects there are more problems to solve.

4.1.7.1 XML input and output

An ENIAC configuration can be read from and written to an XML file. Different types of ENIAC components have different functionalities and different looks. To identify the type of the object, I introduced class `eniac.data.type.EType`. Each of its instances has a unified name, that is used as XML-tag. All instances are collected in class `eniac.data.ProtoTypes` and statically initialized by parsing the "types.xml" file when the class is loaded. Its etype is attached to each model layer object, so the name is present while converting objects to an XML stream.

4.1.7.2 Creating graphical objects from model layer objects

In class `eniac.data.type.EType`, a modal layer class is mapped to a graphical layer class. There are types using the same modal class but different graphical representation, as well as types with different models and the same graphics. Instances are created as what I described in 4.1.5.

4.1.7.3 Identifying nodes

The model layer tree contains nodes having several children of the same type. To identify them and maintain their order, every node has an index.

An example from the Accumulator unit: A program pulse comes through a program connector. The Accumulator can identify the connector by getting its index, let's say it is 2. A program has to be started, so the value of a Operation switch has to be read. The corresponding Operation switch can

be identified by finding a child node of type `ProtoTypes.OPERATION_SWITCH` and index 2.

Analogous an Accumulator unit can find its corresponding blinkenlights by requesting the parent node for a child of type `ProtoTypes.BLINKEN_LIGHTS` and having the same index as the accumulator itself.

Another system to identify nodes is their ID. Every data model object has its unique ID and is registered at the `eniac.data.IDManager`. This is useful for components that are connected by a cable. When writing to XML, the partner's ID is serialized, too. So the cable can be restored later.

4.2 ENIAC components

As I mentioned before, the objects representing the ENIAC are organized in an ENIAC component tree. The architecture is predominantly a two layered architecture, assisted by optional helper classes. So in fact there are two isomorphic trees – one for the model layer and another for the graphical layer.

4.2.1 The model layer

The current state of an ENIAC component is represented by a model layer object. The base class of the model layer class hierarchy is class `eniac.data.model.EData` having the following properties:

- name : `String` – this is displayed as mouseover tooltip
- id : `int` – to identify a connector's partner (see 4.1.7.3)
- index : `int` – to order children of the same type (see 4.1.7.3)
- type : `eniac.data.type.EType` – to identify the component's type (see 4.1.7)
- parent : `eniac.data.model.EData` – the component's parent
- gridNumbers : `int[4]` – to locate a component in the parent's gridBagLayout (this information is used by the component's graphical counterpart but must be stored here because it is part of the eniac.xml file)

A common subclass is given by class `eniac.data.model.sw.Switch` representing a Switch component or an information display like the blinkenlights

related to the Accumulator's decade counter. A switch holds two additional properties:

- value : `int` – the selected value
- enabled : `boolean` – whether the user can change the value. Common switches of course always can be changed, but blinkenlights cannot be clicked when the Accumulator has no power.

A subclass of the previous one is `eniac.data.model.SwitchAndFlag` additionally having a boolean flag. This class is used e.g. for an Operation switch and the corresponding clear correct switch, that are grouped together.

4.2.2 The kindergarten

An important property of a tree node is missing in class `eniac.data.model.EData` – there is no field to store child references. This facility is served by class `eniac.data.model.parent.ParentData`, the superclass for all nodes having children. To store the children, the helper class `eniac.data.ModelKinderGarten` is used.[4]

During parsing from XML, all model layer objects are collected in a temporary list. When parsing is finished, from such a child list a kinderGarten can be created, which stores the objects in an array grouped into sections by their type and ordered by their index. This provides constant time access to any child. Storing in an array is possible, because the number of children won't change. The user might add a cable, but cables are outside the ENIAC component hierarchy.

4.2.3 The graphical layer

The base class of all graphical layer components is class `eniac.data.view.EPanel`, a subclass of `javax.swing.JPanel`. An epanel holds a reference to its corresponding model object, where it is also registered as an `java.util.Observer`. So the epanel is notified whenever the state of the edata

[4]The method to access a child is getKind(EType type, int index) : EData. In fact this was a borderline case for my naming conventions – *kindergarten* is a word of German origin and means *garden of children*. So the getter method also could be named getChild(...), but I decided to choose the method name according to class name, though it might be mistaken as an English German language mix.

On the other hand, you can read *get kind* as pure English in the sense of *a kind of*. According to this interpretation, the method's name means: look for the child specified by the given kind of type and the given index. I like this polysemy.

changed, and can adjust the view. Normally this is done by calling the repaint() method inherited from class `java.awt.Component`, that marks the area covered by this component as dirty. Dirty areas will be painted by a separate thread, the "AWT-EventQueue".

Because painting the components is more time-consuming than computing the new state of the model, a few changes are dropped in the view when speeding up the simulation pulse, and e.g. the blinkenlights seem to "jump". This is irritating if you are watching the highlighted pulse. So when the highlighting state changed, the painting is not delegated to another thread. But the component's paint method is called immediately from the simulation thread that also computes the changes in the model. That causes the flashing to stay in the correct rhythm.

Analogous to the edata, an epanel cannot have children. The parent role is filled by class `eniac.data.view.parent.ParentPanel`. A kindergarten is not needed, because the parent-child relation as inherited from class `eniac.data.view.EPanel` is sufficient.

For faster zooming, the bounds of the children are cached whenever the size of the parentPanel is changed. So when the user toggles the zoom between several states, known bounds don't need to be recomputed, they are just loaded from the cache.

4.2.4 The descriptor

In painting and layout, an epanel is supported by an instance of `eniac.skin.Descriptor`. A descriptor is a small data object containing the natural size of an epanel according to a certain level of detail and various other information such as image file names, colors, mouse sensitive areas and a grid for layout a panels children. Descriptors are read from the skin.xml file that contains one descriptor per `eniac.data.type.EType` and per level of detail. If a panel has no descriptor for a certain level of detail, it is not painted. This rule is useful for a low, icon based level of detail, where just units, trunks and blinkenlights are painted. This is the default LOD for the overview window.

Because the data contained in the descriptors is not hardcoded into java classes, the number of classes in the ENIAC component hierarchy could be dramatically reduced.[5] Also the descriptor concept opens the possibility to introduce several skins that can be chosen by the user. And finally it makes levels of detail easy to handle. An array of descriptors (one per level of detail) is attached to every etype.

[5] see also 5.2.4

4.2.5 The control layer

Roughly speaking, the state of a model layer object can be changed in two ways. First, the user can click a component and for example can rotate a switch. Second, a simulation event can trigger such a change.

The first case is supported by interface `eniac.data.control.Controler`. A controller responses to three kinds of mouse events: When the mouse button is pressed, when it is released, and when the mouse is dragging (the mouse is moving with a button down). The appropriate methods of interfaces `java.awt.event.MouseListener` and `java.awt.event.MouseMotionListener` are just passed to the controller. The use of a controller effects, that push buttons, rotary switches and toggle switches all can be represented by the same graphical class, they just need different controllers.

The controller depends on the current descriptor. So components can show different functionality in different levels of details, too – a feature that isn't used in the current version of the ENIAC simulation. Controller instances are created by a temporary instance of `eniac.data.control.ControlerFactory` during the time that the descriptors are parsed from XML. Because a controller is a stateless static object just offering pieces of code, there are just single instances of each controller class. Because the classes are small, they are collected as private inner classes in the controller factory. For components that cannot be changed by mouse actions, a default controller with empty methods is registered.

The second case is not supported by a helper class. All actions are hardcoded in the model layer class. I don't expect any synergetic effects, because the changes caused by the simulation thread differ from type to type, even on the unit level.

4.3 XML files

Instead of hardcoding data in the ENIAC simulation, information is read from XML files. So a lot of changes can be done without changing and recompiling the source code. Also some data can be changed by loading another XML file by a user's command. In this section the file types are introduced.

4.3.1 Parsing XML using the `org.xml.sax` API

The SAX API provides an easy-to-use, event based way of parsing XML and is contained in the Java SDK since version 1.4. Writing a parser means

extending class `org.xml.sax.helpers.DefaultHandler`. This class contains methods to be called when parsing starts or ends, when an element starts or ends and when whitespace was read. Triggered by these method calls you can create objects from the parsed data. In case of a complex DTD, this process should be supported by an internal state machine. Good programming (or rather: debugging) praxis is to include all code of the methods in a general try-catch clause like this:

```
try {
    // object creation code
} catch (Exception e) {
    e.printStackTrace();
    throw new SAXException(e);
}
```

Otherwise any exception occurs during creating your objects will be wrapped into a `org.xml.sax.SAXParseException`, and you will lose the exception's origin. The parsing is started the following way:

```
MyHandler handler = ...;
javax.xml.parsers.SAXParserFactory factory;
javax.xml.parsers.SAXParser parser;
factory = SAXParserFactory.newInstance();
parser = factory.newSAXParser();
saxParser.parse(in, handler);
```

A disadvantage of this API lies in the fact, that for each DTD you have to write your individual handler. In another XML framework, the `org.w3c.dom` API, so-called *dom trees* containing the parsed data are created automatically. But in this case you have to deal with a tree, whose structure maybe doesn't fit your needs and probably contains some overhead. Or you have to convert the tree to your own objects, what means the same effort as writing a sax DefaultHandler.

An advantage of the SAX API is, that you can write several handlers for the same file DTD, even lightweight ones that don't convert all of the data. This circumstance is used by the proxy concept, which is introduced in the following subsection.

4.3.2 The proxy concept

When the user is given the choice of loading several files, he needs information about the file contents, a kind of meta-information. In the ENIAC simulation, this information is stored on top of the xml file within an element called "proxy". This tag is read by an instance of `eniac.io.ProxyHandler` that creates an instance of `eniac.io.Proxy`, but will ignore all other data. The names of the elements contained in the "proxy" element are not fixed. They are put into a `java.util.Hashtable` and mapped to the whitespace they contain.

The data included to the proxy is the data that will be displayed to the user when selecting the file from a dialog. But also some data snippets that does not fit into the XML file structure can be included. As an example I list the proxy-element of the skin.xml file:

```
<proxy>
    <name>default</name>
    <description>This is the default skin</description>
    <author>Forename Surname</author>
    <email>mail@institution.org</email>
    <preview>skin/preview_0.gif</preview>
    <number_of_descriptors>89</number_of_descriptors>
    <number_of_lods>2</number_of_lods>
    <zoom_steps>80,100, ... ,2282,2853</zoom_steps>
</proxy>
```

When the data finally is loaded, the contents of the proxy tag will be ignored. The proxy concept is used by the eniac.xml, skin.xml and lang.xml files.

There is also a class `eniac.io.ProxyScanner`, which indexes a given filename to a maximum range, and tries to load and parse the indexed files from the classpath (see 4.1.5). E.g. when the user wants to change the language, the proxy scanner tries to parse the files lang_0.xml .. lang_5.xml, because as maximum index 5 is set as the value of `eniac.util.Parameter.MAX_LANGUAGE_INDEX`. It is the webmaster's responsibility to set the index to the correct number.

4.3.3 XML file types

In this subsection the five XML file types used by the ENIAC simulation are described. Of the first three file types, many instances can exist, because they can be loaded from inside the ENIAC simulation by a menu control.

4.3.3.1 eniac.xml

As described in subsection 4.1.7, all data of the model layer tree is parsed from this file. Except a proxy, only elements representing ENIAC component types are contained. Here is an excerpt of the XML representation of an initiating unit (leaving out the name attributes):

```
<initiating_unit id="172" grid="0,1,1,3" index="0">
<clear_button id="173" grid="3,2,4,3" index="0"
     value="normal"/>
<go_button id="174" grid="2,2,3,3" index="0"
     value="normal"/>
<heaters id="175" grid="1,2,2,3" index="0" value="on"/>
<initiating_image id="176" grid="0,0,4,1" index="0"/>
<initiating_symbol id="177" grid="0,2,1,3" index="0"/>
<program_connector id="178" grid="2,3,3,4"
     index="0" io="out" partner="188"/>
</initiating_unit>
```

4.3.3.2 skin.xml

The descriptors parsed from this file are used by the graphical layer to draw ENIAC components on the screen (refer to subsection 4.2.3). The data stored in the descriptors is created by subclasses of abstract class `eniac.skin.Creator`. Instances of these classes itself are created by `eniac.skin.CreatorFactory`, which also contains them as private classes. A similar concept is used to create controller instances (see 4.2.3). The following XML snippet represents the *go button* at the medium level of detail:

```
<descriptor type="go_button" width="30" height="30"
     fill="none">
<single class="Controller" name="controller">
     PushButton
</single>
<array class="Image" name="back_image_array">
<entry>go_button.gif</entry>
<entry>go_button_pushed.gif</entry>
</array>
</descriptor>
```

4.3.3.3 lang.xml

Language data commonly is represented in resource bundles, in Java represented by class `java.util.ResourceBundle`. The Java framework also provides methods to load resource bundle contents from a properties file. Nevertheless I chose the XML representation, in order to maintain a unified data concept and to be able to use the proxy concept (cf. 4.3.2). Also the static access to language data in class `eniac.lang.Words` (cf. 4.1.4) does not work well with the Java resource bundle, because it stores language data in a hash table. Currently only English and German language files exist for the ENIAC simulation. An example of the English language file:

```
<entry key="frame_title">Eniac Applet</entry>
<entry key="overview_window_title">Overview window</entry>
```

Also multi-line whitespace (e.g. for the FAQ) is supported.

4.3.3.4 menu.xml

The menu.xml file is not an indexed xml file, because it is loaded only once in the applet's life cycle. For more details about the menu, please refer to the following section. The following code represents the first two buttons in the ENIAC simulation toolbar:

```
<group key="file">
<action icon="open_configuration.gif"
     class="eniac.menu.action.OpenConfiguration"/>
<action icon="save_configuration.gif"
     class="eniac.menu.action.SaveConfiguration"/>
</group>
```

4.3.3.5 types.xml

Also the types.xml file is loaded once. This is the code to create a go-button. The codes-section defines how the possible values are encoded at the eniac.xml file:

```
<type name="go_button">
     <model>eniac.data.model.sw.Switch</model>
     <view>eniac.data.view.sw.SwitchPanel</view>
     <codes name="value">
          <code>normal</code>
```

```
            <code>pushed</code>
        </codes>
    </type>
```

4.4 Actions

What is an action? An action is an operation triggered by the user that changes the state of an object. Actions usually can be started in one or several of the following ways:

- by activating a user interface component by a mouse action
- by selecting an item from a menu
- by hitting a shortcut on the keyboard.

Because one action can be triggered from several sources, it is useful to separate the action performing (this is a more or less parameterized automaton routine) from the action representation. This is known as the command pattern introduced by Gamma et aliter [11] and is supported by the interface `javax.swing.Action` of the Java framework.

4.4.1 EAction and ToggleAction

In the ENIAC simulation, actions are represented as subclasses of the abstract class `eniac.menu.action.EAction`, itself subclassing `javax.swing.AbstractAction`. Class EAction contains methods to create a `javax.swing.JButton` and a `javax.swing.JMenuItem` as the default components to integrate this action into the GUI.

An eAction[6] represents a single state action that can be enabled or disabled. There are three abstract methods to be implemented by subclasses: one method to perform the action (inherited from interface `javax.swing.Action`) and two methods returning the action's name and short description. The name will be written in the menu, the short description displayed as tooltip. An icon representing the action as jButton is expected to be set during initialization process. An eAction registers itself as listener. So it is informed, when the language changed, and can reload its name and short description. As an example EAction, I'd like to mention

[6]Class names are capitalized according to the javadoc guidelines. If the class itself is focused, the first letter is capitalized. If an instance of that class is meant, the first letter is in lower case.

class `eniac.menu.action.About`, which opens a small window displaying information about the program.

Abstract class `eniac.menu.action.ToggleAction` represents a two-state action controlling a boolean value, e.g. whether the pulse will be highlighted or not. Over here a default implementation of the action performing method is given – a boolean value at the status-singleton, identified by a key, is toggled. The key should be one of the static strings in class `eniac.util.Status`, and should be passed to the ToggleAction constructor. There are integrative methods that return a `javax.swing.JToggleButton` respectively a `javax.swing.JCheckBoxMenuItem`.

4.4.2 MenuManager and MenuHandler

All actions are initialized by the menu manager which is implemented by class `eniac.menu.MenuManager`. This is done by the help of class `eniac.menu.MenuHandler`, which parses class names and icon names from an XML file. Actions are grouped as well in order to create named menu items as `File` or `Zoom`. So you have full control from outside the Java source code of which actions should appear in which order. The menu manager also provides methods to create a `javax.swing.MenuBar` containing all actions as menu items and a `java.swing.JToolBar` containing them as buttons. See **??** about how to add a new action to the ENIAC simulation.

Chapter 5

The software development process

5.1 General aspects

5.1.1 Inner and outer software quality

Software quality strongly depends on a suitable architecture. Why that? Two programs might have exactly the same functionality but rely on a completely different architecture. From the user's point of view both programs are the same, because they do the same. Functionality belongs to the *outer quality* of software. Other aspects of outer quality are performance, scalability and stability.

In contrast, the *inner quality* of software is invisible to the user. Aspects of inner quality are the readability of source-code and functional extensibility. In the life cycle of a software product, about 40% of overall effort is done for upkeep and maintenance. To reduce costs, the software should be designed in a way that you can easily do small changes or functional enhancements.

5.1.2 Top-down versus Bottom-up

In a professional software process, there are usually more than one person involved. So several roles are necessary: The Software Architect designs software by using design tools, the Software Developer implements the software, the Software Engineer converts it to a product and the Project Manager controls this process. Last but not least, you have the customer who ordered the software

If you are writing software as a single person, you – roughly saying –

have to fill these roles by yourself. But in doing this you can avoid a lot of communicational overhead. For the cooperation between a Software Architect and a Software Developer UML is a main tool. The architects fix the design into UML-diagrams that the developers have to implement. This is a Top-down process of designing software. As a single person you don't need a UML-diagram in the design state, because you don't need to explain your design to anybody but yourself. It is sufficient to have the diagram in your head (if your head is clear enough to keep it), and in your head it is easier to modify. On the other hand, you don't need the diagram at all, when you design your software bottom-up.

5.1.3 Maintaining a Prototype

Bottom-up software design means maintaining a prototype. A *prototype* is a piece of software having one or several features that the final product is planned to have. You step into the development process by establishing a prototype, then you include one feature after another until the product is ready. This has the advantage, that you can test a new feature immediately after it is included and can keep on the interaction of the existing features. As well you have something to show to your customer.[1]

If you are architect and developer in one person, you can start coding before having the whole plan. You are writing a swinging Applet? So there will be a starter class extending `javax.swing.JApplet`. Should it be possible to start the program as an application, too? So you need a main-method. Because you want your program to be scalable, there should be a way to pass parameters. In fact there are two ways – the applet way by param-tags, and the application way by the arguments string array. Within the program there should be a global access point to the parameters and its values.

In fact these were the first decisions I did in designing the ENIAC-simulation. And these decisions were made during coding, or rather: *by* coding. As I saw there are two ways to pass parameters, I decided to abstract keeping the parameters from the starter class. Generally speaking, there is a lot of code that can be written independently from the whole plan and by writing this code you can identify the next problem that needs to be solved.

Of course this sometimes leads you to a dead end, but dead ends are part of the concept. In this case you have to change your concept and refit it.

[1] Refactoring, Continuous Integration and Test Driven Development are core practices of Extreme Programming. Please refer to [10]. Pair Programming, another core practice, hardly can be performed by a single programmer.

These development phases are called *refactoring phase*. So your concept of the whole software involves together with your source code.[2]

5.1.4 The project manager and the customer

Software projects are always late. Moreover, they seem to be inherently late. This relies on the many uncertainties in the software developing process. So the project manager's job is to keep these uncertainties as small as possible. One way to do this, is to divide the development process into several steps and bring them into a time line. Then you can estimate the amount of time for every step and make a timetable.

But as I said before, some steps lie in the far future and the circumstances under which they will be done are unknown. So the estimation is a kind of blue. Also some steps might be forgotten or it turns out, that another way to go is more suitable for the project than initially planned. And in fact, planning takes time and can be overhead as well.

From the customer's point of view the product and its time of delivery counts most. He supposedly had a purpose to order the software and is happy if it fits his needs. If the hours the development team spent developing don't count for the costs, the customer don't bother about them. But similar to the project manager he has an interest to track the progress of the project in order to estimate the date of delivery.

As mentioned above, it is hard to plan a software project in all steps at the beginning, especially when you are developing prototype oriented.

---

[2] You can describe the process "writing a program" in the words of "playing a chess game": The opening can be played easily, because these first moves are played a hundred times. In the mid-game you are facing problems and have to think. You need to find a plan that fits to the individual situation of this game. Sometimes your approach is not suitable and you have to change it. The endgame is a matter of technique.

The two ways of thinking during a chess game are called *strategic* and *tactical*. Strategic thinking is a long term approach to reach a good position and is top-down. Tactical thinking means computing the next moves in order to find a way to a good position and is bottom-up. Finding a move needs exact computation and can excellently performed by a computer, while analyzing a position is fuzzy and still is better done by humans than by computers, unless you use neural networks. But this is another diploma thesis.

What I want to point out in this excursion is the fact, that talking about programming is metaphorical. Except 0 and 1, anything in a program is virtual and only can be understood in terms borrowed from another area of reality. If you are talking about a *remote procedure call* you won't have two yodeling Bavarians in mind, but the words indeed are borrowed from the human sphere and mapped to the vast field of bits and bytes. So using concepts from different sources is like lighting the topic from several perspectives and can bring new aspects to light. The important role that metaphors play in reasoning have been pointed out by Lakoff and Johnson [12] in an exemplary manner.

On the other hand, the customer often has no clear idea, what kind of software he needs. It is more like that he realizes a gap, that needs to be filled, like a job routine, that he suspects to be more efficient with software support. So it is in the customer's interest to see unfinished products, maybe lacking functionality at all, because they help him to find out what he wants. Frequent interaction with the customer is important for the success of a software project, because the demands can be redefined.

In a one man developing team, the developer is also the product manager. Because developing takes more time than controlling, he identifies himself more likely with the developer's role. This tempts him to neglect his responsibilities as project manager, because with fewer control, the developer feels more free, and coding becomes rather playing than working. Techniques have to be found to maintain the project manager's firm role.

If you are developing a small tool that manages your photo collection on your hard-disk, you are your own customer as well. But in the case of a diploma thesis, you have to deal with your professor and meet him from time to time. In the following section I will elaborate the initial time-line for my diploma thesis, how I tracked its progress and shot troubles, and in which way I had to modify the time-line.

5.2 Developing the ENIAC simulation

The basic setup for starting the development process was simple. I decided to use Eclipse[3] as development environment and to support building and backing up by a few shell scripts. I took JDK 1.4 as Java version, because I wanted to use swing and take XML for data io.[4] I had written several

[3]The many features of Eclipse are more worth than to be handled in a footnote. But because I didn't use them for the ENIAC simulation, I don't report them. Here is what the Eclipse consortium writes about its product:

> Eclipse is an open platform for tool integration built by an open community of tool providers. Operating under a open source paradigm, with a common public license that provides royalty free source code and world wide redistribution rights, the eclipse platform provides tool developers with ultimate flexibility and control over their software technology.[9]

[4]If you are choosing the suitable JDK, the main choice is between JDK 1.1 or JDK 1.2 an newer. For web browsers there are three families of virtual machines:

- Suns JVM, which was developed continously and is now obtainable in Version 1.5 beta
- The Microsoft VM whose development was stopped in 1999, but which has been

(smaller) Java applets before, so the general architecture was clear to me. I planned a 2-layered GUI architecture and a discrete event simulation embedded into a singleton-based framework. Roughly my time-line for the ENIAC simulation was like this:

5.2.1 A time-line for the ENIAC simulation

1. Establish a suitable developing environment
2. Write a basic framework in order to establish a prototype
3. Program an accumulator and a cycling unit in order to perform the Two accumulators test
   (a) Define an XML representation for the units
   (b) Write the data layer
   (c) Write the graphical layer by using provisional graphics
   (d) Write the discrete event simulation
4. Program the other units one by one
5. Final tasks
   (a) Exchange the provisional graphics by professional ones
   (b) Write multi language support
   (c) Do extensive testing
   (d) Write javadoc-style comments
6. Write this documentation

---

shipped together with the Microsoft Windows operating system until a few years before

- The Symantec VM which was common for the Netscape Navigator 4.x and might be neglected now

The last two virtual machines only support Java 1.1. So if you want to avoid that the user has to install anything before starting your applet, you should choose Java 1.1. But if you rely on swing (as the ENIAC simulation does) you have to use at least JDK 1.2 (which is also known as Java 2, because of its many new features compared to JDK 1.1). In this case the right choice is a version, that has been introduced at least a year before. Within one year almost all university PC pools, company intranets, for which a professional administrator has to care, should be updated to the current version.

5.2.2 The to do list

For further details I established a To Do list.[5] It was divided into the following sections:

- Bugs – bugs that need to be fixed
- Features – functional enhancement of the program
  - New Features – Features that need to be programmed in order to step forward in the time-line
  - Nice features – Features that would be nice to include, but that are not prerequisites for other ones
  - Final features – Features that should be done after finishing the above ones (e.g.: write DTDs for the several kinds of XML files)
  - Future features – Features that go beyond the time-line but would be great for future releases (e.g.: a 3D version of the ENIAC simulation)
  - Maybe features – Features for which it is undecided whether they are welcome (e.g.: Fit the ENIAC simulation window to screen size at startup)
- Refactoring – code that became old and should be rewritten.[6]
- Documentation – documents that support the developing process
- Organizational tasks – contacting the real world (e.g.: arrange an appointment with the professor)

[5] What is the correct spelling of this term? Google counts

- "to do list" – 1,420,000 hits
- "todo list" – 601,000 hits
- "todolist" – 41,400 hits
- "to dolist" – 604 hits

[6] this differs from the dead ends as mentioned above. Refactoring tasks listed here will increase the design quality, but they are not necessarily to be performed. You won't include an "escape from dead end"-task to the list, because escaping from a dead end cannot be scheduled to the future.

5.2.3 The diary

Another document I maintained during developing the ENIAC simulation was a diary. Every day after I finished working for the project I wrote a short note describing what I have done. I also counted the current number of lines of code and the time I used for my work.[7] Tracking the working hours does not mean tracking the progress of the project. But it is a kind of self-control and indicator of my motivation to work at the ENIAC simulation. From the data collected in the diary, facts about the development process can be extracted as I will do in the following subsection.

[7] I didn't count the lines by hand but wrote a shell script to do so. After I spent about an hour looking for a suitable stopwatch I decided that it was faster to code it myself. The tool shows up as a digital clock with a start/stop button attached and is visible in the task-bar as hh:mm:ss.

Later I thought about combining the stopwatch with the to do list in order to provide a GUI for the list and to track the time I spend for every specific task. This would strengthen my project manager role. But as a good project manager I decided that this was counterproductive because a lot of time that would be spent for this subproject.

5.2.4 The factual time-line for the ENIAC simulation

Date	Event
Aug 7, 2003	Downloading Eclipse 3.0 M2 and start of project
Sep 19, 2003	1st version released. It can read units from xml and display them as dummies
Nov 29, 2003	XML output finished
Jan 10, 2004	Downgrading to Eclipse 2.1.2
Jan 14, 2004	2nd version released. It displays a dummy cycling unit and an accumulator with switches, connectors and blinkenlights but without functionality
Feb 5, 2004	Introducing a skin concept. GUI and image data are decoupled so that the user can choose his favorite surface. Units can be displayed in several levels of detail.
Mar 1, 2004	3rd version released. Trunks and Trays are now working.
Mar 13, 2004	Descriptor and DataType concept introduced. Refactoring of the GUI is finished, which lightens the code by about 30 classes and 2500 lines of code.
Mar 17, 2004	3rd version released. User can set cables.
Mar 25, 2004	Menu is read from XML
Mar 31, 2004	Cycling unit is finished
Apr 10, 2004	Discrete event simulation and Initiating unit are finished
Apr 21, 2004	Introducing internationalization.
May 5, 2004	4th version released. Initiating unit, Cycling unit and Accumulator unit are complete. Pulse interaction can be highlighted.
May 30,2004	5th version released, the final version of my diploma thesis. The Constant Transmitter unit is implemented, two Accumulators can be switched together, and the Fibonacci number example was programmed.

Chapter 6

Summary and Outlook

The ENIAC simulation consists out of 116 Java classes containing 14929 lines of code, 10 XML files with 10248 lines of code and 180 GIF images.

In 2003, Peter Hansen programmed a simulation of the ENIAC as his Bachelor thesis.[8] This is the first known simulation of the ENIAC. He used a different approach from mine and implemented the GUI without customized graphics by using the Java 1.1 AWT components. So with fewer effort, about the same number of units as in the ENIAC simulation could be implemented. In contrast to the pulse-leveled ENIAC simulation, Hansen's simulator runs at the addition cycle level. So his simulator is even faster than the original ENIAC.

In one section of his work, similar to this one, Hansen reasons about what kind of extensions to his software could be made. Some of the features listed there are provided by the ENIAC simulation now:

- The low level hardware properties of the ENIAC are represented in the ENIAC simulation.
- Two Accumulators can be interconnected in order to compute 20-digit numbers.
- The Initiating unit provides step-by-step execution.
- The Accumulator neon bulbs are "dancing" during a computation.

Some more features not on this list are implemented by the ENIAC simulation:

- Different ENIAC hardware configurations can be loaded from an XML file.

- An enhanced graphical user interface is provided aiming the look of the original ENIAC.
- Several skins for the graphical representation can be designed and changed dynamically
- Cables and connectors can be highlighted when a pulse goes through them.
- The speed of the ENIAC can be controlled continuously.

But there are tasks open for the future as well:

- Implement the outstanding units of the ENIAC.
- Implement the IBM card reader and a push-card editor.
- Design more complex programming examples like an algorithm to solve the 8-Queen-problem.
- Extend the logging system in order to have an efficient debugging tool.
- Extend the current skin by one more level of detail having about 150% of the size of the currently biggest one.
- Collect photos of the original ENIAC or shoot new ones of the surviving parts in order to create a photo-realistic skin.
- Enable to open several views of the ENIAC, e.g. as split screen.
- Write a "remote control" for the initiating unit – a small dialog window like the overview window containing a second set of the step button, the iteration switch and the frequency slider.
- Write a 3D version of the graphic component layer.
- write a C-Compiler that can translate C-Code to eniac.xml files.
- Design fantasy ENIAC components like a Tic Tac Toe unit.

Bibliography

[1] J. P. Eckert Jr., J. W. Mauchly, H. H. Goldstine, J. G. Brainerd: *Description of the ENIAC and Comments on Electronic Digital Computing Machines*, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[2] H. H. Goldstine and A. Goldstine (1946): *The Electronic Numerical Integrator and Computer (ENIAC)*. In B. Randell (Eds.): *The Origins of Digital Computers*, Springer-Verlag (1982).

[3] J. Van der Spiegel, J. F. Tau, T. F. Ala'ilima, and L. P. Ang (2000). *The ENIAC: History, Operation and Reconstruction in VLSI*. In R. Rojas (Eds.), *The First Computers; History and Architectures*, MIT Press.

[4] J. Van der Spiegel, *ENIAC-on-a-Chip*. `http://www.ee.upenn.edu/~jan/eniacproj.html`

[5] D. Winegrad and A. Akere: *A Short History of the Second American Revolution*, In: *ENIAC's 50th Anniversary: The Birth of the Information Age*, The University of Pennsylvania Almanac N. 42, Jan. 1996. `http://www.upenn.edu/almanac/v42/n18/eniac.html`

[6] K. A. Kleiman: The ENIAC programmers, 1997, `http://www.witi.com/center/witimuseum/halloffame/1997/eniac.php`

[7] Konrad Zuse Internet Archive `http://www.zib.de/zuse/EnglishVersion/`

[8] P. Hansen: *A Java Simulation of the ENIAC*, bachelor thesis at University of Osnabr\"uck, 2003. `http://home.arcor.de/-ph/eniac/`

[9] Eclipse Consortium, `http://www.eclipse.org/org/`

[10] R. Jeffries: *What is Extreme Programming?* In *XP Magazine*, 11/08/2001, `http://www.xprogramming.com/xpmag/whatisxp.htm`

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides: *Design Patterns*, Addison Wesley 1995

[12] G. Lakoff and M. Johnson: *Metaphors we live by*, Chicago and London 1980

[13] Wikipedia – The Free Encyklopedia, `http://en.wikipedia.org/wiki/Main_Page`

[14] The Jargon File – article about *blinkenlights* `http://www.jargon.net/jargonfile/b/blinkenlights.html`

[15] `http://www.fitg.de/images/literaturhaus/eniac.jpg`

[16] `http://edutech.xmu.edu.cn/hongen/pc/newer/rumen/work/img/eniac.jpg`