Web browsers prescribe the ways we access and navigate knowledge and communities online. Since the 1990s browser software has been an arena for artistic interventions ranging from quirky standalone browsers to performative pieces to minimalist browser add-ons. The (im)possibility of navigation is not taken for granted and is probed, questioned, and reformulated through such software practices. We propose navigation as a mode of exploring interactive software that allows researchers to collectively document manifold facets of artists' browsers.

Navigation

Reihe
Begriffe des
digitalen Bildes

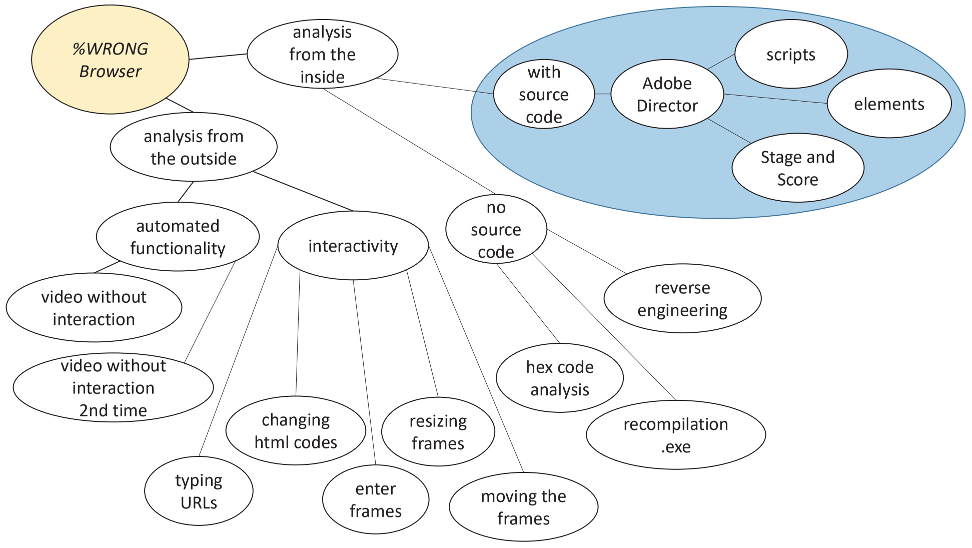
Navigation

Edited by
Inge Hinterwaldner
Daniela Hönigsberg
Konstantin Mitrokhov

Munich, 2022
Open Publishing LMU

Content

Fig.1. Structure of the analysis and documentation of JODI's %WRONG Browser.



Martina Richter

Descending into detail – a top-down approach to documenting JODI's %WRONG Browser (co.kr.exe)

When I started thinking about how to document the *%WRONG Browser* I wanted to do it in a structured way. As a computer scientist, my general approach when solving a problem is to first look at the whole system, then divide it into smaller, more manageable packages. I follow this procedure in an iterative, recursive and methodical manner. With each division, I continued creating new smaller levels of packages until they had a scope that allowed me to easily access the information I needed. On each level, the packages were examined to either find the next smaller package or retrieve the sought-after information.

With this general approach in mind, my examination of this specific software proceeded in the way I have described. I divided the whole software system into as many parts or elements as I could identify to gain an understanding of how the browser application works.

I also used two theoretical lenses in the method of documenting the browser. The first one – the perspective from the outside of the software – takes into consideration what the user can perceive and experience while using the software. The second – the perspective from inside the software – focuses on finding out as much as possible about how the software is built and how it works in its dynamic processes.

Fig. 1 shows a tree diagram displaying the packages I identified and examined successively and which method and access points of investigation I used. The right side of the diagram shows the from-the-inside approach. The part of that branch that is encapsulated in the blue area contains the part of the analysis which would need the source code and was therefore not included here. That leaves this branch with just limited options. From my disciplinary perspective, it was a challenge to conduct the documentation and retrieve the information about how the browser works without including the source code in the analysis, but I succeeded nonetheless in gathering some relevant information about the inner workings of the program. The artists kindly shared the source code of their work with us later so that an analysis based on the source code is included at the end of this volume.

1 First level of division: Perspectives for looking at the application

The first step in approaching the application was to estimate what system parts could be identified and what perspectives would be most effective in approaching them. For that I established the two lenses I previously mentioned: the outside perspective of the user and the attempt to view the application from the inside by various methods.

1.1 Outside lens – The browser from the user's perspective

Taking the user's perspective, the first question I wanted to answer in my documentation of the *%WRONG Browser* was what the user perceives when using the software. As the user is addressed audio-visually, the media I used to document the perceivable output also had to be audio-visual.

As a second question, I sought to find out what exactly the user is able to do when interacting with the application. In other words: what kind of interactivity does the software offer and what possibilities for interaction result for the user? One option, of course, is not to interact at all and simply watch the application run, observing its behaviour. By addressing these two questions I hoped to capture all possible in- and outputs of the software.

1.2 Inside Lens – The perspective from inside the browser software

The next level of the investigation was to discover and document the technical structure of the application: the view from inside the software. The most important point was to find indications about what programming language was used. This information would yield details about the principal structure of the source code. The structure would differ significantly depending on whether it was a program scripted in an object-oriented language or composed of files created with a multimedia-authoring tool like the Macromedia Director software. It would tell me about certain aspects of the project's programming. In an object-oriented language I would find scripts structured by classes and objects. In the Director files, I would find a stage and a timeline binding the multiple scripts and elements together. The result would be a fundamentally different structure of the application's build.

In this branch of the analysis, I also tried to ascertain the functions and elements of the software and determine how they work together. This also involves finding the necessary steps to do this.

2 Second level of the division- View from the Outside

Having determined the first level of division, I then followed the two resulting branches with suitable methods. First, I focused on the examination of the user's perspective, the view from the outside onto the running application. Here, the perceivable dynamics were the main interest of the investigation. Therefore, I took the role of the user and observed the systems behaviour I was confronted with.

2.1 Observation

To find out if the sequence of what I saw and heard stayed the same with every new execution of the application or whether differences could be observed, I started the application *co.kr.exe* on my PC[1] and first watched the screen output without engaging in any interactions. I took notes of the audio and visual outputs. I started the browser again and did the same for a second time; just observing. The result was that there were distinct differences between the two executions of the application. My conclusion was that an unpredictable element was probably used in the form of a randomizing function, to create the deviating output.

2.2 Interaction

After following that trail, my next aim was to interact with the browser. I started playing around and wanted to find out what possibilities the application offered for interaction. I did that for a while, trying to interact as much and in as many different ways as possible. Then I started to collect the differ-

1 Lenovo MT 20T0 BU Think FM ThinkPad T14s Gen 1 with a Intel(R) Core(™) i7 - 10510U CPU, running Microsoft Windows 10 Pro 10.0.19042.

ent interaction possibilities, compiling them in a list to then systematically test them in subsequent trails.

The five forms of interactions that I discerned through this initial visual inspection were activities that could be performed with the mouse: clicking, double clicking, dragging and marking. I was also able to interact by entering characters via a keyboard.

2.3 Documentation of the interaction and documentation tools

As I compiled my collection, the next step was to work out how to document the identified possibilities of interaction.

The aim was to record the visual screen output as well as the sound. Obviously, the appropriate way to address this was to take videos of the screen. Because screenshots cannot show timing, movements or sounds, I discarded that idea immediately. Using the list of opportunities of interactivity, I systematically created separate videos of about 2–3 minutes for each element of the list, showing only the one targeted interaction.

The challenge was to find a suitable PC application that could record the whole screen as well as the sound output and that could be started and stopped by keyboard commands – necessary to prevent the process of switching from the record application to the *%WRONG Browser* from becoming part of the recording. This was important because it allowed me to create clean and discrete videos of the specific interaction behaviours without any distracting activities that were unconnected to the targeted interactions. Unexpectedly, it was not an easy task. It took a long time to find and try different applications. After several trials, I found that OBS Studio fulfilled all my above-mentioned requirements.

The result of this step was two videos without interaction and one for each of the five interactions, a total of seven videos, each several minutes long.

3 Second level of division- View from the Inside

As described above, the second part was to look at the application by applying an inside-lens and to document these findings as well. My aim was to go into the software and divide it into as many parts as possible on a technical level. I wanted to find out how the software was developed and in which programming language. I also wanted to extract the code, to examine the techniques, to identify the components used on the programming level and to see what else I could find just by having the application running on my computer.

3.1 Working without the source code

Normally when software is analyzed on the technical level to determine its functionality, its source code is available to be studied. The source code can be divided into its elements and functions which allows me to analyze what exactly happens while I am running the application.

In this case, I had to find ways to gain equivalent information from the executable application. I followed the idea of reverse engineering, that means the approach of drawing as much as possible from the binary file in order to analyze the system parts and how they work together. That can be done on different levels: on the binary file itself, on a disassembled/ assembler level, that is on a machine language level, and finally on a decompiled level, which means creating a source code in a high-level programming language out of the binary file that is not the originally programmed one but which performs in the same way.

The different approaches I adopted and describe below were not selected or applied in a strictly goal oriented manner but were rather forensic in nature. I tried a variety of methods in order to gain as much information as possible and use this to develop a better understanding of the application's source code and how it is structured.

3.2 Binary file

Applications are usually programmed in a high-level computer language. These are computer languages that are easy for humans to read and write, e.g. C, Perl, Java, or Python. For the computer to read or understand these languages, the code has to be translated into binary code. The result of this translation is the executable program. However, in this form, the code is more or less impossible for humans to read.

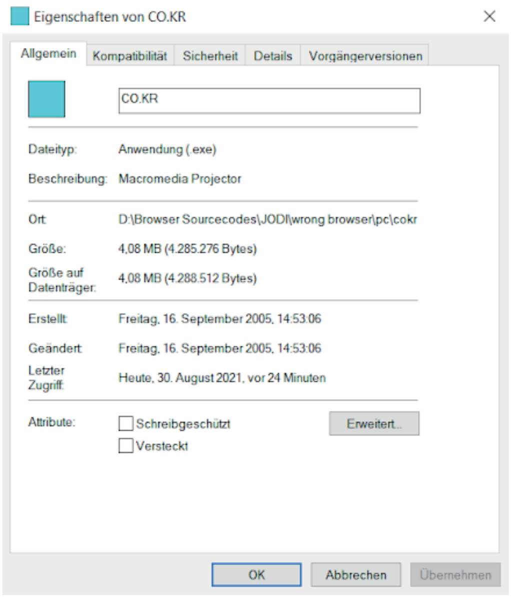
The executable binary file of the application analyzed here, is downloadable as a zipped file cokr.zip here: http://wrongbrowser.jodi.org/. I started by extracting it to co.kr.exe and examining the binary file.

3.2.1 File properties

When looking into the file properties of the executable by a right mouse click, I found some general information about the application (Fig. 2). Looking at the tab "Allgemein" (General) one can see the date and time of the compilation and that it is a Macromedia Projector file. Going to the tab "Details" yields additional information about the Macromedia Director Version.

This information proved useful because knowing the development environment of an executable allows me to find decompilation methods that are particular to the specific version of the environment. The fact that it was a Macromedia Projector file led me to the next step, which was to look for a method or an application to extract more information from the source code by decompilation.

Fig. 2, .exe-file property menu window.



Eigenschaften von CO.KR

CO.KR

Dateityp:	Anwendung (.exe)
Beschreibung:	Macromedia Projector
Ort:	D:\Browser Sourcecodes\JODI\wrong browser\pc\cokr
Größe:	4,08 MB (4.285.276 Bytes)
Größe auf Datenträger:	4,08 MB (4.288.512 Bytes)
Erstellt:	Freitag, 16. September 2005, 14:53:06
Geändert:	Freitag, 16. September 2005, 14:53:06
Letzter Zugriff:	Heute, 30. August 2021, vor 24 Minuten
Attribute:	☐ Schreibgeschützt ☐ Versteckt

3.2.2 Disassembling with a Hex Code Viewer

With the application PE Explorer it was possible to depict the binary co.kr.exe file as a hex coded file. The hex code depiction of a binary file always shows 4 bits together as a hexadecimal number. This display is slightly more readable than a mere series of '0' and '1'.

I used some tools of the PE Explorer to collect additional information and "read" in the binary file. Looking through the information, I was able to identify the operating system on which the source code was compiled, the date and time of compilation and the processing unit.

I also learned more about the software dependencies, meaning what external software libraries were used to compile the source code. I also was able to obtain and save a list of used strings.

The PE Explorer software is able to disassemble the hexadecimal file. This makes it possible to access an assembler software level. Assembler is a machine-near software or language level between high-level language and binary code. The disadvantage of this code level is that no understandably structured source code is being generated. The variables are not discernible and the result is extremely long (in this case 24,988 lines of code). The created code differs so much from the original source code, and is on such a machine-near level, that it does not lead to a significantly better understanding of the code. Or at least it would have taken a very long time to gain any useful information. Therefore, the next step was to try and find a way to further decompile the code to reach a high-level language.

It was possible to gather some information by looking at the hexadecimal coded file and even more when this code was disassembled, but in the end it did not help me to understand how the software works or to determine the structure of the source code.

3.2.3 Decompiling

I embarked on a longer period of Internet research: which applications could help me to obtain more information about the Macromedia Director source code (which is composed of the scripts, elements, score etc.) or even get the source code by decompiling the executable application? I read a lot in blogs and Internet forums, trying to gain a better understanding of what a Projector file is and my chances of success. The results of my research were rather disappointing. I realized that the chances of gaining any insights were very limited and my goal of getting the source code was clearly out of reach using these methods.

My research also revealed the general limitations of decompilation: the decompiled executable application provides a source code that corresponds to the executable, but it will never be the original source code. The reason for that is that programming is never unambiguous as it is possible to reach the same goal, to produce the same effect in the executed program behaviour with completely different source codes. As mentioned before, original variable names and also comments will be missing in the created code because they can not be reconstructed from the binary code and therefore get replaced by random characters or numbers. This detracts immensely from its readability and the chances of understanding its structure. Disregarding these discouraging prospects, I tried two ways of decompiling the binary code of the project.

3.2.3.1 From .exe to source code

My Internet research did not reveal any application that would decompile Projector files, the executables created with Director. Although the language used in Director is Lingo, I turned to an application that usually is used for C++-.exe. My aim was to determine what the result looks like in principle and whether it was worth putting any more energy into it. As previously described, the executable does not offer any kind of information about the high-level language used to write it. Consequently, the result is something that presumes the program was written in C++ and creates a code that could theoretically be the source of the executable in that language.

This procedure produced a result but it was unreadable (Fig. 3). There are, of course, no original variable names, there is no understandable structure, no modules, objects or classes. So all the features or properties that make a code readable and understandable for humans, are not part of the decompiled code.

3.2.3.2 From .exe to Shockwave flash

During my research I found an entry in a Macromedia forum with someone asking for a way to decompile a Shockwave Flash file.[2] This post and numerous other search results pointed to a close relationship between Projector files and Shockwave Flash files – because Lingo is the main language used for Adobe Shockwave Flash, making it potentially possible to use similar tools on both. I had already gained some experience with decompiling Shockwave Flash files during the analysis of another artistic project. Using the same tools on this executable, I hoped to create a Shockwave Flash file, from

2 Anonymous: Help decompiling SWF! In: stackoverflow.com, 11.11.2010, https://stackoverflow.com/questions/4150912/help-decompiling-swf [accessed 27.8.2021].

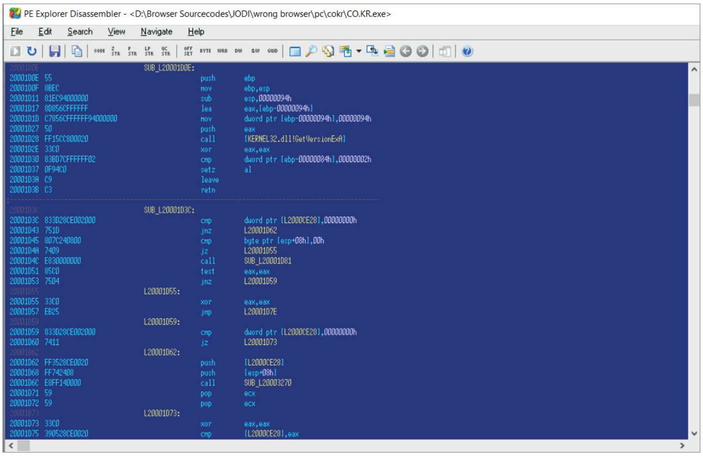
which I could extract Director elements like scripts, images, sounds, timing etc. I looked at several applications, but only some of them allowed the executable to be used as the source for the decompilation. In the end I tried two different applications but the results were as disappointing as with the previous trials. I was only able to extract shredded information, like a vector shape (*.nl*) (Fig. 4) and the frame of a graphical element as well as a white dot (*.com*) for other browsers of the *%WRONG* Browser series I used to see if, in theory, results could be achieved with this tool. But for the *.co.kr* browser nothing at all could be found.

4 Conclusion

The attempt to decompile the executable file concluded my documentation of the *.co.kr* browser. Where the outside-lens on the second level of division provided some information regarding the perceivable elements and the user's options for interacting with them, the underlying structures that were to be explored with the inside-lens remained mostly untouched and therefore could not be documented without the inclusion of the source code. The insights gained when the source code was used in an analysis will be included in a separate text in this volume.

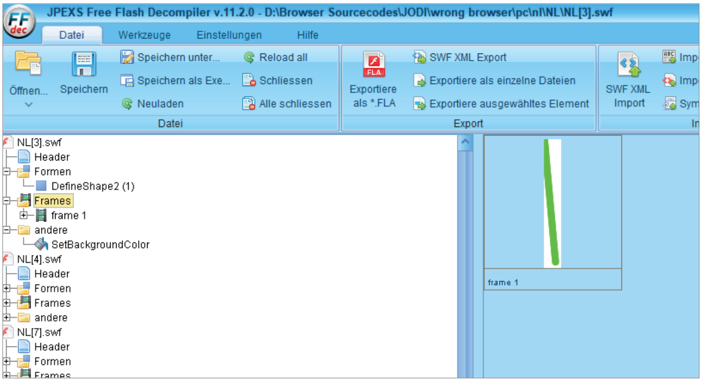
DOI: 10.5282/ubm/epub.93525

Fig.3, Screenshot of the disassembler showing part of the assembler code of the .co.kr.exe.


PE Explorer Disassembler - <D:\Browser Sourcecodes\JODI\wrong browser\pc\cokr\CO.KR.exe>
File
Edit
Search
View
Navigate
Help

```
2000100E                              SUB_L2000100E:
2000100E  55                                          push    ebp
2000100F  8BEC                                        mov     ebp,esp
20001011  81EC94000000                                sub     esp,00000094h
20001017  8D856CFFFFFF                                lea     eax,[ebp-00000094h]
2000101D  C7856CFFFFFF94000000                        mov     dword ptr [ebp-00000094h],00000094h
20001027  50                                          push    eax
20001028  FF15CC800020                                call    [KERNEL32.dll!GetVersionExA]
2000102E  33C0                                        xor     eax,eax
20001030  83BD7CFFFFFF02                              cmp     dword ptr [ebp-00000084h],00000002h
20001037  0F94C0                                      setz    al
2000103A  C9                                          leave
2000103B  C3                                          retn
-------------------------------------------------------------------------------------------
2000103C                              SUB_L2000103C:
2000103C  833D28CE002000                              cmp     dword ptr [L2000CE28],00000000h
20001043  751D                                        jnz     L20001062
20001045  807C240800                                  cmp     byte ptr [esp+08h],00h
2000104A  7409                                        jz      L20001055
2000104C  E830000000                                  call    SUB_L20001081
20001051  85C0                                        test    eax,eax
20001053  7504                                        jnz     L20001059
20001055                              L20001055:
20001055  33C0                                        xor     eax,eax
20001057  EB25                                        jmp     L2000107E
20001059                              L20001059:
20001059  833D28CE002000                              cmp     dword ptr [L2000CE28],00000000h
20001060  7411                                        jz      L20001073
20001062                              L20001062:
20001062  FF3528CE0020                                push    [L2000CE28]
20001068  FF742408                                    push    [esp+08h]
2000106C  E8FF140000                                  call    SUB_L20003270
20001071  59                                          pop     ecx
20001072  59                                          pop     ecx
20001073                              L20001073:
20001073  33C0                                        xor     eax,eax
20001075  390528CE0020                                cmp     [L2000CE28],eax
```

Fig.4, Screenshot of the result of decompiling the .nl %WRONG Browser.



Edited by
Inge Hinterwaldner
Daniela Hönigsberg
Konstantin Mitrokhov

Staatliche Hochschule
für Gestaltung Karlsruhe

DFG-Schwerpunktprogramm ‚Das digitale Bild'
Projekt Browserkunst. Navigieren mit Stil

DAS
DIGITALE
BILD

Erstveröffentlichung: 2022
Gestaltung und Satz: Lydia Kähny und Sophie Ramm
Creative Commons Lizenz:
Namensnennung - Keine Bearbeitung (CC BY-ND)
Diese Publikation wurde finanziert durch die Deutsche Forschungsgemeinschaft.
München, Open Publishing LMU

DFG Deutsche Forschungsgemeinschaft

DOI 10.5282/ubm/epub.93518
ISBN 978-3-487-16315-4
Library of Congress Control Number
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind abrufbar unter http://dnb.dnb.de

Reihe: Begriffe des digitalen Bildes
Reihenherausgeber
Hubertus Kohle
Hubert Locher

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Deutsches Dokumentationszentrum für Kunstgeschichte Bildarchiv Foto Marburg
Philipps Universität Marburg

Das DFG-Schwerpunktprogramm ‚Das digitale Bild' untersucht von einem multiperspektivischen Standpunkt aus die zentrale Rolle, die dem Bild im komplexen Prozess der Digitalisierung des Wissens zukommt. In einem deutschlandweiten Verbund soll dabei eine neue Theorie und Praxis computerbasierter Bildwelten erarbeitet werden.

9 783487 163154

DAS DIGITALE BILD