The Programmer's Brain

WHAT EVERY PROGRAMMER NEEDS TO KNOW ABOUT COGNITION

FELIENNE HERMANS
FOREWORD BY JON SKEET

MANNING
SHELTER ISLAND

contents

foreword *xiii*
preface *xv*
acknowledgments *xvii*
about this book *xix*
about the author *xxii*
about the cover illustration *xxiii*

PART 1 ON READING CODE BETTER 1

1 ***Decoding your confusion while coding*** *3*

1.1 Different kinds of confusion in code 4
Confusion type 1: Lack of knowledge *5* ▪ *Confusion type 2: Lack of information* *5* ▪ *Confusion type 3: Lack of processing power* *6*

1.2 Different cognitive processes that affect coding 6
LTM and programming *7* ▪ *STM and programming* *7*
Working memory and programming *8*

1.3 Cognitive processes in collaboration 9
A brief dissection of how the cognitive processes interacted *9*
Cognitive processes regarding programming tasks *10*

2 ***Speed reading for code 13***

2.1 Quickly reading code 14

What just happened in your brain? 15 ▪ Reexamine your reproduction 16 ▪ Reexamining your second attempt at reproducing code 17 ▪ Why is reading unfamiliar code hard? 18

2.2 Overcoming size limits in your memory 18

The power of chunking 19 ▪ Expert programmers can remember code better than beginners 22

2.3 You see more code than you can read 23

Iconic memory 23 ▪ It's not what you remember; it's the way you remember it 25 ▪ Practice chunking 30

3 ***How to learn programming syntax quickly 33***

3.1 Tips for remembering syntax 34

Disruptions play havoc with your workflow 34

3.2 How to learn syntax quickly with flashcards 35

When to use the flashcards 36 ▪ Expanding the set of flashcards 36 ▪ Thinning the set of flashcards 36

3.3 How to not forget things 37

Why do we forget memories? 38 ▪ Spaced repetition 39

3.4 How to remember syntax longer 40

Two forms of remembering information 40 ▪ Just seeing information is not enough 41 ▪ Remembering information strengthens memories 42 ▪ Strengthen memories by actively thinking 42

4 ***How to read complex code 46***

4.1 Why it's hard to understand complex code 47

What's the difference between working memory and STM? 48
Types of cognitive load as they relate to programming 49

4.2 Techniques to reduce cognitive load 51

Refactoring 51 ▪ Replacing unfamiliar language constructs 52
Code synonyms are great additions to a flashcard deck 55

4.3 Memory aids to use when your working memory is overloaded 56

Creating a dependency graph 56 ▪ Using a state table 59
Combining dependency graphs and state tables 61

PART 2 ON THINKING ABOUT CODE 65

5 *Reaching a deeper understanding of code 67*

5.1 Roles of variables framework 68
Different variables do different things 68 ▪ Eleven roles to cover almost all variables 69

5.2 Roles and paradigms 71
Benefits of roles 72 ▪ Hungarian notation 73

5.3 Gaining a deeper knowledge of programs 75
Text knowledge vs. plan knowledge 75 ▪ Different stages of program understanding 76

5.4 Reading text is similar to reading code 79
What happens in the brain when we read code? 79 ▪ If you can learn French, you can learn Python 81

5.5 Text comprehension strategies applied to code 84
Activating prior knowledge 84 ▪ Monitoring 85 ▪ Determining the importance of different lines of code 86 ▪ Inferring the meaning of variable names 87 ▪ Visualizing 88 ▪ Questioning 89 Summarizing code 90

6 *Getting better at solving programming problems 91*

6.1 Using models to think about code 92
The benefits of using models 92

6.2 Mental models 94
Examining mental models in detail 96 ▪ Learning new mental models 97 ▪ How to use mental models efficiently when thinking about code 97

6.3 Notional machines 102
What is a notional machine? 103 ▪ Examples of notional machines 103 ▪ Different levels of notional machines 105

6.4 Notional machines and language 106
Expanding sets of notional machines 106 ▪ Different notional machines can create conflicting mental models 107

6.5 Notional machines and schemata 108
Why schemata matters 108 ▪ Are notional machines semantics? 109

7 *Misconceptions: Bugs in thinking 110*

7.1 Why learning a second programming language is easier than learning the first one 111

How to increase the chances of benefiting from existing programming knowledge 113 ▪ Different forms of transfer 114 ▪ Already knowing something: Curse or blessing? 115 ▪ The difficulties of transfer 116

7.2 Misconceptions: Bugs in thinking 117

Debugging misconceptions with conceptual change 118 Suppressing misconceptions 119 ▪ Misconceptions about programming languages 120 ▪ Preventing misconceptions while learning a new programming language 122 ▪ Diagnosing misconceptions in a new codebase 122

PART 3 ON WRITING BETTER CODE 125

8 *How to get better at naming things 127*

8.1 Why naming matters 128

Why naming matters 129 ▪ Different perspectives on naming 129 ▪ Initial naming practices have a lasting impact 131

8.2 Cognitive aspects of naming 133

Formatting names supports your STM 133 ▪ Clear names help your LTM 134 ▪ Variable names can contain different types of information to help you understand them 135 ▪ When to evaluate the quality of names 136

8.3 What types of names are easier to understand? 137

To abbreviate or not to abbreviate? 137 ▪ Snake case or camel case? 140

8.4 The influence of names on bugs 141

Code with bad names has more bugs 141

8.5 How to choose better names 142

Name molds 142 ▪ Feitelson's three-step model for better variable names 145

9 *Avoiding bad code and cognitive load: Two frameworks 147*

9.1 Why code with code smells creates a lot of cognitive load 148

A brief intro to code smells 148 ▪ How code smells harm cognition 151

9.2 The influence of bad names on cognitive load 153
Linguistic antipatterns 154 ▪ Measuring cognitive load 155 Linguistic antipatterns and cognitive load 158 ▪ Why linguistic antipatterns cause confusion 159

10 ***Getting better at solving complex problems 160***

10.1 What is problem solving? 161
Elements of problem solving 161 ▪ State space 161

10.2 What is the role of the LTM when you solve programming problems? 162
Is problem solving a cognitive process on its own? 162 ▪ How to teach your LTM to solve problems 164 ▪ Two types of memories that play a role in problem solving 164

10.3 Automatization: Creating implicit memories 167
Implicit memories over time 168 ▪ Why automatization will make you program quicker 170 ▪ Improving implicit memories 171

10.4 Learning from code and its explanation 172
A new type of cognitive load: Germane load 173 ▪ Using worked examples in your working life 175

PART 4 ON COLLABORATING ON CODE 177

11 ***The act of writing code 179***

11.1 Different activities while programming 180
Searching 180 ▪ Comprehension 181 ▪ Transcription 181 Incrementation 181 ▪ Exploration 182 ▪ What about debugging? 182

11.2 Programmer interrupted 183
Programming tasks require a warm-up 183 ▪ What happens after an interruption? 184 ▪ How to better prepare for interruptions 185 When to interrupt a programmer 187 ▪ Some thoughts on multitasking 189

12 ***Designing and improving larger systems 191***

12.1 Examining the properties of codebases 192
Cognitive dimensions 192 ▪ Using CDCB to improve your codebase 200 ▪ Design maneuvers and their trade-offs 201

12.2 Dimensions and activities 202
Impact of dimensions on different activities 202 ▪ Optimizing your codebase for expected activities 204

13 ***How to onboard new developers 205***

13.1 Issues in the onboarding process 206

13.2 Differences between experts and novices 207

Beginners' behavior in more depth 207 ▪ *Difference between seeing concepts concretely and abstractly 211*

13.3 Activities for a better onboarding process 213

Limit tasks to one programming activity 213 ▪ *Support the memory of the onboardee 214* ▪ *Read code together 216*

13.4 Some words to close this book 221

epilogue 221

index 223

foreword

I've spent a lot of my life thinking about programming, and if you're reading this book you probably have too. I haven't spent nearly as much time thinking about thinking, though. The concept of our thought processes and how we interact with code as humans has been important to me, but there has been no scientific study behind it. Let me give you three examples.

I'm the main contributor to a .NET project called Noda Time, providing an alternative set of date and time types to the ones built into .NET. It's been a great environment for me to put time into API design, particularly with respect to naming. Having seen the problems caused by names that make it sound like they change an existing value, but actually return a new value, I've tried to use names that make buggy code sound wrong when you read it. For example, the `LocalDate` type has a `PlusDays` method rather than `AddDays`. I'm hoping that this code looks wrong to most C# developers

```
date.PlusDays(1);
```

whereas this looks more reasonable:

```
tomorrow = today.PlusDays(1);
```

Compare that with the `AddDays` method in the .NET `DateTime` type:

```
date.AddDays(1);
```