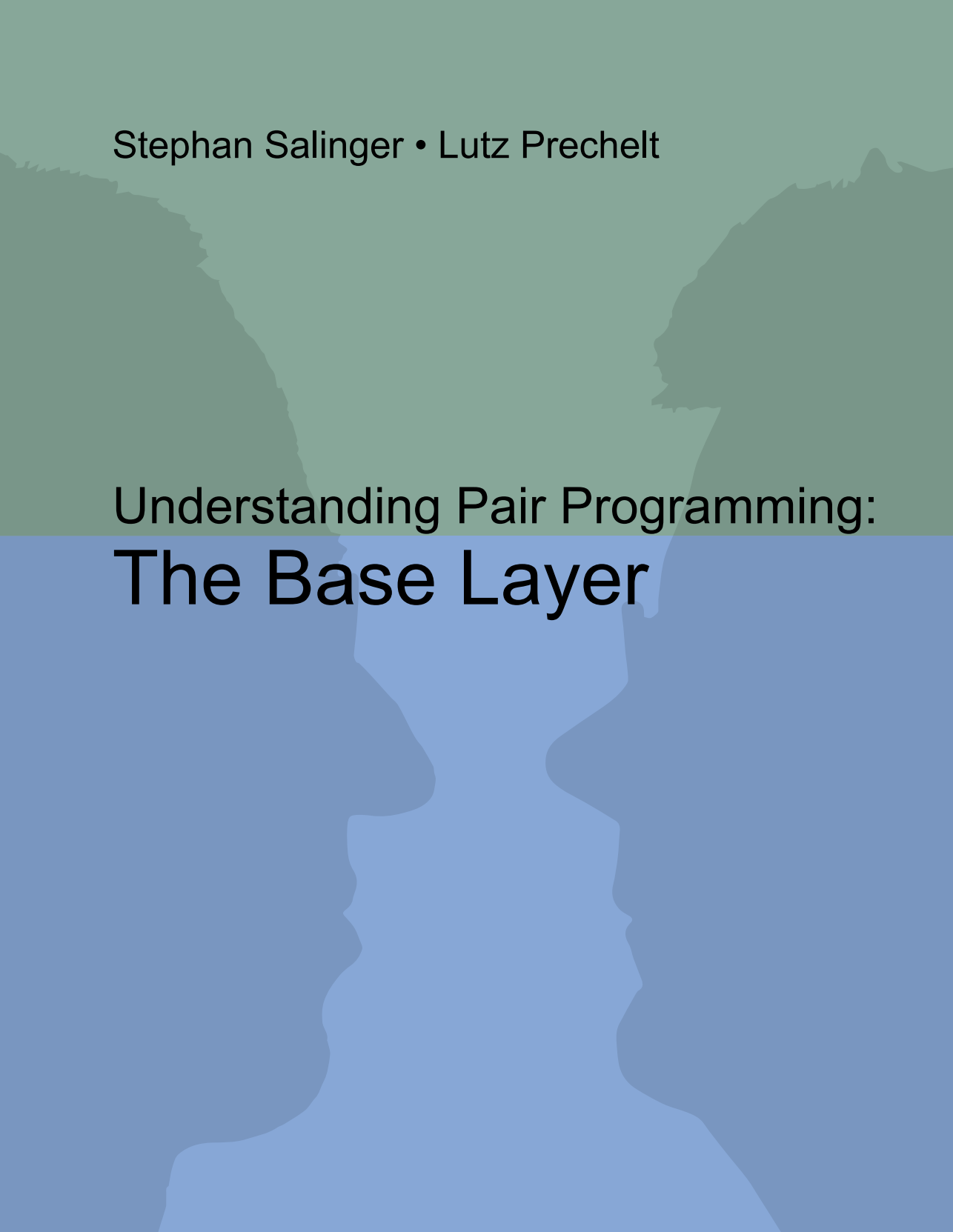


Stephan Salinger • Lutz Prechelt

# Understanding Pair Programming: The Base Layer



The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliographie:

<http://dnb.d-nb.de>

Stephan Salinger, Lutz Prechelt:  
Understanding Pair Programming: The Base Layer

Typeset with LaTeX in Palatino font  
Published and printed by:  
BoD — Books on Demand, Norderstedt, Germany  
[www.bod.de](http://www.bod.de)

ISBN 978-3-7322-8193-0

© Copyright 2013 by Stephan Salinger and Lutz Prechelt

This work is licensed under a Creative Commons  
Attribution–NonCommercial–NoDerivatives 4.0 International License  
CC BY–NC–ND 4.0

<http://creativecommons.org/licenses/by-nc-nd/4.0/>



**Dear reader**, you can get a printed copy of this book at  
<http://www.amazon.co.uk>, but, due to a publishing mishap,  
*not* at <http://www.amazon.com>. We are sorry for any inconvenience!

Stephan Salinger • Lutz Prechelt

# **Understanding Pair Programming: The Base Layer**

Freie Universität Berlin



# Contents

Contents	5
<b>I Introduction</b>	<b>15</b>
<b>1 Introduction</b>	<b>17</b>
1.1 Pair programming	17
1.1.1 What is pair programming?	18
1.1.2 Is pair programming advantageous?	19
1.2 Current understanding of pair programming	19
1.3 The data used for this book	20
1.3.1 Session BA1	21
1.3.2 Session CA2	22
1.3.3 Session ZB7	22
1.4 Our research perspective	22
1.4.1 Basic research perspective: Understanding programming	22
1.4.2 Practitioner perspective: Using pair programming	23
1.4.3 Overall research approach: Work in “layers”	23
1.4.4 Research method: Grounded Theory Methodology	24
1.4.5 On using prior research results	26
1.5 About this book	27
1.5.1 What this book is	27
1.5.2 What this book is not	28
1.5.3 How to read this book	29
1.5.4 How to start performing research based on this book	29
1.6 Terminology and notation	31
<b>2 Overview of the base layer</b>	<b>35</b>
2.1 What are the base concepts?	35
2.1.1 Concepts and concept classes	36
2.1.2 HHI concepts vs. HCI/HEI concepts vs. supplementary concepts	36
2.1.3 HHI concept class groupings	36
2.2 What is the base layer?	37
2.3 Key decisions for the base layer	37
2.3.1 Primarily rely on verbalization	38
2.3.2 Model illocutionary acts	38

2.3.3	Let segmentation emerge . . . . .	39
2.3.4	Crave for behavioristic interpretation . . . . .	39
2.3.5	Model the discourse world, not the activity world . . . .	40
2.3.6	Model dialog episodes . . . . .	40
2.3.7	Design the concepts to reflect relevant phenomena . . . .	41
<b>II</b>	<b>The HHI concepts:</b>	
	<b>Human/human interaction</b>	<b>43</b>
<b>3</b>	<b>Objects and verbs of the HHI concepts</b>	<b>45</b>
3.1	The structure and meaning of concept names . . . . .	45
3.2	The objects . . . . .	46
3.3	The verbs . . . . .	47
3.4	The existing object/verb combinations . . . . .	48
3.5	Types of verbs . . . . .	49
3.6	The notion of “knowledge” . . . . .	52
3.7	<i>propose</i> vs. <i>explain</i> . . . . .	56
3.8	<i>explain</i> vs. <i>think aloud</i> . . . . .	56
3.9	<i>disagree+propose</i> vs. <i>challenge</i> . . . . .	57
<b>4</b>	<b>Product-oriented concepts: <i>design</i></b>	<b>59</b>
4.1	Topic of <i>design</i> concepts . . . . .	59
4.2	<i>design</i> concepts and their properties . . . . .	60
4.2.1	Types and intentions of proposals . . . . .	62
4.2.2	Referring to editing steps . . . . .	63
4.2.3	Proposals with rationale . . . . .	64
4.2.4	<i>decide</i> vs. <i>agree</i> . . . . .	64
4.2.5	<i>amend</i> vs. a new <i>propose</i> . . . . .	64
4.2.6	<i>amend</i> or <i>challenge</i> one’s own proposal . . . . .	64
4.2.7	Indicating agreement vs. indicating attentiveness . . . .	65
4.2.8	Short negations . . . . .	65
4.2.9	Proposal-less questions . . . . .	65
4.2.10	Restricted disagreement . . . . .	66
4.3	Discrimination from similar concepts . . . . .	66
4.3.1	<i>propose_design</i> vs. <i>ask_knowledge</i> . . . . .	66
4.3.2	* <i>_design</i> vs. <i>explain_knowledge/explain_finding</i> . . . . .	66
4.3.3	<i>propose_design</i> vs. <i>propose_step/propose_todo</i> . . . . .	67
<b>5</b>	<b>Product-oriented concepts: <i>requirement</i></b>	<b>69</b>
5.1	Topic of <i>requirement</i> concepts . . . . .	69
5.2	<i>requirement</i> concepts and their properties . . . . .	70
5.2.1	<i>remember_requirement</i> . . . . .	71
5.2.2	<i>propose_requirement</i> . . . . .	71

5.2.3	<i>agree_requirement</i> and <i>challenge_requirement</i> . . . . .	71
5.3	Discrimination from similar concepts . . . . .	72
<b>6</b>	<b>Process-oriented concepts: <i>step</i></b>	<b>73</b>
6.1	Topic of <i>step</i> concepts . . . . .	73
6.2	<i>step</i> concepts and their properties . . . . .	75
6.2.1	<i>propose_step</i> with rationale . . . . .	75
6.2.2	Purpose of making <i>propose_step</i> utterances . . . . .	76
6.2.3	Reserving time . . . . .	77
6.2.4	Imprecise proposals . . . . .	77
6.2.5	<i>decide_step</i> vs. <i>agree_step</i> . . . . .	78
6.2.6	<i>amend</i> , <i>challenge</i> , or <i>disagree</i> one's own proposal . . . . .	78
6.2.7	Indicating agreement vs. indicating attentiveness . . . . .	79
6.2.8	<i>ask_step</i> . . . . .	79
6.3	Discrimination from similar concepts . . . . .	79
6.3.1	* <i>_step</i> vs. <i>explain_knowledge/explain_finding</i> . . . . .	79
6.3.2	<i>propose_step</i> vs. <i>propose_design</i> . . . . .	79
6.3.3	<i>propose_step</i> vs. <i>ask_knowledge</i> . . . . .	81
6.3.4	<i>ask_step</i> vs. <i>ask_knowledge</i> . . . . .	82
6.3.5	<i>ask_step</i> vs. <i>ask_design</i> . . . . .	82
6.3.6	<i>disagree_step</i> vs. <i>explain_knowledge/explain_finding</i> . . . . .	82
6.3.7	<i>amend_step</i> vs. <i>explain_knowledge/explain_finding</i> . . . . .	82
<b>7</b>	<b>Process-oriented concepts: <i>completion</i></b>	<b>83</b>
7.1	Topic of <i>completion</i> concepts . . . . .	83
7.2	<i>completion</i> concepts and their properties . . . . .	84
7.2.1	Short evaluations . . . . .	84
7.2.2	Indirect evaluations . . . . .	85
7.2.3	Evaluation of quality . . . . .	85
7.3	Discrimination from similar concepts . . . . .	85
<b>8</b>	<b>Process-oriented concepts: <i>todo</i></b>	<b>87</b>
8.1	Topic of <i>todo</i> concepts . . . . .	87
8.2	<i>todo</i> concepts and their properties . . . . .	88
8.3	Discrimination from similar concepts . . . . .	89
8.3.1	<i>propose_todo</i> vs. <i>propose_step</i> . . . . .	89
8.3.2	<i>propose_todo</i> vs. <i>explain_knowledge/explain_finding</i> . . . . .	90
8.3.3	<i>propose_todo</i> vs. <i>amend_design/propose_design</i> . . . . .	90
<b>9</b>	<b>Process-oriented concepts: <i>strategy</i></b>	<b>93</b>
9.1	Topic and typology of <i>strategy</i> concepts . . . . .	93
9.1.1	OWP: Organizing Work Packages . . . . .	93
9.1.2	DPR: Determining Procedure Rules . . . . .	93
9.1.3	EXS: Expanding a <i>step</i> into a <i>strategy</i> . . . . .	94

9.1.4	Extensional vs. intensional representation . . . . .	94
9.1.5	Range . . . . .	94
9.1.6	Mixed types . . . . .	95
9.2	<i>strategy</i> concepts and their properties . . . . .	96
9.2.1	Proposal mode . . . . .	96
9.2.2	Proposals with alternatives . . . . .	96
9.2.3	<i>decide_strategy</i> vs. <i>agree_strategy</i> . . . . .	96
9.2.4	Secondary issues . . . . .	96
9.2.5	Forms of <i>amend_strategy</i> . . . . .	97
9.2.6	Distinguishing proposals: <i>amend</i> , <i>challenge</i> , <i>propose</i> . . . . .	98
9.2.7	<i>ask_strategy</i> . . . . .	99
9.2.8	<i>agree_strategy</i> . . . . .	99
9.2.9	<i>disagree_strategy</i> . . . . .	99
9.3	Discrimination from similar concepts . . . . .	99
9.3.1	<i>*_strategy</i> vs. <i>explain_knowledge/explain_finding</i> . . . . .	99
9.3.2	<i>propose_strategy</i> vs. <i>propose_todo</i> . . . . .	99
9.3.3	<i>propose_strategy</i> vs. <i>propose_step</i> . . . . .	100
9.3.3.1	Recycled strategies . . . . .	100
9.3.3.2	<i>step</i> with forward reference . . . . .	100
9.3.3.3	<i>steps</i> aiming at advantage . . . . .	100
9.3.3.4	Multi-part proposals not forming a strategy . . . . .	101
9.3.3.5	The creative act is invisible . . . . .	101
9.3.3.6	Lowly creative acts . . . . .	101
9.3.4	<i>propose_strategy</i> vs. <i>propose_design</i> . . . . .	101
9.3.5	<i>agree_strategy</i> vs. <i>agree_knowledge</i> . . . . .	102
9.3.6	<i>ask_strategy</i> vs. <i>ask_knowledge</i> . . . . .	103
9.3.7	<i>ask_strategy</i> vs. <i>ask_step</i> . . . . .	103
<b>10</b>	<b>Process-oriented concepts: <i>state</i></b>	<b>105</b>
10.1	Topic of <i>state</i> concepts . . . . .	105
10.2	<i>state</i> concepts and their properties . . . . .	106
10.2.1	Short <i>agree</i> utterances . . . . .	106
10.2.2	Lack of reference to a <i>strategy</i> . . . . .	106
10.2.3	Partial disagreement . . . . .	107
10.3	Discrimination from similar concepts . . . . .	107
10.3.1	<i>explain_state</i> vs. <i>explain_completion</i> . . . . .	107
10.3.2	<i>explain_state</i> vs. <i>explain_finding</i> . . . . .	107
<b>11</b>	<b>Universal concepts: What is “knowledge”?</b>	<b>109</b>
11.1	On knowledge . . . . .	109
11.2	The base concepts’ notion of knowledge . . . . .	110
11.3	Priority rules for assigning knowledge concepts . . . . .	111
<b>12</b>	<b>Universal concepts: <i>finding</i></b>	<b>113</b>



12.1	Topic and typology of <i>finding</i> concepts . . . . .	113
12.1.1	<i>finding</i> type P: perceived event . . . . .	114
12.1.2	<i>finding</i> type D: discovered issue . . . . .	114
12.1.3	<i>finding</i> type T: thought . . . . .	116
12.1.4	Priority rules for checking <i>finding</i> types . . . . .	116
12.1.5	<i>finding</i> type indicators and examples . . . . .	117
12.2	<i>finding</i> concepts and their properties . . . . .	120
12.2.1	Aggregation of utterances . . . . .	120
12.2.2	Repeated statements . . . . .	121
12.2.3	Thinking aloud . . . . .	121
12.2.4	Revoking and replacing <i>findings</i> . . . . .	122
12.2.5	“Additional” findings . . . . .	122
12.2.6	Justifications of proposals . . . . .	125
12.2.7	Justifications of <i>findings</i> . . . . .	125
12.2.8	<i>disagree_finding</i> , <i>challenge_finding</i> . . . . .	125
12.2.9	Reasons for agreement . . . . .	127
12.2.10	Doubt . . . . .	128
12.2.11	<i>ask_finding?</i> . . . . .	128
12.3	Discrimination from similar concepts . . . . .	129
12.3.1	<i>finding</i> vs. other universal concepts . . . . .	130
12.3.2	<i>explain_finding</i> vs. <i>propose_design</i> . . . . .	130
12.3.3	<i>explain_finding</i> vs. <i>*_step</i> . . . . .	130
12.3.4	<i>explain_finding</i> vs. <i>explain_completion</i> or <i>explain_state</i> . . .	130
<b>13</b>	<b>Universal concepts: <i>hypothesis</i></b>	<b>131</b>
13.1	Topic of <i>hypothesis</i> concepts . . . . .	131
13.1.1	Uncertain knowledge . . . . .	131
13.1.2	Hard-to-verify assumptions . . . . .	131
13.1.3	Readily verifiable conjectures . . . . .	132
13.1.4	Issue types addressed by hypotheses . . . . .	132
13.2	<i>hypothesis</i> concepts and their properties . . . . .	133
13.2.1	<i>propose_hypothesis</i> . . . . .	133
13.2.2	<i>agree_hypothesis</i> , <i>disagree_hypothesis</i> , <i>challenge_hypothesis</i> .	133
13.2.3	Conditional agreement . . . . .	134
13.2.4	Revoking or replacing one’s own hypothesis . . . . .	134
13.2.5	<i>amend_hypothesis</i> : One hypothesis or several? . . . . .	134
13.2.6	Justification of hypotheses . . . . .	136
13.2.7	Justification by hypotheses . . . . .	136
13.3	Discrimination from similar concepts . . . . .	136
13.3.1	<i>propose_hypothesis</i> vs. <i>explain_finding</i> . . . . .	136
<b>14</b>	<b>Universal concepts: <i>standard of knowledge</i></b>	<b>139</b>
14.1	Topic of <i>standard of knowledge</i> concepts . . . . .	139
14.1.1	PT: Preparing knowledge transfer . . . . .	139

14.1.2	RT: Refusing knowledge transfer . . . . .	140
14.1.3	AT: Acknowledging knowledge transfer . . . . .	140
14.2	<i>standard of knowledge</i> concepts and their properties . . . . .	141
14.2.1	<i>ask_standard of knowledge</i> . . . . .	141
14.2.2	AT with paraphrasing . . . . .	141
14.2.3	<i>standard of knowledge</i> in the making . . . . .	142
14.2.4	<i>explain_standard of knowledge</i> may be findings . . . . .	143
14.2.5	Limited-knowledge proposals . . . . .	143
14.2.6	Implicit statements . . . . .	143
14.2.7	Backward-looking statements . . . . .	143
14.2.8	Signaling ongoing thinking . . . . .	144
14.3	Discrimination from similar concepts . . . . .	144
14.3.1	<i>explain_standard of knowledge</i> vs. <i>ask_knowledge</i> . . . . .	144
14.3.2	<i>explain_standard of knowledge</i> vs. <i>agree_finding</i> or <i>disagree_finding</i> . . . . .	144
14.3.3	<i>explain_standard of knowledge</i> vs. <i>explain_finding</i> . . . . .	145
14.3.4	<i>explain_standard of knowledge</i> vs. <i>agree/disagree</i> for a proposal . . . . .	145
14.3.5	<i>explain_standard of knowledge</i> vs. <i>propose_hypothesis</i> . . . . .	145
<b>15</b>	<b>Universal concepts: <i>gap in knowledge</i></b> . . . . .	<b>147</b>
15.1	Topic of <i>gap in knowledge</i> concepts . . . . .	147
15.2	<i>gap in knowledge</i> concepts and their properties . . . . .	148
15.2.1	<i>explain_gap in knowledge</i> . . . . .	148
15.3	Discrimination from similar concepts . . . . .	148
15.3.1	<i>explain_gap in knowledge</i> vs. <i>explain_standard of knowledge</i> . . . . .	148
15.3.2	<i>agree_gap in knowledge</i> vs. <i>agree_standard of knowledge</i> . . . . .	148
15.3.3	<i>explain_gap in knowledge</i> vs. <i>propose_step</i> . . . . .	149
<b>16</b>	<b>Universal concepts: <i>knowledge</i></b> . . . . .	<b>151</b>
16.1	Topic of <i>knowledge</i> concepts . . . . .	151
16.2	<i>knowledge</i> concepts and their properties . . . . .	153
16.2.1	Evaluations and judgments . . . . .	153
16.2.2	Unprompted knowledge transfer . . . . .	154
16.2.3	Rhetorical questions . . . . .	154
16.2.4	Aggregation of utterances . . . . .	154
16.2.5	<i>amend_knowledge?</i> . . . . .	154
16.2.6	“Different” answers . . . . .	154
16.2.7	Modes of agreement . . . . .	155
16.2.8	Indicating agreement vs. indicating attentiveness . . . . .	157
16.2.9	Opposition and controversy . . . . .	157
16.2.10	Disagreeing by agreeing to the opposite . . . . .	158
16.2.11	Opinions . . . . .	158
16.2.12	Limited conviction . . . . .	158

16.2.13	<i>ask_knowledge</i> is not always that . . . . .	158
16.2.14	Questions including possible answers . . . . .	158
16.2.15	Statement or question? . . . . .	158
16.3	Discrimination from similar concepts . . . . .	159
16.3.1	<i>explain_knowledge</i> vs. <i>propose_step</i> . . . . .	159
16.3.2	<i>explain_knowledge</i> vs. <i>propose_design</i> . . . . .	159
16.3.3	<i>explain_knowledge</i> vs. <i>explain_finding</i> . . . . .	160
16.3.4	<i>explain_knowledge</i> vs. <i>amend_finding</i> , <i>challenge_finding</i> , <i>dis-</i> <i>agree_finding</i> . . . . .	161
16.3.5	<i>explain_knowledge</i> vs. <i>agree_design/disagree_design</i> . . . . .	161
16.3.6	<i>ask_knowledge</i> vs. <i>propose_hypothesis</i> . . . . .	161
16.3.7	<i>ask_knowledge</i> vs. <i>explain_finding</i> . . . . .	162
16.3.8	<i>explain_knowledge</i> vs. <i>explain_standard of knowledge</i> . . . . .	162
16.3.9	<i>ask_knowledge</i> vs. <i>explain_standard of knowledge</i> . . . . .	163
16.3.10	<i>agree_knowledge</i> vs. <i>explain_standard of knowledge</i> . . . . .	163
<b>17</b>	<b>Universal concepts: activity</b>	<b>165</b>
17.1	The notion of facade concept class . . . . .	165
17.2	Topic of <i>activity</i> concepts . . . . .	165
17.3	<i>activity</i> concepts and their properties . . . . .	169
17.3.1	Granularity of <i>think aloud_activity</i> . . . . .	169
17.3.2	<i>think aloud_activity</i> phenomena leading to questions . . . . .	170
17.3.3	HCI/HEI activities resulting from an utterance . . . . .	170
17.3.4	Disconnect of HCI/HEI activity and verbalization . . . . .	171
17.3.5	The partner commenting on activity vs. on verbalizations . . . . .	171
17.3.6	<i>challenge_activity</i> . . . . .	171
17.3.7	<i>agree_activity</i> , <i>disagree_activity</i> . . . . .	172
17.3.8	Comments before the fact . . . . .	173
17.3.9	Comments after the end . . . . .	173
17.3.10	<i>amend_activity</i> vs. <i>challenge_activity</i> . . . . .	173
17.3.11	<i>stop_activity</i> . . . . .	174
17.3.12	Interjections leading to activity change . . . . .	174
17.3.13	<i>think aloud_activity</i> by the “observer” . . . . .	174
17.3.14	Self-criticism . . . . .	174
<b>18</b>	<b>Universal concepts: Miscellaneous</b>	<b>177</b>
18.1	<i>mumble_sth</i> . . . . .	177
18.2	<i>say_off topic</i> . . . . .	177
<b>III</b>	<b>Other concepts</b>	<b>179</b>
<b>19</b>	<b>The HCI/HEI concepts</b>	<b>181</b>
19.1	<i>write_sth</i> . . . . .	182

19.2	<i>search_sth</i> . . . . .	183
19.3	<i>explore_sth</i> . . . . .	185
19.4	<i>verify_sth</i> . . . . .	186
19.5	<i>read_sth</i> . . . . .	189
19.6	<i>sketch_sth</i> . . . . .	189
19.7	<i>show_sth</i> . . . . .	189
19.8	<i>do_sth</i> . . . . .	190
19.9	On drivers, observers, and co-action . . . . .	190
<b>20</b>	<b>Supplementary concepts</b>	<b>193</b>
20.1	<i>become_driver</i> . . . . .	193
20.2	<i>work in parallel_sth</i> . . . . .	193
20.3	<i>work alone_sth</i> . . . . .	193
20.4	<i>wait for_sth</i> . . . . .	194
20.5	<i>react to_interrupt</i> . . . . .	194
<b>IV</b>	<b>Using the base concepts</b>	<b>195</b>
<b>21</b>	<b>Guidelines for annotating</b>	<b>197</b>
21.1	How to pick appropriate HHI concepts . . . . .	197
21.2	How to pick appropriate HCI/HEI concepts . . . . .	198
21.3	What to consider as context . . . . .	199
21.4	When to use double HHI annotations . . . . .	199
21.5	How to segment utterances . . . . .	200
21.6	How to handle specific phenomena . . . . .	201
21.6.1	How to annotate implicit announcements . . . . .	201
21.6.2	How to annotate thematic shifts . . . . .	201
21.6.3	How to annotate repetitions . . . . .	203
21.6.4	How to annotate incomplete agreement or disagreement . . . . .	204
21.6.5	How to annotate self-corrections . . . . .	204
21.6.6	How to annotate justifications . . . . .	204
21.7	Method hints . . . . .	205
21.7.1	Step back . . . . .	205
21.7.2	Paraphrase . . . . .	205
21.7.3	Peek into the future . . . . .	206
<b>22</b>	<b>Guidelines for modifying the base concept set</b>	<b>207</b>
22.1	What makes a good concept set . . . . .	207
22.2	When to shift a boundary . . . . .	208
22.3	When to add a new property value . . . . .	208
22.4	When to add a concept and which . . . . .	209
<b>23</b>	<b>Guidelines for creating new concept sets</b>	<b>211</b>

<i>Contents</i>	13
23.1 The idea of layers . . . . .	211
23.2 Granularity . . . . .	212
23.3 Properties and property values . . . . .	212
23.4 Forming “nice” layers . . . . .	213
23.5 Go! . . . . .	213
<b>Bibliography</b>	<b>215</b>
<b>Index</b>	<b>217</b>

**Note**

The PDF version of this book contains very many cross-reference hyperlinks. It may be convenient to use the paper version for learning but then the PDF version for actually working with the base layer.

**Acknowledgments**

Sincere thanks to Laura Plonka for collecting a large part of our session recordings and for working closely with Stephan in the early stage of our analysis, to Franz Zieris for the first serious third-party use of the base layer, to Franz Zieris, David Socha, and Helen Sharp for their feedback on the book draft, to Gesine Milde for proofreading, and to all the pairs that agreed to be recorded and scrutinized.

## **Part I**

# **Introduction**

... in which we explain what this book is all about, how to best use it, and what notation we will use for the examples.





# Introduction

This book is a handbook for researchers attempting to make sense of what is going on in pair programming sessions; it is based on Stephan Salinger’s Ph.D. dissertation [11]. The present chapter will introduce pair programming (in Section 1.1), summarize what research has so far found out about it (Section 1.2), explain the the raw data we have used (Section 1.3) and the research approach we propose (Section 1.4), propose how to make use of the book (Section 1.5), and introduce a few key terms and notations (Section 1.6).

## 1.1 Pair programming

Assume you have a Ph.D. in dancing science and are the only non-programmer at a party full of programmers. According to the stereotype, it is hard to talk to these people. Your best bet would be to grab two or three of them at once and ask

*“Is pair programming a good engineering practice?”*

The ensuing discussion will be lively and despite talking to techies you can have your part in the discussion!

Pair programming is a subtle matter and so any good answer to the question ought to begin with “Well...”, but (and that is what makes the discussion so lively) many people appear to have a simplified notion of it and a correspondingly clear opinion.

Why is that so? And what, exactly, *is* pair programming anyway?

### 1.1.1 What is pair programming?

Pair programming is an old technique. Fred Brooks (of Mythical Man-Month fame) reports: “*Fellow graduate student Bill Wright and I first tried pair programming when I was a grad student (1953–56). We produced 1500 lines of defect-free code; it ran correctly first try.*” [15, p.8]. Its modern popularity is largely due to Kent Beck’s 1999 book on eXtreme Programming (XP) [2], a holistic method for small-team software development consisting of twelve practices, a core one of which is pair programming. In the section on pair programming, Beck states “*Pair programming really deserves its own book. It’s a subtle skill*” [2, p.100], and indeed such a book appeared in 2002: “*Pair Programming Illuminated*”. It offers the following characterization:

*“Pair programming is a style of programming in which two programmers work side by side at one computer, continually collaborating on the same design, algorithm, code, or test. One of the pair, called the driver, is typing at the computer or writing down a design. The other partner, called the navigator, has many jobs, one of which is to observe the work of the driver, looking for tactical and strategic defects.”* [15, p.3]

Note that more than half of this definition is concerned with describing the roles of driver and navigator (the latter is now more (Google-)commonly called “observer”). But once you have read this book (or any substantial part of it), you will know that while the first part of the definition is alright, the second part is misleading: The description of both roles is wrong in many respects and the whole driver/observer distinction does not go far in characterizing the pair programming process anyway.<sup>1</sup>

Kent Beck’s description is shorter: “*Pair programming—All production code is written with two programmers at one machine.*” [2, p.54]. There is elaboration later, but this is arguably his definition of this all-important practice. The 2004 second edition of the book is more explicit:

*“Write all production programs with two people sitting at one machine. Set up the machine so the partners can sit comfortably side-by-side. Move the keyboard and mouse back and forth so you are comfortable while you are typing. Pair programming is a dialog between two people simultaneously programming (and analyzing and designing and testing) and trying to program better.”* [3, p.26]

“Sitting comfortably” sounds like trivial information compared to the presumably illuminating driver/observer characterization, but it is relevant. And once

---

<sup>1</sup>We will occasionally use the terms driver and observer anyway (and without a particular connotation) when they are handy to express something.

you have read the present book, you will appreciate that the above definition captures, very inconspicuously, a key property of pair programming: *“Pair programming is a dialog”*. Yes!

At this point, we have nothing to add to that.

### 1.1.2 Is pair programming advantageous?

This leaves the other question: Why do some people have such a simplified (and then strong) notion of whether pair programming is a good engineering practice? The strongest ones tend to be the strict opponents: their attitude is usually the belief that the obvious cost of pair programming (occupying two precious software developers rather than just one) is so large that no corresponding benefits can possibly outweigh it.

More thoughtful discussants will not readily agree because the list of potential benefits is impressive. Here is (in paraphrased form) the one presented in *“Pair Programming Illuminated”* [15, p.4]:

- The resulting code may contain fewer defects.
- The pair will likely finish faster than an individual would.<sup>2</sup>
- *“Pair programmers are happier programmers.”*
- Pair programming builds within-team trust and improves teamwork.
- Unless you use fixed pairs only, a developer will become acquainted with larger fractions of the overall code, design, and requirements.
- Pair partners learn from each other.

How much do we know about which of these are true and to what degree? Not much.

## 1.2 Current understanding of pair programming

Since the pioneering study of Nosek (which appeared even before Kent Beck’s book) in 1998 [10] there have been many empirical studies on pair programming, in particular controlled experiments comparing it to solo programming, but the amount of knowledge produced by these studies is not large; an overview of research until 2007 is provided by Hannay et al. [8]. We do not aim at a detailed overview here. Roughly speaking, there is good evidence that pairs tend to be faster than solo programmers, some evidence that their

---

<sup>2</sup>The book claims “in about half the time”, but that is oversimplifying matters; see Section 1.2.

work tends to have fewer defects, and beginning evidence that the designs produced are better. The size of each of these effects, however, is hardly understood: The results of individual studies differ so much (and those differences remain unexplained) that taken together the results are inconclusive.

What is worse, their validity is highly questionable as the conditions under which most of them were created are highly unrealistic: mostly non-professional programmers, normally non-gelled pairings, usually either development from scratch or work on fairly small programs, generally little or no relevance of domain knowledge. Even the most ambitious of the controlled experiments, which hired 295 professionals for one day, concluded: *“It is possible that the benefits of pair programming will exceed the results obtained in this experiment for larger, more complex tasks and if the pair programmers have a chance to work together over a longer period of time.”* [1]. This statement is also one of the few exceptions of the disturbing tendency that most studies tacitly assume there is no such thing as a specific pair programming skill distinct from general software development skill. We believe that this assumption is wrong and that successful pair programming research needs to reflect that. This implies a lot of qualitative pair programming research will be required before meaningful designs for quantitative pair programming studies can even be formulated.

For such qualitative types of research questions, the amount of work done so far is much smaller<sup>3</sup> although the number of questions is larger: There is evidence that different capability levels of the pair members play a role [5] and some evidence that personality characteristics of the pair members may play a modest role, too [9]. Only few studies discuss high-level behaviors or mechanisms and those do not do much decomposition or analysis yet, e.g. [6], or are even based on anecdotal evidence only, e.g. [16].

In our view, the most conclusive of the qualitative studies showed that the description of the driver and navigator roles from the above definition does *not* represent reality: Rather than working on different levels of abstraction (low and high for the navigator versus medium for the driver) as the definition assumes, the partners in fact strongly tend to move through these abstraction levels together [4, 6]. Work towards a more meaningful roles model is still in its infancy [13].

### 1.3 The data used for this book

The results and all examples presented in this book are based on complete recordings of individual pair programming sessions. The recordings consist of audio, a pixel-precise recording of all screen activity, and a webcam recording of the pair (usually recorded from atop the monitor). We use Techsmith Camtasia

---

<sup>3</sup>We ignore surveys here, because surveys observe attitudes only, not actual practice.

Studio<sup>4</sup> for recording and place the webcam video into the lower-right corner of the screen video. See [12] for a few more details.

We possess a substantial collection of such recordings of typically one to three hours length. 55 recordings stem from pairs of 48 different volunteer industrial software developers (called A1 to K4, see session descriptions below) doing their normal work in their usual environment (domain, code, task, tools, hardware, office, etc.) in one of 11 different companies (called A to K). The reality distortion of these videos is presumably negligible; the pairs do not show (nor report when interviewed afterwards) any acute awareness of being recorded beyond a minute into their work. The videos reflect a variety of domains, developer constellations, and task types; most tasks can be subsumed under extension programming. They reflect only small cultural variety, though: All sessions are from German companies and involve German-speaking developers. See the note on translation in Section 1.6.

Further 28 recordings stem from pairs of 56 different volunteer graduate students (called Z1 to Z56) working in one of 5 different controlled laboratory settings (called ZA to ZE). The advantage of these recordings is that the researcher has a good understanding of the code base, the task, and correct solutions for the task, making it often much easier to understand what is really going on in the session.

Only 7 of these recordings (6 professional/industrial and 1 student/laboratory) were used for the research reflected here. For the concepts reported here, we reached theoretical saturation (see Section 1.4.4) with only this many.<sup>5</sup> For the examples presented in this book, we even confine ourselves to only three of these sessions, so that over time you can get better acquainted with their respective topics; some of the examples are even understandably related. These three sessions are the following:

### 1.3.1 Session BA1

An industrial session (with a duration of 1:47 hours) of two professional programmers B1 and B2 who worked for a large community portal operator B and had paired several times before. They built an extension to the community portal, which is implemented in PHP. The task difficulty had several aspects

---

<sup>4</sup><http://www.techsmith.com/>

<sup>5</sup>Actually, most subsets of any random three of them would have sufficed. This will be different in subsequent research when investigating more specialized pair programming phenomena which do not occur frequently, yet still have many facets and variations that need to be understood. For our purposes here, however, attributes such as the level of programming skill, the amount of pair programming experience, the exact nature of the task play, and many others are of minor relevance, as we are concerned only with uncovering phenomena, not with determining their frequency or interplay.

including understanding the design and design rationale of the pre-existing code, which had been written by nearshore programmers.

### 1.3.2 Session CA2

An industrial session (duration 1:16 hours) of two professional programmers C2 and C5 who worked for a software product company C. The product they work on is a geographic information system (GIS) desktop GUI application written in Java. The design of this software uses abstraction elaborately; the task involves a small functional extension and its main difficulty lies in understanding and properly applying the existing design abstractions.

### 1.3.3 Session ZB7

A laboratory session (duration 2:58 hours) of two graduate students Z19 and Z20 who had worked together as a pair several times before. They built a small extension to a cleanly designed Java EE web shop system with which they were modestly familiar. The main task difficulty lay in the need to apply certain Java EE technologies (JMS, JNDI, JBoss application server) that the developers had learned about in a recent graduate course but had not applied often beforehand.

## 1.4 Our research perspective

The purpose of this book is to lay the groundwork for a stream of research aiming at thoroughly understanding pair programming. We will now explain why we believe this is relevant from the perspective of basic software engineering research (Section 1.4.1) as well as from a practitioner perspective (1.4.2), what the overall architecture of this research will look like (1.4.3), which specific research method we suggest to primarily use (1.4.4) and what the benefits are with respect to science's principle of knowledge accumulation ("standing on the shoulders of giants", Section 1.4.5).

### 1.4.1 Basic research perspective: Understanding programming

Several decades after research began that attempted to understand what is going on in the activity we call "programming", this understanding is still very much in its infancy. Pair programming provides a wonderful opportunity for making a lot of progress there, because rather than having to rely on artificial think-aloud data gathering techniques, pair programmers verbalize naturally much of the time.

Pair programming will surely be different from solo programming in many respects, but probably also fundamentally similar. And while think-aloud

studies may occasionally be possible even in industrial work contexts, they tend to be difficult to arrange. In comparison, pair programming data can be gathered more easily and almost unobtrusively in industrial work contexts on real work tasks; see Section 1.3. This whole basic research aspect, however, is more a fringe benefit, not the core reason why we started this line of work.

### 1.4.2 Practitioner perspective: Using pair programming

Our overall research goal is to understand the mechanisms of pair programming sufficiently well to provide practitioners with detailed advice regarding (a) in which situations to use pair programming and (b) how pair members might behave to make pair programming effective, smooth, and efficient.

The basic idea for achieving this is to understand many sub-behaviors at work within pair programming and formulate this understanding into one or more patterns or antipatterns of behavior for each. This research will be almost purely qualitative; better quantitative research can then be started based on this differentiated and advanced understanding.

### 1.4.3 Overall research approach: Work in “layers”

The goals described in Sections 1.4.1 and 1.4.2 are far too ambitious for a single research project; the work needs to be modularized somehow. This, however, will not be easy: Initially, many fundamentals need to be understood before even the first few useful patterns will emerge. Later on, of the various topics studied, many will be interdependent or at least layered on top of each other.

Our overall approach is therefore to first lay a foundation of elementary concepts useful for analyzing and understanding pair programming sessions. This is what the current book is about. We call this foundation the *base layer*. It consists of a set of *base concepts* (surprisingly called the *base concept set* and introduced in Chapters 3 to 20) and rules for its use (Chapter 21) and extension (Chapter 22).

On top of this foundation, a subsequent study of some pair programming topic X (such as “decision-making”) can then build an X-layer of concepts that together characterize X. While working on the X-layer, the study can make use of the base layer and of the concepts found in subsequent studies performed earlier on other topics A, B, C (say, “pair programming roles” and others). If, for understanding X, some other topic Y (say, “knowledge transfer”) is relevant, the study on X will obtain a minimal understanding of Y required internally but needs not work it out fully.

Once the study of Y has been performed later (which may also use the X-layer fully), the X-layer can be consolidated into also using the Y-layer. This will break the layering for the overall results (pair programming is a holistic

activity after all!), but still keeps a convenient mostly-layered work style for the individual sub-studies.

Each such study may provide a number of behavioral patterns and antipatterns. The role of the base layer is special because it provides common terminology that not only jumpstarts but also connects the other studies such as to form a whole rather than a set of separate pieces. The number of concepts in the base layer is sufficiently small to allow the various researchers to stay on top of them, so there are good chances of actual (near-)consistency between studies even of different researchers rather than only formal pseudo-consistency.

#### 1.4.4 Research method: Grounded Theory Methodology

When we started with this work, we felt that many of the common statements made about pair programming were likely misleading or at least naive, but we had no expectation of what a better characterization would be like. We shared Kent Beck's view that pair programming is "*a subtle skill*". So once we had made the decision to analyze session recordings such as those described in Section 1.3, we had no idea which aspects of them would be relevant: The dialog content? Its wording? Phrasing? Intonation? Screen content? Changes of screen content? Human activity on the computer? Gestures? Facial expressions? The list went on and on. We quickly decided it would be important to pick a research method that was as empty of assumptions as possible.

Ethnographical approaches are rather far away from software engineering thinking, so we decided for Grounded Theory Methodology (GTM) [14] as our basic research approach. We selected the Straussian variety because we expect its higher degree of structuredness to be more appealing to software engineers compared to the Glaser style – and we believe that both methods, if understood correctly, will lead to similarly valuable results.

We will not give a primer on Grounded Theory Methodology here. If you have not used GTM before, you might want to get a textbook about it and read it up; there are a number of such books. The Strauss/Corbin book (or its second edition but preferably not the third) is a possibility although other books may be easier to work with. To summarize it in a nutshell, GTM suggests to work as follows:

- GTM work aims at a conceptual explanation (*theory*) of some phenomenon of interest for which each element of the explanation (called a *concept* or *category*; we will only use the former term) is directly connected to one or more raw observations (*grounding*).
- Formulate your research interest. In our case this was "*Define the elementary behaviors which constitute pair programming.*"<sup>6</sup> The research question is allowed

---

<sup>6</sup>Note this aims only at elements of a theory, not the theory itself and hence requires not all



to drift freely during GTM work.

- Obtain some observation data. In our case this was the first handful of session recordings. GTM work does not require to pre-plan the data collection nor to achieve any kind of representativeness. Additional data will be collected once the researcher has found for what sub-phenomena more data is needed (*theoretical sampling*). For instance, a study of knowledge transfer in pair programming might find that the general knowledge level difference within the pair appears to be highly relevant. If no recording of an expert working with a true novice is yet available, the researcher would look for such a context and make a recording there. Representativeness is not required because GTM results focus on explaining things that exist, not on making claims about their frequency.
- Work through the observation data and annotate labels to phenomena that appear “interesting” with respect to the research focus. You need *theoretical sensitivity* to select relevant phenomena and appropriate labels. The phenomenon can be anything and of any granularity. Each label is the name of a (preliminary) *concept*; see Chapters 4 to 20 for examples. It is meant to be reused in several places in the data. Each concept is chosen such as to help explain some aspect of the phenomenon (*theoretical coding*). This process is called *open coding*.
- When assigning the same label again, make sure the phenomena are similar so that you will obtain a consistent concept. To do so, compare to all previous annotations of this concept (*constant comparison*), determine the commonalities, and record them in a *memo*. Make sure your concept assignment is fully grounded, that is, is based *only* on phenomena actually present in your data, not on any prior knowledge you might have (or rather: assume). The ungrounded use of any prior assumption when assigning a concept is called *forcing*.
- If the differences between phenomena annotated with the same concept appear relevant, represent them by auxiliary concepts: attributes (*properties*) and attribute values (also *properties*); apply constant comparison and memoing to them as well. This process is called *dimensionalization*. Avoid forcing.
- If you have accumulated enough isolated concepts, start discovering relevant relationships between concepts and validate them for specific phenomena. The relationships may pertain to context factors, constraints, causes, effects, the actor’s strategies, etc. This process is called *axial coding* and should also involve constant comparison as well as a lot of memoing. Avoid forcing. Meanwhile, open coding continues as well.
- If you have accumulated enough relationships, determine the core of the subject matter and extract those concepts around it that allow to formulate

---

components of the GTM. Only subsequent studies will aim at actual theories of (some aspects of) pair programming.

a narrative (*grounded theory*) that explains what is going on around this core concept. This process is called *selective coding*. Beware of forcing! Selective coding can start as soon as you have the first idea for it and should start no later than when you find you are detecting only known concepts, not creating new ones (*theoretical saturation*). Selective coding will often point out gaps in your conceptualization and hence trigger theoretical sampling, in particular if you start it early.

Working in this manner (with mostly open coding, some dimensionalization, a little axial coding, and no selective coding) and considering all of the above-mentioned aspects of the data, we were initially totally overwhelmed by the amount of information residing in our recordings. To cope with this, we developed several additions to plain GTM (see [12] for details), in particular:

- A perspective on the data: GTM suggests to initially conceptualize “everything” that may be of relevance and only start focussing on fewer concepts during selective coding. This approach does not work for data as rich as ours with a research question as open as ours. We decided early on that we would need to constrain ourselves to behavioristic concepts as much as possible (see Section 2.3.4 for details) and soon thereafter to conceptualize verbal interaction in far more detail than other behaviors (see Section 2.3.1).
- Structured concept names to further constrain and structure the applicable concept universe in order to make it manageable. See Section 2.1.1 and Section 3.1 for details.
- Pair conceptualizing: Doing GTM in pairs (which we originally called *pair coding*) helps to quickly weed out or improve inadequate conceptualizations, in particular early in a study when the concept set is still small and hence open to a multitude of possible additions, including additions that lead astray. This practice can save inordinate amounts of time and frustration.
- Furthermore, most GTM books recommend transcribing the data but adequate transcription of hour-long audio/video data that is as fine-grained and feature-rich as ours is hardly practical. So we annotate these data directly (without transcription) in the *ATLAS.ti*<sup>7</sup> data analysis software.

### 1.4.5 On using prior research results

When doing GTM, knowing a lot about your phenomenon in advance is a mixed blessing: On the one hand, such prior knowledge can greatly enhance your theoretical sensitivity and hence speed up the research process a lot. On the other hand, it can lead to forcing and thus ruin the validity of your results if you are not careful.

---

<sup>7</sup><http://en.wikipedia.org/w/index.php?title=Atlas.ti&oldid=559282786>

The solution suggested by (in particular Glaserian) GTM is to treat all sources of such prior knowledge (even if you have written them up yourself!) as additional observations – of a very different kind, but still to be considered. This idea is called *all is data*.

For the derivation of the base concept set, we were afraid of forcing and so have chosen to err on the side of knowing too little rather than too much (or the wrong things). So we started from the **epistemological**<sup>8</sup> stance of **Pragmatism**<sup>9</sup> (very roughly speaking: knowledge should be considered true if it leads to satisfactory results when applied in the world), hoped that we would have sufficient talent for **abductive reasoning**<sup>10</sup> to invent helpful concepts, went ahead, and read up on related work only *after* we had found and conceptualized the respective phenomena. The most important example of this is the notion of an illocutionary act (see Section 2.3.2) which we developed (and validated as consistent) in constant comparison manner before we searched for it in the literature, found it (as a central idea of speech act theory), and hence convinced ourselves that the approach was probably sound and valuable.

But this approach can obviously not continue if we are to use the overall research approach outlined in Section 1.4.3: If subsequent studies build on top of the base layer, this will involve using prior assumptions (in form of the base concepts) and obviously creates a danger of forcing. The danger is small because the concepts are grounded in data from the very domain. Also, the base layer safeguards against this danger by pointing out in many places that modifying certain details of individual concepts may be sensible in subsequent studies and by providing guidelines for how to do that in an orderly fashion; see Chapter 22 for details.

## 1.5 About this book

### 1.5.1 What this book is

- This book aims at providing a foundation for qualitative research into pair programming; research that aims at explaining the pair programming process.
- As this foundation, this book introduces and explains the base layer of concepts for understanding basic events in pair programming sessions.<sup>11</sup> The base layer consists of the base concept set plus rules for its use.

---

<sup>8</sup><http://en.wikipedia.org/w/index.php?title=Epistemology&oldid=565995152>

<sup>9</sup><http://en.wikipedia.org/w/index.php?title=Pragmatism&oldid=567854814>

<sup>10</sup>[http://en.wikipedia.org/w/index.php?title=Abductive\\_reasoning&oldid=562949053](http://en.wikipedia.org/w/index.php?title=Abductive_reasoning&oldid=562949053)

<sup>11</sup>Additional discussion can be found in [11].

Subsequent studies can build on this to create higher-level concepts and eventually theory.

- The base layer has two goals:
  - Helping a pair programming researcher to make faster progress with any one study.
  - Helping the pair programming research community to produce studies the results of which are compatible and that can easily be related to one another.
- This book explains each concept or concept class via a combination of (1) abstract definition (by means of prototypical properties that instances of each concept will usually share), (2) ostensive definition (by means of contextualized examples such as Example 3.1, Example 8.2), and (3) separate discussion of the delineation between pairs or tuples of concepts where the definition is not obvious from (1)+(2) alone.
- This book is a handbook for researchers (as opposed to a textbook). To make practical work handy, it is full of cross reference hyperlinks and often prefers hyperlinks to good Wikipedia articles over references to more scholarly<sup>12</sup>-looking paper-based sources – you can find such sources in the Wikipedia articles.

We guess that the base concepts may also be helpful when investigating types of technical pair work other than pair programming, even outside the software domain.

### 1.5.2 What this book is not

- This book does not contain advice for pair programming practitioners.
- This book does not contain a theory (complete or partial) of pair programming.
- This book does *not* provide a coding scheme (see Section 2.1).
- This book introduces no concepts for which we have never seen instances in our research data (but accurately grounded concepts only).
- This book provides (almost) no introduction into the qualitative research method we suggest to use (Grounded Theory Methodology).

---

<sup>12</sup>We would have liked to make this word a link to the “[scholarly method](#)”<sup>13</sup> Wikipedia article. Unfortunately, that article does *not* have good quality, so we do not.

- This book provides only little description of the research process by which we have arrived at the concepts explained herein. Refer to [11] if you need more detail.

### 1.5.3 How to read this book

Since we consider this book a handbook, not a textbook, there is no need to read it in full. Rather, you should develop a good understanding of its basic ideas and then consult it whenever a decision is unclear that arises in your research work. Here is a proposal how to develop your understanding of the basic ideas:


1. Read chapters 1 to 3 to obtain an overall orientation of how it all works.
2. Read the introduction and first subsection of each of the chapters 4 to 17 to deepen your understanding of the major concept classes and their distinction. As you go, take note which of them you find most interesting or intriguing.
3. Now pick the two most intriguing concept classes and read the respective two chapters in full in order to appreciate the subtlety of discriminating the concepts consistently. Study the examples thoroughly. Follow some of the cross references to get a feel for the hypertext character of the book. Make use of the index a few times, e.g. to locate additional examples or discussion involving a particular concept.
4. Read the introduction of Chapter 19 and skim at least its subsections 19.2, 19.3, and 19.4 to understand the non-dialog-oriented part of the base concept set.
5. For sake of completeness, have a short look at chapters 18 and 20.
6. Skim chapters 21 to 23.

An alternative method would be to start off to read the whole book, but then skip to the beginning of the next chapter whenever you get bored or overwhelmed.

Done? Congrats! You are now ready to go with your own base-layer-based pair programming research.

### 1.5.4 How to start performing research based on this book

To give you a rough idea how research based on this book might proceed, we provide a short sketch for your first attempts here. You should develop your own style over time.

1. Learn about the base layer by following the instructions in Section 1.5.3.
2. Formulate your research question.
3. Obtain a handful of recordings of pair programming sessions. If the phenomena you investigate are frequent, three to five recordings will be sufficient in the beginning; you can get additional ones later once you have a better idea what you are looking for (theoretical sampling).
4. Exercise with the base layer, for example by fully annotating all base concepts that apply to at least a five-minute stretch of each of your sessions. Intensely make use of the book as you go. Try to pick an interesting-looking stretch by first viewing the whole session once in TV fashion and taking some notes.
5. When you proceed to develop your first own concepts, there is a big difference depending on your research interest. For some interests, one or a few base concepts will lend themselves for an easy start. For example, if you are interested in knowledge transfer phenomena, you might start by annotating all *explain\_knowledge* utterances and build islands of annotations around them. For other research interests, this does not work, because there are no obviously related base concepts to start from. For instance, if you are interested in pair members' roles, you cannot pick out a few helpful base concepts that are obviously relevant. Rather, you would perhaps start annotating all phenomena with base concepts until you detect interesting phenomena and start introducing your own concepts for them. There is no need to think about layers yet; you can decide this later.
-  6. If you find yourself *modifying* the base layer (rather than just adding to it) in places where this possibility is not mentioned in this book<sup>14</sup>, re-read Chapter 22, be honest with yourself, and make sure you are doing the Right Thing. (Such modifications are dangerous, because they may reduce the compatibility of base-layer-based studies which is a key idea of our research approach.)
7. Now several things need to be done for which there is no obvious order. Do them all at once or find your own best ordering: Refine your own concepts; learn annotating base layer concepts and your own concepts concurrently in one go or in some efficient sequential fashion; find out which base layer concepts you do not need to annotate for your research focus (or at least not always).

---

<sup>14</sup>The possibilities that *are* mentioned in the book are marked by the pen symbol in the margin.


8. Move from open coding into axial coding and selective coding; do not forget to write enough memos; formulate your theory. Write your research article.

## 1.6 Terminology and notation

The goal for the base layer is to provide a set of abstractions that can be used to explain<sup>15</sup> what pair programmers say and do. We call each such abstraction a *concept*. If the concept is about saying, the piece of saying explained by the concept is called an *utterance*, if it is about doing, the piece of doing explained by the concept is called an *activity*. The act of applying a concept to explain an utterance or activity (by assigning the concept to the utterance or activity) is called *annotation*. All other terminology will be introduced as we go.

Much of the discussion in the book will make use of quotes taken from the raw data used for our research. Such quotes are marked by quotation marks, color, font, and an icon in the margin as follows:


*“OK, what did we want to do?”* 

Some of these quotes are offset from the text (as above) but most appear inline, like this: *“The VirtualAttribute is here”*. Program identifiers (such as the `VirtualAttribute` here) are typeset in non-proportional font. 

The quotes are not verbatim: As the original recordings featured only German speakers, the utterances have been re-composed in English in such a way as to best reflect the illocution (see Section 2.3.2) and “feel” of the utterance and its apparent real-time thought process, rather than the wording or normal English phrasing – the German phrasing was far from textbook German as well.

If a quote is fictitious rather than real, it is quoted and italicized, but appears in black and without the icon, like this: *“I never said this. Nobody did.”*

Many quotes not only contain words and standard punctuation marks but also other marks, using a very simple transcription system with only the following markup elements:


- (...): Speaking pause  
One or more dots in parentheses. Each dot represents a pause of about one second.  
Example: *“To go on here (.) I think it’s good to reflect for half an hour in your head how really (..) to go the way.”* 

<sup>15</sup>For “explaining” versus “describing”, see Section 1.4.4.

- (;;;): Speaking pause with other activity  
 One or more semicolons in parentheses. Each semicolon represents a speech pause of about one second but meanwhile the speaker performs some manual activity (typically operating the keyboard or mouse).  
 Example: *"OK. (;;;) That would be (!...!)"*
- (~): An incomprehensible word
- (~~): Multiple incomprehensible words
- (~word): An almost incomprehensible (and hence unreliable) word
- <\*information\*>: Comment added by the transcriber  
 The content of triangle-star brackets is not spoken at all. These brackets contain additional contextual information added by the researcher during the transcription to make the actual verbal content more intelligible.  
 Example: *"The VirtualAttribute is here in pro <\*points on screen\*>."*
- (quiet words): Quiet speaking  
 A word or words in parentheses are spoken very quietly, more to oneself than to the partner.  
 Example: *"The get (..) (missing)"*
- <\*>replacement\*>: Replacement of a proper noun  
 A name appearing in the quote has been taken out and replaced by a generic term in triangle-doublestar brackets in order to protect the anonymity of a person or company.  
 Example: *"The thing is, <\*>Developer\*> and I yesterday discussed that we change the FeatureProxySet."*
- (!...!): breaking off  
 A bang-tripledot-bang in parentheses means the utterance ends prematurely; the speaker breaks off or trails off in mid-sentence for no observable reason in particular.  
 Example: *"Really great would be, if we could return something that always (!...!) (..) hm (...) Wait-a-sec that always (!...!)"*
- (!!...!!): getting interrupted  
 A bangbang-tripledot-bangbang in parentheses means the utterance ends prematurely because the partner starts speaking, thus interrupting the previous speaker. What the partner actually says may be quoted subsequently, paraphrased in subsequent discussion, or ignored entirely.  
 Example: *"They could send every minute or every ten (!!...!!)"*
- (!!subutterance!!): both partners speaking at once  
 The partner begins to speak (and the utterance is shown in a separate quotation) while the speaker continues to speak (and the respective part





of the utterance is shown with the markup in the present quotation).

Example: “*Um, then you can easily, then you can easily change it. (!!That had confused me!!)*” 

When annotating concepts to quotations, it sometimes becomes necessary to annotate more than one. There are two notations for this. Where two concepts, A and B, are annotated, A + B means “A applies to some part of the utterance and B applies to some other part (but we still consider it a single utterance)”. In contrast, A/B means “both concepts apply at once to an ambiguous utterance as a whole”. In Example 8.2 (2), both of these notations appear at once:

C5.*explain\_knowledge* + C5.*amend\_strategy*/C5.*disagree\_step*  
means one part of the utterance contains an *explain\_knowledge* and another part has aspects of both an *amend\_strategy* and a *disagree\_step*.

The real-world phenomena explained by the concepts described here are far too complex to provide complete instructions for classification. You will often get to a point where our instructions end and you need to make up your mind yourself. Insofar as we are aware of these spots, the most interesting cases will be marked with the “Think!” symbol shown on the side of this paragraph. The respective page numbers are collected in the index under “Think!” 

Finally, paragraphs that discuss potential modifications of the base layer or extensions to it are marked with the pen symbol (like at the present paragraph). The respective page numbers are collected in the index under “base layer modification” .

Outside of quotations, italics are used to discriminate X from an utterance regarding X. The later are what most concepts refer to and what will be typeset in italics; see the discussion in Section 2.3.5.



# Overview of the base layer

## 2.1 What are the base concepts?

As mentioned previously in Section 1.6, the base concepts explain what pair programmers say and do on the level of individual utterances and individual activities. As explained in Section 1.4.3, the base concepts are intended as a foundation for a variety of subsequent studies of pair programming. To do that, the base concepts remain neutral: they are not geared towards any particular research question and attempt to be as generic and flexible as possible.

The base concepts are designed to form a coherent whole: the base concept set. Most of this book (Chapters 4 through 20) defines the base concept set by explaining the base concepts grouped into topical blocks. We cannot provide a complete and exact definition of any concept, because our concepts represent complex, real-world phenomena. Rather, we will provide a multitude of explanations that each aim at distinguishing *some* cases of instances from non-instances of a concept. In particular, we sometimes provide what we call the *primary characterizing attributes* of a concept. These are characteristics that are by construction necessary to be able to identify all instances of a concept; see for example the typology of *strategy* utterances in Section 9.1.

It is important to understand that the base concept set does not aim to be a *coding scheme*. A coding scheme would attempt to define each code (concept) in such a way that it can be identified as mechanically as possible in order to maximize inter-rater agreement. The idea of a coding scheme is incompatible with Grounded Theory Methodology. In contrast, the base concepts will be defined in a manner that attempts to maximize the reader's capability of *thinking flexibly* about what it is that appears to be going on in the pair programming session and what might be an appropriate manner of conceptualizing it. We will frequently encourage you to modify a base concept if it does not appear to

be able to express the phenomena of interest to your research well.

### 2.1.1 Concepts and concept classes

The concepts (with very minor exceptions) have structured names consisting of a verb and an object, such as *propose\_design*, *propose\_step*, *agree\_step*. We call the set of concepts that share a common object a *concept class*.

This structure makes it reasonably easy to remember and use correctly the concepts although their overall number is not small: There are more than 70 of them. This idea (and the individual verbs and objects) will be explained in more detail in Chapter 3.

### 2.1.2 HHI concepts vs. HCI/HEI concepts vs. supplementary concepts

The structuring of the base concept set does not stop at concept classes. Above them, there are larger groups of concepts we call concept categories. The primary concept category, forming the heart of this book, is the HHI concepts, for “human-human interaction concepts”. They conceptualize *saying*: the verbal dialog between the pair members. The HHI concepts (part II of the book) consist of 13 concept classes (explained in Chapters 4 to 17) and two extra concepts (Chapter 18).

The secondary concept category are the HCI/HEI concepts, for “human-computer interaction and human-environment interaction concepts”. They conceptualize *doing*: interaction of a pair member with the computer or with the rest of the environment (other than the partner and the computer). These concepts are far less important for the base concept set and hence much less detailed and refined. The strange two-part name HCI/HEI stems from the fact that these concepts do not discriminate between e.g. reading off the screen versus reading off paper or between pointing to the screen versus pointing to a poster on the wall. The HCI/HEI concepts are explained in Chapter 19. There is only one chapter because the notion of concept classes fully applies to the HHI concepts only: All HCI/HEI concepts share the same pseudo-object *sth* (for “something”) and some other concepts use that pseudo-object, too.

Beyond these two concept categories there are only a few “supplementary concepts” explained in Chapter 20. Together with the HCI/HEI concepts, they form the rather short part III of the book.

### 2.1.3 HHI concept class groupings

There is an intermediate level of grouping in between the concept classes and the HHI category as follows:

1. **Product-oriented concepts** (2 concept classes) concern proposals (and their discussion) regarding the content, structure, and placement of the program artifacts, ranging from requirements and architectural design down to individual identifiers and operators.
2. **Process-oriented concepts** (5 concept classes) concern proposals (and their discussion) regarding the work process. Furthermore, we will call the union of product-oriented and process-oriented concepts the “P&P” concepts.
3. **Universal concepts** (6 concept classes) concern utterances that request, transfer, or judge knowledge, state the level of available knowledge, or formulate or judge an assumption or hypothesis. They are called universal because they apply to the product context as well as the process context. The “Facade concepts” (1 concept class) are a subset of the universal concepts and provide a simplified view for (currently only one type of) complex phenomena. They provide a context in which a complex phenomenon can be split up and its parts be described by other concepts, whether base concepts or higher ones. Currently, the only facade concepts concern the verbalization or evaluation of HCI/HEI activities.
4. **Miscellaneous concepts** (2 concepts) mark utterances that are unintelligible or have nothing to do with the pair programming task at hand.

## 2.2 What is the base layer?

The base layer is the union of the base concept set plus guidelines for its use. These guidelines are spread throughout the book and then supplemented and summarized in the final part IV of the book. Part IV consists of

- a summary (in Chapter 21) of the annotation procedure rules scattered throughout parts II and III;
- guidelines for adapting the base concepts to the needs of a specific subsequent study (Chapter 22);
- preliminary guidelines for creating your own concept layers (Chapter 23).

## 2.3 Key decisions for the base layer

The base concepts would look very different had we not made the fundamental decisions given below. For understanding the base concepts and for applying them correctly, it is helpful to keep these decisions in mind.

### 2.3.1 Primarily rely on verbalization

One possible attitude when analyzing pair programming sessions via GTM would be “Pair programmers do many different things. One of these things is talking. But what counts is the code produced, so I am more interested in what they type into the computer.”

However, we quickly found that the talking contains *far* more interesting information than all of the other information channels and so decided to model the talking in much greater detail than the rest. This is reflected in the HHI part of the base concept set being far larger than the rest, both in terms of the number of concepts and the level of detail in their description.

### 2.3.2 Model illocutionary acts

Our second insight was that one must not take the talking at face value but rather probe for its actual semantics in the pair’s dialog. For instance, when a speaker asks a question, this is often neither intended nor understood as a request for information. Rather, it often is a proposal to do something specific – and that is just how the partner will react: with various forms of agreement or disagreement. To be useful, the base concepts should model such an utterance as a proposal, not as a question.

To put it a little more formally: Most of the HHI concepts designate what the **speech act theory**<sup>1</sup> of Searle<sup>2</sup> and Austin<sup>3</sup> calls the *illocutionary act* of the utterance. The illocutionary act is the action that is performed by making the utterance. For *ask* this would be *asking a question*, for *propose* it would be *making a proposal* even though the verbal and even the grammatical form may suggest something different. For instance, some pair programmers may formulate most of their proposals as questions.

The researcher needs to make an effort to identify the actual illocution of an utterance rather than sticking too closely to the explicit formulation. Often the intonation provides a good hint. If it does not, recognizing the illocution is often difficult for an utterance in isolation, but is usually clear once one considers other utterances from the session context before (and perhaps even after) the given utterance. Do not worry: The many examples provided along with the detailed discussion of the concepts below will make this process very clear over time.

Assuming sane speakers, the illocutionary act basically is the speaker’s *primary intention*. This raises two problems. First, is there always exactly one primary

---

<sup>1</sup>[http://en.wikipedia.org/w/index.php?title=Speech\\_act&oldid=566894006](http://en.wikipedia.org/w/index.php?title=Speech_act&oldid=566894006)

<sup>2</sup>[http://en.wikipedia.org/w/index.php?title=John\\_R.\\_Searle&oldid=567171344](http://en.wikipedia.org/w/index.php?title=John_R._Searle&oldid=567171344)

<sup>3</sup>[http://en.wikipedia.org/w/index.php?title=J.\\_L.\\_Austin&oldid=567339888](http://en.wikipedia.org/w/index.php?title=J._L._Austin&oldid=567339888)

intention? (And what does “primary” mean anyway?) Second, speech act theory knows hundreds of different illocutionary acts. How can this be compatible with our need for a modestly-sized base concept set?


Neither question has a short, neat answer and the detailed answers to both problems cannot be given here; they are spread over the discussions of the individual HHI concept classes in chapters 4 to 17. Chapter 21 will provide a summary.

### 2.3.3 Let segmentation emerge

For some qualitative research methods that annotate raw data with concepts, it is very important to segment the raw data in a canonical manner, and elaborate rules are often needed to provide that manner. For our purposes, this is unimportant: Early on, our work found that the *utterance* is usually (there are occasional exceptions) the right level of granularity for defining and applying HHI concepts; we do not even need to determine sharp boundaries for each and every utterance. (For the segmentation of non-HHI *activities*, see Chapter 19.)

But what *is* an utterance? Our seemingly paradoxical answer is: An utterance is that which is neither too small to be explained by one concept nor so large that the application of one concept no longer suffices to explain it.

This circular definition works well in practice because the researcher brings in sufficient common sense knowledge of communication and its units of meaning. From this starting point, the HHI concepts can be developed from utterances and the notion of utterance can be derived from the granularity of the HHI concepts. Much like the notion of illocutionary act above, this idea may look mysterious, but we assure you that you will find it perfectly natural once you have finished working through this book.

This principle applies to our derivation of the base concepts as well as your use of them. It may have to be changed when progressing towards studies using qualitative data analysis for quantitative purposes, e.g. comparing settings regarding the frequency of certain events. 

### 2.3.4 Crave for behavioristic interpretation

Determining the illocution of an utterance invariably involves some interpretation. Our fourth insight was that such interpretation can become a slippery slope: There is constant danger of making assumptions that are unwarranted and hence lose the grounding of the annotations made.

**Behaviorism**<sup>4</sup> is a psychological school of thought that assumes the mechanisms of behavior can be adequately modeled by referring to observable stimuli and

---

<sup>4</sup><http://en.wikipedia.org/w/index.php?title=Behaviorism&oldid=568442418>

observable responses alone, without referring to internal processes (“thinking”). Methodologically, behaviorism implies to abstain from the interpretation of observations as much as possible. Behaviorism has helped psychology to focus more on falsifiable statements and get on a proper scientific track.

To escape the slippery slope of losing grounding in pair programming research, a behavioristic style of observation and description is a helpful ideal for the researcher. The base concepts were created by applying that ideal to the extent that appeared reasonable: Unfortunately, utterances are far too ambiguous to determine their illocution in a purely behavioristic fashion without any interpretation and assumption, but a behavioristic ideal creates a useful limit to the kind of concepts one is still willing to introduce.

As a result, we think the base concepts, while usually requiring interpretation when annotating (not rarely involving substantial uncertainty), almost always create annotations that a large majority of researchers will be able to agree on as acceptable. Note that “acceptable” is a far lower degree of agreement than “objectively correct”, the latter being impossible in this domain. Behavioristic interpretation (a contradiction in terms!) is an ideal, not a goal.

### 2.3.5 Model the discourse world, not the activity world

The exact meaning of a base concept can be confusing until you realize the difference between a thing *X* and an utterance about *X*.

For instance the base concepts use the term *step* for referring to atomic units of work (in a sense that will be explained in Chapter 6). But the *step* object that forms base concepts such as *propose\_step* or *disagree\_step* does not refer to a step, it always refers to an utterance about a step.

This was our fifth insight: In contrast to the HCI/HEI concepts, HHI base concepts never<sup>5</sup> refer to an element of the activity world (a deed), they always refer to an element of the discourse world (an utterance). Corresponding activities may exist (at that time or some other) or not exist.

The book uses non-italicized words (such as *step*) to refer to the activity world and italicized ones (such as *step*) to refer to the discourse world. Some of the explanations are only intelligible when paying attention to this difference.

### 2.3.6 Model dialog episodes

It would be helpful if an encoding using the base concepts provided value beyond individual utterances and suggested a segmentation of the session into *episodes*. The objects of the base concepts do exactly that. An object (for example *strategy*) represents an individual topic (a particular strategy *X*) brought up

---

<sup>5</sup>Exception: The statement does not hold for the *activity* concepts.



in the session. The various verbs represent the lifecycle of the topic: It starts with an initiative verb (Section 3.5) such as *propose* (perhaps preceded by a query: *ask*), goes through any number of evaluations (*agree, disagree, decide*) and modifications (*challenge, amend*), and either ends with an evaluation, with switching from talking to action, or is superseded by some other topic for the time being – note that the same speaker may speak several times in a row, perhaps with pauses, in this process. The annotations will hence directly represent (some) relationships between the utterances.


This effect is convenient, but you should be aware of its limitations:

- The particular topic is not a fixed object (such as strategy X), but is modified during the discussion (as in Example 9.3), sometimes beyond recognition.
- These \*\_strategy annotations may be interspersed with others, in particular using universal concepts.
- Also, the HHI concepts, except for the *activity* concepts, do not indicate if an utterance relates to the result of an HCI/HEI activity. For instance, an utterance may relate to a stretch of code that has been highlighted using the mouse. Relationships are less obvious in such cases.
- The episode marking is implicit and can hence be ambiguous. In particular, an utterance may relate to more than one topic and several episodes may mix.

When annotating base concepts to data, making and keeping relationships visible is one of the criteria for the concepts to choose. However, the highest priority is always given to intersubjective validity: Your choices need to be explainable and agreeable. Note that the above notion of episode is useful for understanding and applying the base concepts, but once you start your own pair programming research you will need to introduce a notion of episode that is suitable for your particular research question.

### 2.3.7 Design the concepts to reflect relevant phenomena

The above decisions in no way make the choice of concepts canonical. There are many possibilities left how to structure the domain: The base concept set is not the inevitable result of natural science, it is a construction. In our decisions how to construct it, we have followed our best judgment (and the preliminary insights that arose during our analysis) in order to create concepts that reflect interesting and relevant phenomena.

If and where the results do not fit your particular research topic, you can and should modify them appropriately. 



## **Part II**

# **The HHI concepts: Human/human interaction**

...in which we define the concepts that aim at verbal interaction. You have reached the heart of the book and are going to be pumped through its arteries now; hold on.



# Objects and verbs of the HHI concepts

## 3.1 The structure and meaning of concept names

As mentioned in Section 2.1.1, the base concept set uses structured concept names. For the HHI concepts and the HCI/HEI concepts alike, each concept name consists of a verb and an object, joined by an underscore. Examples of HHI concept names would be *explain\_knowledge* or *propose\_step*. Some of the objects are not words but phrases, as in *explain\_standard of knowledge*.

As the same verbs and objects tend to occur in multiple concepts, this chapter provides a simple explanation of each verb and each object as a first approximation of the HHI concepts. Be aware that many of the concepts and discriminations between concepts are subtle (that is why we need a whole book to explain them), so this approximation will be coarse.

Each object induces a concept class: the set of all concepts involving this object (precisely: all concepts whose name contains the object. Still more precisely: whose name contains the name of the object. However, while the difference between a concept name and the actual concept is a major topic of this book, we can and will afford to largely identify the name of an object with the object itself, likewise for the verbs.). We will structure most of our discussion of the concepts along these object-based concept classes (object classes).

We will sometimes (but not often) use the name of an object to refer to the object itself and sometimes (and more commonly) to refer to the concept class induced by the object. For example, *knowledge* may occasionally refer to a certain kind of knowledge (which we will carefully define and delineate later in the book) but will more frequently refer to the set of concepts *explain\_knowledge*, *agree\_knowledge*, *disagree\_knowledge*, *challenge\_knowledge*, *ask\_knowledge*. Remember,

however, that concepts describe utterances (Section 2.3.5). The third use of *knowledge* is therefore to explicitly or implicitly refer to *knowledge* utterances – and thus to instances of a concept rather than a concept class. Likewise for all other HHI concept classes.

This third use is because, by virtue of being a concept, each object is a class itself: The class of all instances of that object. For example, when using the concept *propose\_step* to annotate a particular utterance found in a particular pair programming session, *step* will refer to an utterance about one particular step being proposed out of an infinite number of conceivable steps being proposed or executed in any pair programming anywhere. When a concept is used in this manner, we call it a code. In this book, most mentions of concept names refer to the concept, not to a code. The primary exception are the illustrative examples; all full-blown examples (pair programming episodes) are highlighted.

Each verb induces a verb concept class as well, but we will not make much use of those.

If you are confused by these explanations, do not worry: if you pay attention to the difference between the discourse world and the activity world as introduced in Section 2.3.5, things will become clear over time.

## 3.2 The objects

Our analysis for HHI concepts found 16 objects:

*activity*: An HCI or HEI activity that is currently ongoing.

*completion*: The degree of completeness of working through a particular work step. See also *step* and *state*.

*design*: A possibility or choice for the content of the artifact(s) being worked on. May pertain to individual elements (such as the name of a method) or to higher-level structure.

*finding*: An insight that was achieved shortly before (and then verbalized) by one pair member. Indicates the extension of knowledge based on cognitive processes. See also *knowledge* and *standard of knowledge*.

*gap in knowledge*: A lack of a particular piece of knowledge in both pair members. See also *standard of knowledge*.

*hypothesis*: A hypothesis or conjecture, often regarding properties of the artifacts being worked on.

*knowledge*: Explicit knowledge that is neither meta-knowledge of the *gap in knowledge* or *standard of knowledge* types nor newly won knowledge of the *finding* type, nor a *hypothesis*, nor is knowledge packaged as a suggestion regarding

the work product or work process. Whether true or not, the speaker assumes it to be true.

*off topic*: Anything that has nothing to do with the task or its solution.

*requirement*: A given or assumed requirement or constraint for the solution to be produced.

*something/sth*: An unspecifiable object. For HHI concepts, this is used only with the verb *mumble*.

*standard of knowledge*: How much one pair member knows with respect to a certain topic, except where this is a *gap of knowledge* (of both participants).

*state*: The degree to which a strategy has been worked through. See also *strategy* and *completion*.

*step*: A potential next step in the work process that the speaker considers to be an atomic unit of tactical behavior. See also *strategy*.

*strategy*: A somewhat longer-term (explicit or implicit) work plan for solving a (sub)problem that has not yet been fully worked through. Strategies typically cover multiple steps. See also *step*.

*todo*: A subtask or step that is to be performed not now but rather at some specified or unspecified later time during the same or a future programming session.

All of these only refer to entities a pair member has verbalized, if perhaps rather implicitly.

### 3.3 The verbs

Our analysis for HHI concepts found 13 verbs:

*amend*: Extending, complementing, or detailing either a previous utterance (HHI, typically *propose*) or a currently performed activity (HCI/HEI). This mostly expresses agreement, not disagreement.

*ask*: Asking a question; typically an open one, sometimes a closed one.

*agree*: Expressing consent with either a previous utterance (HHI, typically *propose*) or with a currently or just previously performed activity (HCI/HEI). *agree* does not include explanations; if explanations occur, they are covered by an additional *explain* concept. See also *decide*.

*challenge*: Expressing dissent with either a previous utterance (HHI, typically *propose*) or with a currently or just previously performed activity (HCI/HEI) and (or by) making a counter-proposal. See also *disagree*.

*decide*: Selecting one from among a number of possibilities (typically presented by the partner in form of a *propose*). See also *agree*.

*disagree*: Expressing dissent with either a previous utterance (HHI, typically *propose*) or with a currently or just previously performed activity (HCI/HEI) without making a counter-proposal. See also *challenge*.

*explain*: Explaining something to the partner. Can occur as a reaction (to e.g. an open or closed question from the partner, see also *ask*) or without an observable trigger.

*mumble*: Making an utterance that is so fragmentary or phonetically so unclear as to be uninterpretable. Will only be used with the object *sth*.

*propose*: Making a proposal that does not refer to a recent other proposal. The proposal may consist of one possibility or several (as a choice). The partner can react with *agree* (in case of one possibility), *decide* (in case of several possibilities), *challenge*, *disagree*, or *amend*.

*remember*: Reminding oneself and the partner of something specific.

*say*: Saying something. Will only be used with the object *off topic*.

*stop*: Proposing to terminate a particular HCI/HEI activity.

*think aloud*: Verbalizing one's own current activity or its related considerations.

### 3.4 The existing object/verb combinations

Overall, there are 60 HHI concepts, so by far not all of the 208 combinations of 13 verbs with 16 objects do occur. This has a number of reasons:

- Some combinations simply make no sense.
- Some combinations make sense and can occur in practice, but are sufficiently rare that we never encountered them in our data. Since all of the concepts explained in this book are fully grounded in data to make sure they are valid, such unseen concepts are missing in our concept set. As discussed in Section 22.4, you can and should introduce them as soon as you encounter instances of them in your own data and need the respective concept for your particular analysis.
- Some combinations make sense, but would overlap in meaning with some other concept. To make coding canonical we have thus ruled out one of them. For example, there is no *explain\_step* concept, because the base layer defines that all explicit knowledge transfer is to be represented by means of the *knowledge* concept class.



- There are two broad auxiliary concepts (*mumble\_sth* (mumble something) and *say\_off topic*) of low importance for which it was defined that their verbs must not be combined with anything else.

Most of the time most of the verbs designate the illocutionary act (Section 2.3.2).

Figures 3.1 and 3.2 summarize all HHI concepts, grouped into concept classes and the larger concept class groupings. The subsequent chapters (pages 59 to 178) will work through the individual HHI concept classes object by object in a roughly uniform structure.

### 3.5 Types of verbs

The HHI concepts have three different types of verbs:

*Initiative verbs* introduce a new aspect into the discourse. Usually, no direct reference (whether explicit or implicit) is made to the content of previous utterances. Initiative utterances do thus not have the character of a reply, although they may still address a topic previously discussed. The base concepts provide three initiative verbs: *propose*, *remember*, and *stop*.

*Reactive verbs*: Utterances that directly (whether explicitly or implicitly) refer to one or more previous utterances.

The verbs *agree*, *amend*, *challenge*, *decide*, and *disagree* can be used as reactive verbs (but also to comment on activities).

*Bivalent verbs*: The verbs *explain*, *think aloud*, and *ask* can be used in an initiative fashion as well as a reactive fashion. For instance, a pair member may attempt a knowledge transfer either based on a previous question of the partner or on any other trigger (observable or not), but both cases are conceptualized as *explain*.


The verbs *mumble* and *say* carry only little meaning and are not classified.

There is a second verb classification besides the above one:

*Constructive verbs* are those where the utterance constructively introduces additional aspects into the discourse: *amend*, *challenge*, *explain*, *propose*, and *remember*.

*Unconstructive verbs* are those where the utterance only requests new aspects or information or judges such aspects or information. These are *ask*, *agree*, *decide*, and *disagree*.

For this second classification, *think aloud* and *stop* are ambiguous and hence not classified.

We call all concepts containing a constructive verb *constructive concepts*, etc. This book will not make much use of these classifications but they may be 

product-oriented concepts		process-oriented concepts		
<b>amend_design</b>	<b>ask_design</b>	<b>amend_step</b>	<b>ask_step</b>	<b>explain_completion</b>
Extend a given proposal regarding the structure and content of the program without rejecting the proposal.	Ask for a concrete proposal regarding the structure and content of the program.	Extend a given proposal regarding the next tactical work step without rejecting the proposal.	Ask for a concrete proposal regarding the next tactical work step.	Make a statement regarding the degree of completion of the current tactical work step.
<b>challenge_design</b>	<b>agree_design</b>	<b>challenge_step</b>	<b>agree_step</b>	<b>agree_completion</b>
Reject a given proposal regarding the structure and content of the program and make an alternative proposal instead.	Signal agreement with a given proposal regarding the structure and content of the program.	Reject a given proposal regarding the next tactical work step and make an alternative proposal instead.	Signal agreement with a given proposal regarding the next tactical work step.	Signal agreement with a statement regarding the degree of completion of the current tactical work step.
<b>decide_design</b>	<b>propose_design</b>	<b>decide_step</b>	<b>propose_step</b>	<b>challenge_completion</b>
Select one from among several alternative proposals regarding the structure and content of the program.	Make one or several alternative proposals regarding the structure and content of the program.	Select one from among several alternative proposals regarding the next tactical work step.	Make one or several alternative proposals regarding the next tactical work step.	Reject a statement regarding the degree of completion of the current tactical work step and make an alternative statement.
<b>disagree_design</b>		<b>disagree_step</b>		<b>explain_state</b>
Reject a given proposal regarding the structure and content of the program without making an alternative proposal.		Reject a given proposal regarding the next tactical work step without making an alternative proposal.		Make a statement regarding the degree to which the current strategy or work plan has been worked through.
	<b>remember_requirement</b>	<b>amend_strategy</b>	<b>ask_strategy</b>	<b>agree_state</b>
	Remind the pair of a given (pre-specified) functional or non-functional requirement of the program.	Extend a proposed strategy or work plan without rejecting it.	Ask for a concrete proposal regarding the strategy or work plan to be chosen.	Signal agreement with a statement regarding the degree to which the current strategy or work plan has been worked through.
<b>challenge_requirement</b>	<b>agree_requirement</b>	<b>challenge_strategy</b>	<b>agree_strategy</b>	<b>challenge_state</b>
Reject a given or proposed requirement and propose an alternative one instead.	Signal agreement with a given or proposed requirement.	Reject a given proposal regarding the strategy or work plan and make an alternative proposal instead.	Signal agreement with a given proposal regarding the strategy or work plan.	Reject a statement regarding the degree to which the current strategy or work plan has been worked through and make an alternative statement.
	<b>propose_requirement</b>	<b>decide_strategy</b>	<b>propose_strategy</b>	<b>propose_todo</b>
	Propose one or several alternative program characteristics that should be considered to be a requirement.	Select one from among several alternative proposed strategies or work plans.	Propose one or several alternative strategies or work plans.	Suggest that a certain work item will need to be taken care of later in the process.
<b>mumble_sth</b>	<b>say_off topic</b>	<b>disagree_strategy</b>		<b>agree_todo</b>
Make an incomprehensible utterance (highly fragmentary or acoustically unclear).	Make an utterance that has nothing to do with solving the programming task.	Reject a given proposal regarding the strategy or work plan without making an alternative proposal.		Signal agreement with a statement saying that a certain work item will need to be taken care of later in the process.
<b>miscellaneous</b>				

Figure 3.1: The HHI concepts, part 1: P&P concepts (i.e., produce-oriented and process-oriented concepts) and auxiliary concepts.

universal concepts			
<b>explain_gap_in_knowledge</b> Verbalize that certain knowledge is not possessed by either member of the pair.	<b>agree_gap_in_knowledge</b> Signal agreement with a given gap in knowledge.	<b>explain_standard_of_knowledge</b> Explain or recapitulate one's own level of knowledge with respect to a certain topic.	<b>ask_standard_of_knowledge</b> Ask the partner for his/her level of knowledge with respect to a certain topic.
		<b>ask_knowledge</b> Ask the partner for information of type 'declarative knowledge'.	<b>stop_activity</b> Suggest to stop or abort the current HCI or HEI activity.
<b>explain_finding</b> Verbalize a new insight; this includes interpreting an observed event.	<b>propose_hypothesis</b> Formulate a hypothesis or conjecture, e.g. regarding a property of the program, or the environment.	<b>explain_knowledge</b> Transfer information to the partner that is assumed to be correct declarative knowledge.	<b>think_aloud_activity</b> Verbalize aspects of one's own current HCI or HEI activity.
<b>agree_finding</b> Signal agreement with a verbalized insight or interpretation.	<b>agree_hypothesis</b> Signal agreement with a given hypothesis or conjecture.	<b>agree_knowledge</b> Signal agreement (i.e. judge as correct) knowledge stated by the partner.	<b>agree_activity</b> Signal agreement with all or part of the current HCI or HEI activity.
<b>challenge_finding</b> Reject the content of a verbalized insight or interpretation and suggest an alternative one.	<b>challenge_hypothesis</b> Reject a given hypothesis or conjecture and formulate an alternative one.	<b>challenge_knowledge</b> Declare transferred knowledge as fully, partially, or potentially wrong by opposing it with one's own knowledge.	<b>challenge_activity</b> Reject all or part of the current HCI or HEI activity and suggest an alternative activity.
<b>disagree_finding</b> Declare transferred finding as fully, partially, or potentially wrong without explaining why.	<b>disagree_hypothesis</b> Reject a given hypothesis or conjecture.	<b>disagree_knowledge</b> Declare transferred knowledge as fully, partially, or potentially wrong without explaining why.	<b>disagree_activity</b> Reject all or part of the current HCI or HEI activity.
<b>amend_finding</b> Extend a verbalized insight or interpretation without rejecting it.	<b>amend_hypothesis</b> Extend a given hypothesis or conjecture without rejecting it.		<b>amend_activity</b> Propose an extension to the current HCI or HEI activity.

facade concepts

Figure 3.2: The HHI concepts, part 2: universal concepts, including the facade concepts.

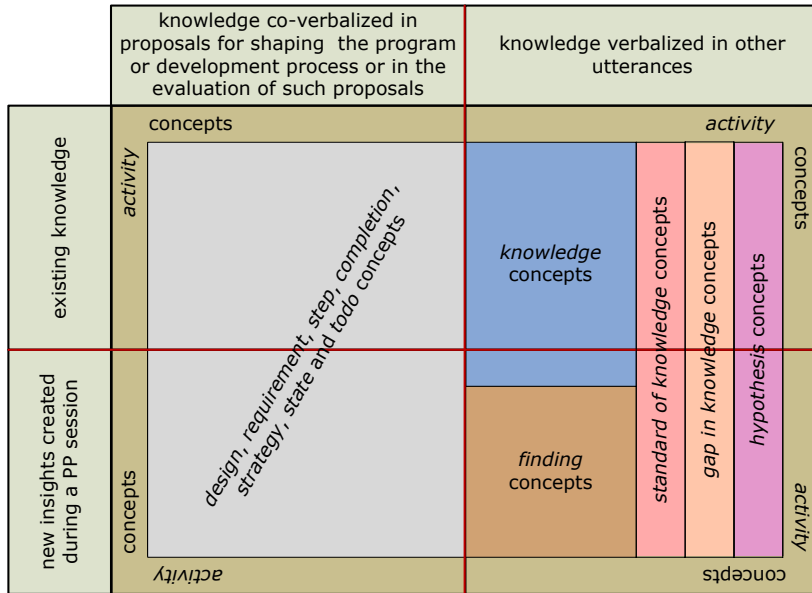


Figure 3.3: The landscape of knowledge-related utterances

helpful for use during higher-level analyses. You may then want to classify a few cases differently; for instance, questions (*ask*) can sometimes be considered constructive too.

### 3.6 The notion of “knowledge”

Most of this book is concerned with separating one concept from other similar ones. A few of these separations are particularly important or particularly difficult and we will therefore summarize some of this information upfront in this and the following three subsections. This provides the first pieces of understanding of base concepts that connect concept classes to one another.

Knowledge transfer is a key aspect of pair programming. In the base concepts, it is represented via the concept classes *knowledge*, *finding*, and *standard of knowledge*, but also as an aspect of several (in fact almost all) other concept classes. The following discussion illuminates some aspects of how the base concepts handle the various subtle problems that knowledge and knowledge transfer create during the analysis of pair programming. Use Figure 3.3 for guidance.

**The notion of “knowledge” in the base concept set:** Grounded Theory Methodology requires to ground all concepts and statements in data, that is, in concrete

observations. We therefore quickly decided we need to follow a behavioristic approach in our analysis: Rely only on what is directly observable and avoid modeling internal cognitive processes as best you can. For the notion of knowledge this means we can represent only that knowledge that its owner communicates<sup>1</sup> and where we observe this communication. In the following please note the discrimination between “knowledge” in general and “*knowledge*” in the narrower sense used in the base concepts. (Later on we will often more loosely write knowledge where in fact we mean *knowledge* if precision is not important.) These considerations lead to the following definition of *knowledge*:

- The *knowledge* concept covers only explicit knowledge (that its owner can consciously access and use), not tacit knowledge, and it covers only knowledge that is communicated, rather than only possessed.
- The communication of knowledge can be explicit or implicit. The explicit communication of knowledge can be verbal, this is the typical case, or wholly or partially nonverbal by sketches, gestures, etc. The implicit communication of knowledge happens by doing something and can occur unconsciously. *knowledge* is only explicitly communicated knowledge, not implicitly communicated knowledge. This restriction makes the current *knowledge* concept much more reliable, but may be too restrictive for some future analyses and should perhaps be lifted then.
- Knowledge is something the owner considers to be true, but it does not need to be actually true. This restriction is necessary because the researcher is very often not able to verify the owner’s belief. This restricted view of knowledge as belief also gets rid of many of the epistemological problems that the notion of knowledge otherwise holds.
- These first three clauses define the necessary conditions for seeing an instance of *knowledge*. The sufficient conditions are much more complicated, mostly (but not only) because most of the non-*knowledge* HHI concepts concern objects that fall under the above three-clause definition of knowledge as well, but the base layer defines that in those cases the *knowledge* concept should *not* also be used. For instance, when a pair member explains a certain design pattern as such, this will be annotated as *explain\_knowledge*. If this explanation occurs as part of a suggestion how to structure the program, it will be encoded as *propose\_design* instead, not mentioning the knowledge aspect separately at all. This business is subtle: When the design pattern explanation is not *part* of the proposal, but rather a *justification* for it, the total utterance will be split and the

---

<sup>1</sup>More precisely: the owner communicates (in some form of expression) information that aims to represent the knowledge and that may or may not turn into knowledge again at the receiver’s side. We ignore these transformations in the terminology of our book, which will be difficult enough without this additional complication.

parts annotated as *propose\_design* and as *explain\_knowledge*. Many more such distinctions are required. It is due to such distinctions that we need a whole book to explain the base layer and in most cases we will state the reasons for making them.

- Various things need to be subtracted from the definition of *knowledge* thus obtained. They will be discussed below.

The above definition of *knowledge* is almost synonymous to knowledge transfer, so we will often use this more explicit latter term. Transfer here stands for potential transfer, not necessarily successful or actual transfer, and in principle it indicates only the fact of potential transfer, not necessarily the intention of transfer. Furthermore, we will sometimes speak of information instead of knowledge.

**Separation of some forms of meta-knowledge:** The base concepts discriminate between a certain form of knowledge about knowledge (meta-knowledge) and other forms of explicit knowledge: The class *standard of knowledge* represents assessments one pair member makes about his or her own gradual level of knowledge (or lack thereof) with respect to a certain fact or topic.

**Partial separation of existing knowledge vs. new knowledge:** Quite obviously, things the pair finds out during a session are of particular relevance when analyzing the pair programming process. The base concepts hence discriminate knowledge that has been possessed by one or both of the partners for some time (existing knowledge) from knowledge that just a moment ago sprang into existence (new knowledge: insights). The latter is represented by the object class *finding*. Not all finding concepts represent verbalizations of insights, though. For instance, an utterance annotated with *amend\_finding* may contain only existing knowledge, but it *pertains* to a finding.

All knowledge the partners presumably already possessed before the session is considered existing knowledge, but *findings* become existing knowledge for later parts of the session as well.

**Partial separation of certain knowledge vs. uncertain knowledge:** The classes *knowledge* or *finding* are used if the speaker is subjectively certain of his or her knowledge. If s/he reveals s/he is not, *hypothesis* is used instead, no matter whether the knowledge is new or existing.

**Separation of product or process proposals:** Another aspect besides findings that is obviously key to understanding the pair programming process is proposals (pertaining to the composition of the artifacts being worked on or to the steps and strategies of the development process itself) as well as the evaluations of such proposals.

Most of the proposals and many of the evaluations contain and transmit knowledge but this is not annotated separately. So even an obviously knowledgeable

product-related proposal will only be annotated with *propose\_design* and not with an additional *explain\_knowledge*. However, if the proposal is subsequently justified, typically by explaining its rationale, that utterance will be annotated separately and as *explain\_knowledge* (or perhaps *explain\_finding*):

Example 3.1: An episode from session **BA1** (14:08:27–14:09:00) containing a design proposal complemented by a rationale based on a finding. The pair is working on an if statement that will stop the script if one of a number of conditions is met. The pair had previously marked some of the conditions as needing change (“TODO”).

(1) B2.*ask\_knowledge*

*“What happens if this one is not set?”*

The question refers to a script parameter.

(2) B1.*explain\_knowledge*

*“That makes the script continue at least as long as we (.), er, (.) do not make the ‘Change’ at each spot – write the Memcache. (...)”*

The driver answers the question immediately. We have no indicators that let us assume a *finding*, the answer comes from existing knowledge. After his reply, he pauses for three seconds.

(3) B1.*explain\_finding*

*“Although if we don’t write it to the Memcache, he’ll return it anyway, some time.”*

The combination of pausing and “although” suggests this was not existing knowledge.

(4) B1.*propose\_design*

*“We can leave that up there for now. (...)”*

Based on his insight, the driver now makes a proposal: Let us not change the marked conditions or at least not yet.

(5) B1.*ask\_standard of knowledge* + B1.*explain\_finding*

*“Know\_what\_I\_mean? If we now only change this script here (.) then it would only (.) this would never become true. So we would never return exit(‘NOCHANGE’) and just continue here – and that would be just the same functionality.”*

After a short pause, B1 justifies his proposal by explaining his insight again in different words. He points to various spots in the code while doing so.

Such separate annotation is used likewise for process-related proposals such as *propose\_step* or *propose\_strategy*, their status-evaluation cousins *explain\_completion* / *explain\_state* (where *finding* will be more typical than *knowledge*), and so on.

### 3.7 *propose vs. explain*

The same idea of discriminating objects concerned with a few important specialized types of knowledge from more general ones recurs for the verbs. For the objects, all proposals (and comments on such proposals) that concern the content of artifacts or the shaping of the development process are encoded by concepts from the *design*, *requirement*, *step*, *strategy*, or *todo* classes even if (as they usually do) they include knowledge transfer, while “pure” knowledge transfer (and comments about its content) is encoded primarily by *knowledge* and *finding* concepts.

This discrimination is reflected in the use of the verbs *propose* versus *explain*. For instance, there is a concept *propose\_design*, but no concept *explain\_design*. Explanations around a design are either explanations of the design as such (these are encoded as *propose\_design*, no matter how detailed) or justifications, explanations of the design’s rationale. The latter are considered “pure” knowledge transfer and are thus encoded as *explain\_knowledge*. Note that in practice it is often quite difficult to separate a proposal from its justification, because the proposals are often rather implicit; see Example 4.1.

Somewhat in between those two cases is the *hypothesis* class: It clearly belongs into the “pure” knowledge transfer, but is still used with the verb *propose* (not with *explain*), because in real sessions hypotheses are (at first) typically only stated, not elaborated.

### 3.8 *explain vs. think aloud*

What is the difference between the seemingly similar verbs *explain* and *think aloud*? The latter is used only in the rather peculiar concept *think\_aloud\_activity* which serves to bracket whole sequences of utterances describing the content, meaning, or purpose of HCI/HEI activities of the speaker that occur at roughly the same time.

In contrast to all other concepts, for which it is expected that only one concept will usually be annotated to any one utterance (see Section 2.3.3 for a discussion of segmenting speech into utterances), *activity* concepts are defined such that any of their utterances can be annotated with additional concepts, including *explain\_knowledge*, as needed. If you find this confusing or extravagant: It will be explained in Chapter 17.



### 3.9 *disagree+propose vs. challenge*

A *challenge* is basically a *disagree* combined with a *propose* (for P&P concepts) or with an *explain* (for universal concepts), so why do we introduce a whole additional verb concept class for this case?

There are two reasons: First, the fact that disagreeing and proposing are combined in a single step is important and would be lost when using separate concepts. Second, *propose* has been classified as an initiative verb, but *challenge* clearly is and must be reactive.



# Product-oriented concepts: *design*

## 4.1 Topic of *design* concepts

*design* concepts concern proposals (and their discussion) regarding the content, structure, and placement of the program artifacts in the broadest sense, insofar as they can be decided upon by the pair. These artifacts are typically execution-related ones such as program code or configuration files (and our observations covered only those), but could be other types as well. This definition includes different aspects ranging from architectural design down to individual identifiers and operators, and touches issues on various levels of granularity such as the expression level, statement or variable declaration level, control construct level, method level, class level, or package level.

Here are a number of examples:

Example 4.1: Separate examples of *design* phenomena from Sessions **ZB7**, **BA1**, and **CA2**. The explanations show how utterances can often be interpreted only using context information such as previous utterances (Example (c)) or recent computer outputs (Example (e)).

(a) **ZB7**: *Z20.propose\_design*

“A `TopicConnection` we need.”

Z20 is editing a newly created method clone. The original method sent an object via a JMS Queue, the clone is to be changed such that it uses a JMS Topic instead. Z20 is suggesting to change a declaration from type `QueueConnection` to `TopicConnection`.

(b) **BA1**: *B1.propose\_design*

“And how we call the thing? `id`?”

B1 is driver and suggests a name for a variable. The initial question is not annotated as *ask\_design*, because B1 answers it himself immediately and also starts typing right away.

(c) **BA1:** B2.*agree\_design*

“Ummmmm.”

Agreement to a previous proposal (in fact the one right above).

(d) **BA1:** B2.*propose\_design*

“What do we return? Shall we (!...!) (.) getFriendsLastChange. We want a timestamp, right? Or true or false?”

Observer B2 makes alternative proposals for possible return values. He frames the two choices as questions, but the initial question is rhetorical only: An answer follows right away. It is therefore not separately annotated as *ask\_design*.

(e) **CA2:** C2.*propose\_design*

“Yeah, that needs to go.”

C2 is driver and refers to a method call he intends to remove that is currently highlighted in the editor.

(f) **CA2:** C5.*propose\_design*

“We need to use the FeatureProxy to retrieve the values.”

C5 is observer. He suggests an approach for accessing required data.

Excluded from the *design* concept class are decisions fixed (at least conceptually) before the pair programming session; those are modeled by *requirements* concepts as defined in Chapter 5. Also excluded is the detection of the unavoidable need for corrections due to programming errors as discussed in Section 12.1.2.

## 4.2 design concepts and their properties

As shown in Figure 3.1, we have observed *ask\_design*, *propose\_design*, *agree\_design*, *decide\_design*, *disagree\_design*, *amend\_design*, and *challenge\_design*.

The following subsections explain how to recognize corresponding phenomena. Example 4.2 provides a complete design discussion episode.

Example 4.2: A design discussion episode from Session CA2 (12:55:47–12:56:49). The observer C5 suggests to refactor the constructor of class `DisplayNameFeatureProxyTableModel`.

**(1) C5.propose\_step**

*"I would (.), let us look at the TableModel, how that works, how that fetches the values."*

Before C5 finishes this utterance, C2 has opened the source file in the editor.

**(2) C5.propose\_design**

*"Well (.) I (.) think (.) that TableModel shouldn't know—mustn't know—that it's using virtual columns or non-virtual columns."*

C5 says this after a pause of 5 seconds during which C2 has not changed the view, which primarily shows the constructor. During this utterance, C2 starts scrolling downwards.

**(3) C2.agree\_design**

*"Okayyyyyy."*

C2 agrees half-heartedly.

**(4) C2.explain\_standard of knowledge**

*"That's the abstraction I'd thought of before."*

The context shows that C2 had apparently thought about this problem before the session. While speaking, he scrolls back to the constructor.

**(5) C2.mumble\_sth**

*"Have you 'n principle, haven't you 'n principle (!...!)"*

Fragment, unclear. May be a cut-off agreement.

**(6) C5.amend\_design**

*"That should (...), that should get itself a ValueProvider from the configuration."*

C5 adds an implementation detail to the proposal, referring to the constructor's parameter `attributeConfiguration`.

**(7) C2.agree\_design**

*"You're not wrong there, essentially."*

C2 more or less agrees.

**(8) C2.explain\_standard of knowledge**

*"Yes, that's the abstraction I'd thought of before."*

(see above)

(9) C2.mumble\_sth

“That means (...) OK. (!...!)”

Fragment, unclear.

(10) C5.amend\_design

“that you, that we extend the attribute configuration by such a ValueProvider and that will give it the Feature (.), that will give it the the the that that that FeatureProxy to fetch the values (.) and the rest happens after that.”

After a two-second pause, C5 continues the sentence of C2 and substantiates his design proposal.

### 4.2.1 Types and intentions of proposals

Most design discussion episodes start with a *propose\_design*, just as it is shortly described (for *strategy*) in Section 2.3.6. As you can see in Example 4.1, proposals are not often explicit. More commonly, and as explained in Section 3.7, no proposing verb is provided and only the content of the proposal is spelled out – often as an assertion (see (a) in Example 4.1). Another popular form is that of a question (see (b) in Example 4.1). Both forms can be combined by appending the question to the statement (e.g. “OK?”).




Whether the speaker considers the utterance to be a proposal or rather a case of courteously informing the partner of final decisions (if the speaker is observer this means: giving instructions) can often be decided only by relying heavily on the session context or not at all. When giving information or instructions, the speaker would neither expect nor desire a verbal reaction. For instance, driver-speakers often start typing right away (see (b) in Example 4.1) or even during their utterance.

The base concepts ignore this difference in intentions and encode both cases and all their variations as *propose\_design* (and likewise for *amend\_design* and *challenge\_design*), so that the concept addresses a broad variety of utterances that can be summarized as follows.

**Required and optional elements of *propose\_design* utterances:** A *propose\_design* can transport three types of information:

- i. A design aspect
- ii. The speaker’s positive evaluation of this aspect
- iii. A request to the partner to evaluate the aspect

Only the first of these is required, which leads to three plausible modes of *propose\_design* utterances:

- Mode OE (utterance provides i+ii+iii): Obtain evaluation. Example from CA2: “Well, I would, overall I would make this so that the FeatureProxySet (.) uses those TableModel properties (.) Yah.” This is a refactoring proposal. At its end, the speaker turns to his partner and looks at him – only this establishes the OE mode. 
- Type PI (i+ii): Provide information. Example from CA2: “You must overwrite it.” The observer instructs the driver C5 to overwrite a certain inherited method. The driver, however, does not do that. Instead, he suggests to have a look at the superclass first. 
- Type LO (i+iii): Look for orientation. Example from BA1: “The question is: Should we return anything at all?” See Example 4.3 for discussion. 

We have seen all of these three modes in our data but not the unlikely fourth in which the proposal would serve no purpose at all. The modes are often difficult to tell apart; in particular, the evaluation request tends to be implicit and gradual. They are nevertheless helpful as a mental model.

***propose\_design* and indirect speech acts:** The above three modes of speaker intention cannot usually be determined based on the syntactical form of the utterance or the terms used in it. For instance, an utterance in the form of a question may well be a proposal and can even be a PI-mode proposal. Searle speaks of **indirect speech acts**<sup>1</sup>: An illocutionary act of asking a question is involved, but it is only the secondary illocution. The actual primary illocution is making a proposal and the utterance is hence called an indirect speech act. The base concepts ignore the secondary act in such cases and represent the primary one only.

As mentioned above, all of these considerations also apply accordingly for *amend\_design* and *challenge\_design*. Very similar ones apply to other concept classes involving proposals, most notably *step* and *strategy* (*propose\_step*, *amend\_step*, *challenge\_step*, *propose\_strategy*, *amend\_strategy*, *challenge\_strategy*).

## 4.2.2 Referring to editing steps

We said above that *propose\_design* always transports a design aspect (information type i). Note that this transport needs not happen verbally.

Here are two non-verbal examples:

---

<sup>1</sup>[http://en.wikipedia.org/w/index.php?title=Speech\\_act&oldid=566894006#Indirect\\_speech\\_acts](http://en.wikipedia.org/w/index.php?title=Speech_act&oldid=566894006#Indirect_speech_acts)

- In Session CA2, C2 modifies a method and during that process utters “*It ought to have been implemented like this.*”. This constitutes a *propose\_design* in mode PI.
- In Session CA2, C5 executes an IDE wizard for creating a new Java package, enters the package name, and before clicking “Finish” asks “*Correct?*”. This constitutes a *propose\_design* in mode OE.

### 4.2.3 Proposals with rationale

A *propose\_design* is often accompanied by a justification: an explanation of the proposal’s rationale. Such explanations should be annotated separately as *explain\_knowledge* or *explain\_finding*. The same holds for justifications that accompany instances of *challenge\_design*, *amend\_design*, or *disagree\_design*.

- For instance, the observer in BA1 refuses a design proposal by saying “*But that’s horrible. Then we have two loops; that’s crap.*” The first sentence should be annotated as *disagree\_design*, the second as *explain\_knowledge*. We will revisit this rule in Section 16.1 about the *knowledge* concepts.

### 4.2.4 *decide vs. agree*

Using *decide\_design* requires that at least two different design options have been proposed previously. Then, *decide\_design* is an explicit choice of one as opposed to the other(s) – in contrast to a mere *agree\_design* with respect to the last-mentioned variant. *decide\_design* works much like *propose\_design*: If the partner subsequently prefers a different choice, this will be annotated as *challenge\_design*, not as another *decide\_design*.

We will see such an episode (regarding *strategy*) in Example 9.3.

### 4.2.5 *amend vs. a new propose*

We use *amend\_design* if the speaker makes a previous proposal more concrete by adding detail or an extension. It is often difficult for the researcher to discriminate such amendments from entirely new proposals, because the reference to the original proposal is almost always purely implicit.

### 4.2.6 *amend or challenge one’s own proposal*

It is quite common to find *amend\_design* (see Example 4.2) or *challenge\_design* utterances not only with respect to proposals coming from the partner, but also as self-corrections of the speaker’s own previous proposals. This statement may be true for all types of *amend* and *challenge* concepts in the base concept set, although we have not actually observed it in all concept classes. We will revisit this in Section 21.6.5.



### 4.2.7 Indicating agreement vs. indicating attentiveness

Many of the utterances that could be affirmative are quite short: “OK.”, “Right.”, “Yea.”, “Yes.”, “Hmm.”, and so on. This makes it often hard for the researcher to decide whether this is affirmation of content (and hence *agree\_design*) or rather what linguists call **backchannel**<sup>2</sup> communication: A mere indication of attentiveness and interest, without content-related meaning. That same ambiguity exists for the listening partner, and the speaker is expected to know this, so the base layer considers all potentially affirmative backchannel utterances to mean *agree*. This holds accordingly for all *agree* verb classes in the base layer.

### 4.2.8 Short negations

Likewise, short utterances that could be either rejection or backchannel (mostly a variety of grunting sounds) should be annotated as *disagree*. A particularly problematic example would be an appropriately pronounced “Hmm.”, because the intonation might be right in the middle between affirmation and rejection. This, too, holds accordingly for all *agree* verb classes in the base layer.

### 4.2.9 Proposal-less questions

We use *ask\_design* when the speaker asks for how something ought to be designed without indicating any design idea of his or her own. This is in contrast to design proposals that are merely formulated as a question.

Example 4.3: Partial episode from Session BA1 (14:10:51–14:10:59) that starts with *ask\_design*. The question starts a 1-minute episode during all of which the pair discusses the return value of a PHP script that is to be called by an external service provider for retrieving information from the system being developed here. The discussion is on the fringe between design discussion and requirements discussion (see Chapter 5), but is classified as *design* because one can see (just not in the example part here) that the external provider has no voice in deciding the interface. In the part shown, B1’s question does not achieve its goal: B2 does not make a proposal – so B1 cautiously formulates one himself.

(1) B1.*ask\_design*

“What should be pass on to them really?”

(see caption) The question addresses only the return value for one particular if case.

(2) B1.*explain\_finding*

<sup>2</sup>[http://en.wikipedia.org/w/index.php?title=Backchannel\\_\(linguistics\)&oldid=565247022](http://en.wikipedia.org/w/index.php?title=Backchannel_(linguistics)&oldid=565247022)

“NOCHANGE is surplus data here.”


B1 continues right away; B2 does not say anything. The script returns the NOCHANGE string, but as it is called frequently, this may be inefficient. This finding justifies the previous question.

### (3) B1.propose\_design

“The question is: Should we return anything at all?”

Three seconds later, during which nothing happened, B1 suggests to maybe not return anything. This transforms his original question into an LO-mode proposal.

## 4.2.10 Restricted disagreement

 Rejections of proposals may be formulated such that the speaker sounds fully certain, perhaps uncertain, likely uncertain, or explicitly uncertain. For instance, in Session CA2 after the statement “*This we could pull out there.*” C2 reacts by saying “*(Hmm, yes.) (..) I’m not sure whether I (.) would pull it out there.*” The example shows that even verbal agreement and disagreement can be hard to discriminate: The first part is grammatically affirmative, but the speaker utters it rather quietly. The second part explicitly expresses uncertainty but has more rejecting character than affirming character and is uttered after some consideration, so we annotate the whole as *disagree\_design* – but clearly the disagreement is far weaker than 100 percent. We will revisit this issue in Section 21.6.4.

## 4.3 Discrimination from similar concepts

### 4.3.1 propose\_design vs. ask\_knowledge

As discussed above, questions that ask for comments on a fresh(!) design possibility are always annotated as *propose\_design*, never as *ask\_knowledge*: First, design aspects have been excluded from *knowledge*, so *knowledge* can never apply. Second, we treat the question as an indirect speech act, that is, we ignore the secondary illocution of asking and annotate the primary illocution of proposing.

### 4.3.2 \*\_design vs. explain\_knowledge/explain\_finding

The general rule for design proposals and their discussion is: *design* takes annotation precedence over both *knowledge* and *finding*. Only justifications (if present) are annotated separately and then use either *explain\_knowledge* or *explain\_finding*. (This holds likewise for justifications of other proposals,

most importantly *step* and *strategy*). However, the following corner cases need separate discussion:

**Packaged design proposals:** Design proposals might be enclosed in a *knowledge* utterance, that is, formulated such that the knowledge transfer aspect appears to be the primary one. A double annotation using both the *knowledge* and the *design* concept should then be used, because either of the two alone would lose an important (rather than just a secondary) aspect of the utterance. See Example 17.2 in the discussion of the *activity* concepts.

**Packaged agree/disagree for a design proposal:** Likewise, agreements and refusals can also be enclosed in a *knowledge* utterance and again a double annotation is appropriate in order not to misrepresent the utterance. See Example 16.2 (for a *step* proposal).

**ask\_design vs. ask\_knowledge:** Queries for obtaining a design proposal are not design proposals, but are still covered by the precedence rule: Annotate them specifically (as *ask\_design*) not generically (as *ask\_knowledge*).

There will be deeper discussion of these rules in the *knowledge* Chapter 16.

### 4.3.3 *propose\_design* vs. *propose\_step/propose\_todo*

As we see, the discrimination of *design* concepts from *explain\_knowledge* is normally straightforward. The same is *not* true (sometimes) for discriminating *propose\_design* from *propose\_step* and from *propose\_todo*. To understand those differences, however, a detailed understanding of the latter's concept classes is required, so we postpone the discussion until the respective Chapter 6 (for *step*) and Chapter 8 (for *todo*).



# Product-oriented concepts: *requirement*

## 5.1 Topic of *requirement* concepts

The *requirement* concepts talk about requirements imposed on the pair's work results from outside the session. Such requirements can be functional (including API requirements), non-functional (including constraints of any kind) and their key characteristic is that the pair cannot change them. In particular, design discussion must be annotated as *requirement*, if and insofar a certain design has been imposed on the pair from outside.

However, requirements are not necessarily known and clear. There are four major cases:

- Requirements known since before the session.
- Requirements recognized during the session, including those merely assumed by the pair, often temporarily, during the session.
- Requirements recognized during the session to be missing.

As all our concepts, the notion of *requirement* was not predefined or taken from an external source, it was developed from phenomena in our data and is fully grounded in them; see the examples below and subsequently.

Example 5.1: Some *requirement* utterances.

(a) **BA1:** B1.propose\_*requirement*

"OK, let's assume they have one hour difference."

A computation depends on time of day, but the pair does not know whether the clocks of the two servers in question will be properly synchronized. The utterance proposes a constraint that may help understand and tackle the problem.

**(b) CA2: C5.remember\_requirement**

*“And the other thing I had understood that we try (.) to make this invisible for the other things by the Facade.”*

Driver C5 refers to an agreement from (apparently) before the session. This agreement imposes the structural requirement that the module being worked on here shall have a Facade structure. For the overall product, this is a design aspect; for this particular session, it is fixed a-priori and is hence a requirement.

**(c) CA2: C5.remember\_requirement**

*“The thing is, <Developer\*> and I yesterday discussed that we change the FeatureProxySet. It shall have a getTableModel.”*

C5 refers to an agreement how to change the present class. He intends to apply this agreement in many of the subsequent steps.

## 5.2 requirement concepts and their properties

As shown in Figure 3.1, we have observed *remember\_requirement*, *propose\_requirement*, *agree\_requirement*, and *challenge\_requirement*.

The following subsections explain how to recognize corresponding phenomena. Example 5.2 provides a short requirement discussion episode.

Example 5.2: Discussion of a missing constraint requirement from Session BA1 (16:25:18–16:25:34). Starting point is the pair’s new finding that the script they are developing will only be able to return the correct results if the clock on the caller’s system (outside the pair’s control) is synchronized with the clock on the local system.

**(1) B1.propose\_requirement**

*“Ideally, they could simply send a timestamp of now. (...)”*

To solve the problem, B1 suggests a requirement that should be imposed on the caller.

**(2) B1.challenge\_requirement**

“But they cannot send a timestamp each time so we can look ‘hey is your clock off?’. They could send every minute or every ten (!!...!!)”

After three seconds of actionless pause, B1 changes his own proposal. He stops in mid-sentence because B2 starts speaking.

**(3) B2.challenge\_requirement**

“They could send a keepalive.”

B2 makes a counter-proposal, to which B1 reacts without any delay:

**(4) B1.agree\_requirement**

“That’s an idea.”

B1 agrees to that third proposal.

### 5.2.1 *remember\_requirement*

The speaker recapitulates a *requirement*, typically because he or she has just become aware (or aware again) that it exists or that it is relevant, sometimes to remind the partner of it or consciously bring up the topic. This is the only concept that uses the verb *remember*, because this case of what could otherwise have been covered by *explain\_knowledge* is special in that the knowledge represents a condition that has arbitrarily been imposed on the pair and overlooking it will unavoidably produce problems (whereas other knowledge can often be substituted).

### 5.2.2 *propose\_requirement*

What superficially looks like *remember\_requirement* may in fact talk about requirements that nobody has yet imposed but that the pair finds necessary in order to proceed. The subsequent discussion of a *propose\_requirement* is more similar to *propose\_design* than it is to *remember\_requirement*, because the pair has freedom to make their own (suitable) decision rather than merely interpret and handle something given; see Example 5.2.

Sometimes, these “requirements” are temporary assumptions that serve to reduce complexity and aid thinking; see (a) in Example 5.1. As such, they are provisional and may later be fixed, changed, or may disappear entirely.

### 5.2.3 *agree\_requirement and challenge\_requirement*

These two concepts can occur in the context of a *propose\_requirement* as well as of a *remember\_requirement*, but their character is different in either case. In the

*propose* case, they talk about how to use a degree of freedom, in the *remember* case, they talk about the truth of facts.

*disagree\_requirement* and *amend\_requirement* could naturally occur as well, but we have not seen them in our data.

### 5.3 Discrimination from similar concepts

All relevant considerations have already been explained:

- Use *remember\_requirement* where externally imposed constraints are concerned and *propose\_design* where the pair's leeway in decision-making is concerned. This could have been (even should have been) externally imposed, but was not; so the pair makes up its mind itself.
- Conceptually, *remember\_requirement* is a specialization of *explain\_knowledge*. Where the former applies, use it, but do not use the latter. See Chapter 16 for details.
- Conceptually, *propose\_requirement* is a specialization of *explain\_finding*. Where the former applies, use it, but do not use the latter. See Chapter 12 for details.



## Process-oriented concepts: *step*

In contrast to the product-oriented concept classes *design* and *requirement* discussed in the previous chapters, the process-oriented concepts talk about how the pair steers its development process (mainly within this particular pair programming session). There are five concept classes that cover three topic areas:

- Utterances regarding tactical behavior, i.e., regarding short-term acts performed currently or in the immediate past or future. These are addressed by the *step* and *completion* concept classes that will be discussed in the present chapter and in Chapter 7, respectively.
- Utterances regarding somewhat longer-term, strategic behavior. These are addressed by the *strategy* and *state* concept classes that will be discussed in Chapter 9 and Chapter 10, respectively.
- Utterances regarding deferring work steps or subtasks in order to perform them at some later time. These are addressed by the *todo* concept class that will be discussed in Chapter 8.

The terms tactical, strategic, short-term and longer-term are vague; lengthy explanations will be required to make them well-defined. The key to telling them apart will be the notion of *strategy*. The explanation, however, works better if we start with *step*.

### 6.1 Topic of *step* concepts


When analyzing pair programming sessions, it quickly becomes obvious that a large fraction of the utterances has to do with deciding the next work step.

This is what the *step* concept class talks about. By *next* work step we refer to activities that can be started right away rather than needing substantial preparation. By *work* step we only refer to process issues, because product issues are already covered by the *design* (or possibly *requirement*) concept class. (And as for *design*, the concepts only address the work step proposal itself; justifications for it will be annotated as *explain\_knowledge* or *explain\_finding*.)

By work *step* (as opposed to some larger unit) we refer to atomic units of work. But what is atomic? Is, for instance, closing a window atomic? Or do you need to go down to the electron-level events in the nervous system that result in contracting a muscle which results in clicking a mouse button which results in closing the window? We found that neither of these is sufficiently relevant to become a base concept – nor is the outcome of any other “objective” definition of atomicness.

It turns out the only useful definition is: A step *is* atomic if a pair member *considers* it atomic. More precisely: If s/he talks about it in a way that represents the step as atomic (behavioristic perspective) – so that the researcher concludes that s/he considers it atomic<sup>1</sup>. In this subjective view of atomicness, the defining characteristic is not that the work step *cannot* be divided at all, but rather the speaker currently views it such that it *needs not* be divided in order to be decided and begun. Once the actual execution of the step has started, the step may turn into something non-atomic that requires further discussion of its innards.

But outside its execution, a step in this sense is a single means to a particular end and a *step* utterance does not place it in the context of further actions. The end may be mentioned or (more often) not.

 In practice, *step* utterances can refer to simple activities such as executing a certain well-understood command on the computer (“*Do a 'save'.*”) or to more complex ones such as performing a manual test (“*We could make an a first test now.*”), discussing an interface to be created, understanding some existing code (“*I would (..) let us have a look at TableModel how that works how it fetches the values.*”), or implementing a particular method.

Note that work steps to be performed later rather than now are represented by *todo* rather than *step*, and work stretches considered non-atomic during their discussion are represented by *strategy* rather than *step*.

Example 6.1: Utterances annotated with *step* concepts and referring to simple actions (such as (d)) or complex activities (such as (b)). Examples (c) and perhaps (a) show cases where a reply is not expected, the utterance has purely informational character.

<sup>1</sup>This conclusion could be wrong if the talking misrepresented the thinking for some reason but we have never found any evidence that this was happening in our sessions.

**(a) BA1:** B1.explain\_finding + B1.amend\_step

*“Errrr, (.) yes, complete nonsense of course; we can search in our Working Set.”*

The pair is searching for calls of a particular function. B1 had started a search of the whole workspace and now finds it will take long. He proposes a more restricted search and starts it.

**(b) CA2:** C5.propose\_step

*“Then (.) (it is so) I’ll first show you what I’ve done.”*

The pair will have to work on code sections worked on by C5 before the session. C5 suggests to bring C2 up-to-date.

**(c) CA2:** C5.propose\_step

*“(OK. Refactor.)”*

C5 has proposed to move a class to a different package. He now scrolls through the Eclipse IDE’s package explorer, finds the class, opens the context menu on it, and makes the utterance. Less than a second later he starts the refactoring from the context menu.

**(d) CA2:** C5.propose\_step

*“Properties.”*

The pair is testing the application. The driver hesitates and the observer steps in with a proposal where to click next.

**(e) CA2:** C5.propose\_step

*“Also change the TODOs above into NOWs.”*

The driver is just tagging a new, empty method as TODO\_NOW. The observer suggests to do the same for the class’ other empty methods, currently tagged TODO.

## 6.2 step concepts and their properties

As shown in Figure 3.1, we have observed *propose\_step*, *agree\_step*, *decide\_step*, *disagree\_step*, *challenge\_step*, *amend\_step*, and *ask\_step*.

### 6.2.1 propose\_step with rationale

A proposal can be complemented by a justification giving its rationale. Just like for *propose\_design*, such explanations should be annotated separately, either

with *explain\_knowledge* or *explain\_finding* as appropriate.

For instance, after the *amend\_step* utterance from Example 6.1, the speaker added



*I thought I had anyway checked out the <\*Working Set\*> only, but (!...!).*

This justifies why he changes his former proposal: His previous assumption that workspace and working set are the same was wrong (*explain\_finding*).

In principle, all other *step* utterances could also come with a rationale, but we have seen only few instances of that. We will revisit the discrimination between *explain\_knowledge* and *propose\_step* in Section 16.3.1.

## 6.2.2 Purpose of making *propose\_step* utterances

By far not all *propose\_step* utterances appear to constitute requests for comments. In particular when made by the driver during HCI activities, *propose\_step* is often merely information about what the speaker intends to do next; Example 6.1 (c) is a good illustration of this type.



Some *propose\_step* utterances quite explicitly call for a reaction from the partner, for instance “*Perhaps you could quickly have a look at it?*” or “*We can close it, right?*”, but in (for example) Session CA2 only 12% of the *propose\_step* utterances clearly had such character.

Overall, the same proposal types apply to *step* than we had explained for *design* in Section 4.2.1. Example 6.2 contains a *step* utterance in LO mode.

Example 6.2: An LO-mode *propose\_step* utterance from Session CA2 (12:58:08–12:58:16). The driver had previously made a design proposal that amounted to doing a refactoring and apparently the pair had no doubts that this was appropriate at least in principle. Whether to actually do it (or what else if not), still needed to be decided, however.

### (1) C2.*propose\_step*

*“The question is whether I dare do it now, whether we dare do it now.”*

The observer asks whether the pair should perform a proposed refactoring or not in so open a manner that he clearly wants a comment from the partner and states no preference of his own. (The utterance is actually a borderline case in between *step* and *strategy*; see Section 9.3.3. The context provides too little evidence of strategy-ness, however.)

### (2) C5.*agree\_step*

*“I would like to dare do it now.”*

The driver interprets the question as a proposal and agrees.


(3) C2.*agree\_step*

“OK.”

The observer agrees as well, effectively adding late a judgment of his former proposal.

Just like for *propose\_design*, if a proposal is phrased as a question, only the primary illocutionary act will be considered in the annotation; see the discussion in Section 4.2.1. A proposal can also be made to contradict a *finding* as in Example 12.6 (2).

### 6.2.3 Reserving time

Sometimes the speaker does not actually mention the intended step, but only asks for the time needed to do it. A typical example is “*Wait a second.*” from Session CA2. The step itself (such as reading or thinking) will then often be done alone. 

### 6.2.4 Imprecise proposals

Proposals can be incomplete, vague, or ambiguous. Even if the observer makes a request or gives an order, the proposal may leave the driver a lot of leeway; see Example 6.3.

Example 6.3: An *amend\_step* episode from Session CA2 (12:18:25–12:18:31). Observer C2 makes a vague proposal more specific when he recognizes that his partner does not act as he had intended. The driver had previously clicked the first ‘error’ entry in the Eclipse IDE’s Problems View. This had sent the pair to an editor view that showed several lines as containing defects. After a short pause, the observer makes the proposal shown. The episode shows that we cannot always decide whether an utterance is indeed an *amend\_step*: We may either assume the second utterance simply delivers additional precision and annotate it as *amend\_step*, or assume the “maybe better” means the observer has indeed changed his mind and annotate as *challenge\_step*. We chose the former, but not without writing a coding memo to make a note of the problem.

(1) C2.*propose\_step*

“Point the, point the mouse, please.”

The observer asks the driver to hover the mouse over one of the defects shown in the editor in order to display the tool tip that will contain the error message. He does not say which defect, though. The driver chooses the last defect shown and its message opens.

(2) C2.amend\_step

“No, there; maybe better hover that one.”

After the pair has looked at the message for about a second, the observer points his finger to a different error icon and makes his former proposal more specific.

### 6.2.5 *decide\_step* vs. *agree\_step*

See the discussion in Section 4.2.4. We have seen *decide\_step* only once.

### 6.2.6 *amend*, *challenge*, or *disagree* one’s own proposal

As described for *design* in Section 4.2.6, it is not at all unusual if a speaker supplements his or her own proposal with further detail. This can even happen if the partner has not said anything in between to make sure the partner acts or understands as intended or to supply new thoughts formed in the meantime.

Continued thinking may also lead to challenging or even sometimes revoking (*disagree\_step*, see Example 6.4) one’s own proposal. As Example 6.3 shows for *amend* vs. *challenge*, it is not always easy to tell these cases apart. Similar phenomena should be expected in most other HHI concept classes as well.

Example 6.4: Episode from Session BA1 in which B2 takes back his own step proposal after he recognized that the step would not lead to the desired result: The output is basically timestamps of certain database entries, but there is no reason yet why they would change. The episode illustrates two things: First, findings need not be triggered by external stimuli. Second, spontaneously changing one’s mind is often evidence of a finding.

(1) B2.propose\_step

“Let’s do a refresh again.”

The observer suggests to refresh the browser view, which would call the PHP script again that the pair is currently editing.

(2) B2.disagree\_step + B2.explain\_finding


“Ah, no, we haven’t done Set yet.”

After about one second, the observer takes back the suggestion and explains why.

### 6.2.7 Indicating agreement vs. indicating attentiveness

See Section [4.2.7](#).

### 6.2.8 *ask\_step*

Questions that ask for the next work step without proposing one are annotated as *ask\_step*. Such questions can be general or pertain to a particular context, and they can be new or request recapitulation of something previously discussed. The question “*What did we want to open?*” from Session [ZB7](#) that talks about the next file to be analyzed is of the context-and-recapitulation type. 

## 6.3 Discrimination from similar concepts

We will talk about discrimination of *step* concepts from *knowledge* concepts and from *design* concepts only. Discrimination from *todo* concepts and *strategy* concepts is postponed until those classes have been explained.

### 6.3.1 *\*\_step* vs. *explain\_knowledge*/*explain\_finding*

Proposals for the next work step can be interpreted as knowledge. The general rule for step proposals and their discussion is: *step* takes annotation precedence over both *knowledge* and *finding*. Only justifications (if present) are annotated separately and then use either *explain\_knowledge* or *explain\_finding*. (This holds likewise for justifications of other proposals, most importantly *design* and *strategy*). For additional discussion see Section [12.3.3](#).

### 6.3.2 *propose\_step* vs. *propose\_design*

Quite obviously, realizing any proposed design aspect requires certain work steps at some point. Therefore, design proposals often go hand in hand with work step proposals. Since in such contexts the *design* concept class is the more specific one and double annotations are generally frowned upon in the base layer, *propose\_design* takes precedence and *propose\_step* should then be avoided as illustrated by Example [6.5](#). Note that design proposals are often formulated in procedural wording and will then superficially look like step proposals; see Example [6.6](#).

Example 6.5: Episode from Session **BA1** (14:51:32–14:51:43) that illustrates both the difference and the interplay between *propose\_design* and *propose\_step*. The pair wants to insert calls to `updateFriendsLastChangeTime` in all relevant places in the code. They neither yet know where these places are nor whether they should previously refactor.

(1) *B2.propose\_design*

*“And then also the region when we delete again.”*

The driver has just completed calling `updateFriendsLastChangeTime` in method `setFriendship`. Now the observer suggests adding the same call into several delete operations.

(2) *B1.amend\_design*

*“OK, then we can either attach ourselves to this Hook, (!!...!!)”*

The driver makes the idea more concrete. While speaking, he scrolls to method `deleteFriendsIDsFromMemcache` which he deems one of those operations.

(3) *B2.propose\_step*

*“Let’s see how that looks where the User is deleted.”*

The observer interrupts the driver’s utterance and suggests looking at the method responsible for deleting a User. Whether this pertains to B1’s utterance at all or not we do not know.

(4) *B1.challenge\_step*

*“We could also first look for all the places this is used.”*

B1 immediately dissents by making yet another proposal. While speaking, he points the cursor to method `deleteFriendsIDsFromMemcache`.

Steps 1 and 2 show how design proposals imply work steps. Step 3 shows how *propose\_step* may imply the refusal of a previous proposal (*disagree*), but since such refusal can rarely be recognized unambiguously, the base layer suggests not to annotate it.

Example 6.6: A design proposal that directly leads to action from Session **BA1** (13:53:56–13:54:08). No *propose\_step* is used in such cases.

(1) *B2.propose\_design*

*“We could simply, er, convert that thing in a, in a normal id, I mean, code\_to\_id and then id\_to\_code again.”*



The observer suggests how to modify a certain if-condition that tests properties of an external call argument.

(2) B1.agree\_design

“OK.”

The driver agrees and immediately starts executing the proposal.

### 6.3.3 *propose\_step* vs. *ask\_knowledge*

Questions of type “*Should we do X next?*” or “*Should we next do X or rather Y?*” mention possible steps. They are therefore annotated with the more specialized *propose\_step* rather than the more general *ask\_knowledge* (and with the closed-question verb *propose* rather than the open-question verb *ask*).

Although this sounds like a hard and fast rule, there are difficult cases:

**Implicated action announcements:** The question

*“Do you know how the (.) how I access the function for changing a method?”*



(from Example 16.2) superficially asks for how to access a certain IDE functionality and should be annotated with *ask\_knowledge*. However, the question also implies what the speaker intends to do next, which warrants a *propose\_step*. Indeed the partner hears both of these aspects and answers “*That doesn’t get you anywhere.*” The base layer suggests to double-annotate such cases with both *ask\_knowledge* and *propose\_step*. More generally, questions for knowledge, in particular procedural knowledge, may imply subsequent *steps*. Section 21.6.1 will specify that such implications should be made explicit whenever they are recognized.



**Remind of a proposal by asking:** Sometimes a formerly proposed (and perhaps discussed) step that was not executed is proposed again later. For various reasons, this may take question form, for instance because the speaker is not sure of the exact proposal anymore:

*“OK, we said delete and, umm, what else?”*



Again, both aspects are strong enough here and so the base layer suggests to annotate such cases with both *ask\_knowledge* and *propose\_step*; see Section 21.6.3 for details on handling repetitions.

### 6.3.4 *ask\_step* vs. *ask\_knowledge*

Any open question specifically for a *step* will be annotated specifically with *ask\_step*, not generically with *ask\_knowledge*.

### 6.3.5 *ask\_step* vs. *ask\_design*

For a question such as



*“OK, what did we want to do?”*

it will often be unclear (even when consulting the context) whether it aims at *step* or at *design*. *ask\_step* as the more general concept may then be more appropriate. Annotating both concepts may be appropriate to express ambiguity that appears *intended* by the speaker.

### 6.3.6 *disagree\_step* vs. *explain\_knowledge/explain\_finding*



The rejection of a proposed *step* needs not be explicit, it can be wrapped in a knowledge transfer as in the *“That doesn’t get you anywhere”* in Section 6.3.3 above. Both aspects of this indirect speech act are relevant in the base layer perspective, so we annotate both the primary and the secondary illocutionary act, although which one is primary may be different from case to case: the knowledge transfer (*explain\_knowledge* or *explain\_finding*) or the rejection of the proposal (*disagree\_step*). The same consideration sometimes applies to *challenge\_step* as well.

### 6.3.7 *amend\_step* vs. *explain\_knowledge/explain\_finding*

Just like for *propose\_step*, we annotate a justification or rationale separately for *amend\_step* if one is supplied and use either *explain\_knowledge* or *explain\_finding* as appropriate.

## Process-oriented concepts: *completion*

Where the *step* concepts talk about an individual work step, the *completion* concepts talk about whether (or to what degree) the step has been fully and adequately performed and completed or not.

The *strategy* and *state* concept classes form a corresponding pair for compound work.

### 7.1 Topic of *completion* concepts

A *completion* utterance refers (explicitly or implicitly) to a work step. It talks about the evaluation of the degree or quality of fulfillment of the step's execution or of some intermediate stage of the step's execution.

The step may have been addressed by a previous *step* dialog episode or (frequently) not.

Example 7.1: Utterances annotated with *completion* concepts, all from various stages of Session **BA1**

(a) B1.*explain\_completion*

"I still do not like this very much."

The driver says this after 15 minutes of work that started with an *amend\_design* about adding functionality to a certain method. To avoid side effects, the pair had decided to check where and how that method was being called, which had led to analyzing many spots in the code neither of the two were familiar with. The pair had suggested a number of modifications but performed only few of them. The *explain\_completion* utterance evaluates the success with respect to a step that was never formulated explicitly.

**(b)** B1.*explain\_completion*

"I'd say we now have the problem conceptually (.) narrowed down and solved."

Uttered after a three-minute-long discussion that had never been proposed but had started spontaneously after a *propose\_design*.

**(c)** B1.*explain\_completion*

"Well, that wasn't so bad really."


After modifying a number of *phpDocumentor* comments, the driver leans back in his chair and makes this utterance. The lean-back gesture underlines the interpretation that the speaker considers the current work step, which had never been verbalized, done.

**(d)** B2.*challenge\_completion*

"But we haven't done anything yet!"


Answer to (c). The observer points out that the actual editing steps still have to be made. This is not *disagree\_completion* because the speaker offers his own, alternative evaluation.

## 7.2 completion concepts and their properties

 As shown in Figure 3.1, we have observed only *explain\_completion*, *agree\_completion*, and *challenge\_completion*, but of course a researcher may add the remaining ones as soon as they are encountered.

Most of the discussion about these three verbs from the previous concept classes applies. We offer only three additional remarks.

### 7.2.1 Short evaluations


 In contrast to the cases shown in Example 7.1, most *explain\_completion* utterances we have seen were short and simple (often just "OK.") in particular from the driver to announce the end of editing steps.

### 7.2.2 Indirect evaluations

An *explain\_completion* can be indirect by stating that something needs *not* be done.

An example is the “*Well, no need to go up here any more*” after four minutes of work in Session **BA1**. 

### 7.2.3 Evaluation of quality

The base layer does not discriminate evaluations of the quality of a work result from less differentiated statements; both are annotated simply with *explain\_completion*. Separate elaboration may be annotated with *explain\_finding* (or perhaps *explain\_knowledge*), but higher concept layers may need to add much more differentiation. 

## 7.3 Discrimination from similar concepts

*completion* applies to *step* in the same way that *state* applies to *strategy*. Their discrimination is easy if the work type has previously been discussed, because then the decision in favor of either *step* or *strategy* has already been made. For *completion* or *state* utterances of *implicit* steps and strategies, however, it can be subtle.

The distinction of a proposed evaluation (*explain\_completion*) and its justification (which should then be annotated with *explain\_finding*) is harder than for *design* or *step* because a *completion* proposal itself may have so little body that it becomes mostly implicit (as in Example 7.1 (c)); see the discussion in Chapter 12.



## Process-oriented concepts: *todo*

### 8.1 Topic of *todo* concepts

Like *step*, the concept class *todo* talks about individual work steps. While *step* talks about the current or next step, *todo* utterances serve to make note of steps that at least one pair member intends to delay until either some indefinite time or some time beyond the end of the current session.

Example 8.1: *todo* episode from Session CA2. No particular time for execution nor a particular ordering of steps is scheduled.

(1) C5.*explain\_knowledge*

*“There’s a test for it.”*

After the pair discussed changes to a class, C5 remarks that a corresponding test class exists; he opens the test package in the IDE’s package explorer to validate or illustrate this.

(2) C5.*propose\_todo*

*“We must remember, uh?”*

Without break, he adds that the pair should not forget to adapt the test class at some appropriate later time.

Example 8.2: Episode from Session CA2 (12:20:27–12:20:46) in which the pair agrees on a *todo* in the midst of a *strategy* discussion. Deferring the step helps avoiding discussion complexity excess.

(1) C2.*amend\_strategy*

*“That means, I guess we should do that now, perhaps we should check in first?”*

C2 has just taken the driver role and extends the two-part strategy (see Section 9.1) he had just proposed by another step, to be done first: checking in the code. While he speaks, he turns towards his partner.

(2) *C5.explain\_knowledge + C5.amend\_strategy/C5.disagree\_step*

*“Um, I would rather(!...!). There are few tests there and I would rather not check in (!!...!!).”*

After a two-second pause, C5 points out there are few unit tests for the code. His primary intention appears to be to refuse the check-in step, but since he had agreed to the overall strategy beforehand, *disagree\_strategy* would not be an appropriate interpretation; see Section 21.6.2.

(3) *C2.propose\_todo*

*“Then let’s write some tests.”*

C2 interrupts and proposes to write tests without saying when. The subsequent events show that C2 does not mean this to be a *step* proposal.

(4) *C5.agree\_todo*

*“Yes, we should certainly do that too.”*

C5 agrees but still specifies no time. The “too” sounds like “but not now”.

(5) *C5.amend\_strategy*

*“Er, rather not check in before I’ve tried the GUI.”*

After his agreement, C5 completes his formerly interrupted statement. Only now we can see his original intention to do an *amend\_strategy* by adding one step.

(6) *C2.agree\_strategy*

*“OK, then let’s try the GUI.”*

C2 agrees and the pair next tests the GUI.


## 8.2 *todo* concepts and their properties



We have observed relatively few *todo* utterances. As shown in Figure 3.1, only the types *propose\_todo* and *agree\_todo* occurred, but of course a researcher may add the remaining ones as soon as they are encountered.

How we define *todo* phenomena so far is:



- The proposals were not very concrete and never specified a time.
- Some, however, may specify the conditions under which the step would be needed; it would not be needed if those conditions never hold. A straightforward case is Example 8.3. Although these utterances have a strong component of *explain\_knowledge* or *explain\_finding*, we interpret them as *propose\_todo*: “I propose we perform step X in case condition C materializes.”
- Our *todo* phenomena all had the form of declarative sentences, such as “Need to mind that, too.” 

Example 8.3: Episode from Session CA2 (12:43:03–12:43:08) involving a conditional *todo* that points out the consequences of a pending decision.

(1) C2.*decide\_design*

“I guess it mustn’t go into Abstract.”

Currently, class `FeatureAttributeConfigurationProxy` is a subclass of `AbstractFeatureConfiguration` (which the speaker calls `Abstract`). The pair discusses whether this should remain true and agrees the inheritance would be appropriate only if the superclass had less functionality. C2 decides to go that way: remove some functionality from the superclass.

(2) C5.*amend\_design/propose\_todo*

“Well, OK, then we have to implement this in very many places.”

C5 agrees but points out laborious consequences: the functionality removed from the superclass will eventually have to be inserted into many existing subclasses.

## 8.3 Discrimination from similar concepts

The discrimination of *design*, *step*, and *strategy* on the one hand (current planning) versus *todo* on the other (markings for future planning) is likely important for the control of a pair programming session, so we should discriminate them carefully. Since *strategy* is complex and has not yet been introduced, we postpone its discussion until Chapter 9 and only discuss discrimination from *design* and *step* here.

### 8.3.1 *propose\_todo* vs. *propose\_step*

As said before, *step* talks about the next work step, *todo* talks about a work step at some later time. However, pair programming dialog utterances convey

much of their information implicitly and so actual *step* and *todo* utterances may look rather similar if they are purely claims of necessity.

These properties suggest a *propose\_step* utterance:

- The utterance invites to action (“Let’s...”) without specifying a time.
- It uses terms such as “now” or otherwise indicates immediacy.
- It followed by the execution of a corresponding step.

These properties suggest a *propose\_todo* utterance:

- The utterance uses future tense.
- It uses terms such as “later” or “eventually” or otherwise indicates deferral.
- It states a particular later time or condition when to perform the step. Statements of time are rare, because the speaker will rarely make the effort of deciding a suitable time unless s/he is formulating a strategy. So if a time is specified, always check whether you are seeing e.g. a *propose\_strategy* (etc.) rather than a *propose\_todo*.
- The purpose of the utterance appears to be to get rid of an objection quickly, as in item 3 of Example 8.2 that appears to “protect” the speaker’s original suggestion to check-in the code by calming down the partner, even though that requires interrupting the partner in mid-sentence.



Even if these symptoms are missing or ambiguous, reliance on context information will normally allow to discriminate *todo* utterances reliably. Note the remark on double annotations with *todo* and *design* concepts below.

### 8.3.2 *propose\_todo* vs. *explain\_knowledge/explain\_finding*

As discussed before in the context of *design*, *step*, and *strategy*, *todo* utterances have the character of *explain\_knowledge* (or *explain\_finding*) but take precedence when annotating, because *todo*, being more specialized, is more informative. The class exists because of its plausibly important role for how pair programming sessions usually work.

### 8.3.3 *propose\_todo* vs. *amend\_design/propose\_design*

An amendment for a design proposal can be accompanied by, or even entirely consist of, a statement of future work the proposal entails. In both of these cases you should annotate both concepts, *propose\_todo* as well as *amend\_design*, because both aspects are separately important for the session; see Example 8.3.

This idea will occasionally apply to other *design* concepts as well, in particular to *propose\_design*.



## Process-oriented concepts: *strategy*

Where the *step* concepts talk about an atomic work step, the *strategy* concepts talk about planful compound work.

### 9.1 Topic and typology of *strategy* concepts

The *strategy* concepts address those parts of a dialog (or monolog) in which the pair attempts to plan, decide, or coordinate complex stretches of work with respect to some higher-level goal (see Section 9.3.3). A strategy in this sense is an idea or approach for longer-term, planful action.

As all other concepts in the base layer, this notion of strategy (which we will explain in a lot more detail below) was not taken from the extensive existing literature on that term but rather fully extracted from our observations in pair programming sessions only. This extraction was a lengthy process during most of which 'strategy' appeared to be a confusing and untidy concept. The process became satisfactory only after we started to discriminate three types of strategies: OWP, DPR, and EXS.

#### 9.1.1 OWP: Organizing Work Packages

The current task (or a part thereof) is divided into specific (and typically non-atomic) pieces, which we call subtasks. An OWP-type *strategy* usually specifies or implies a certain work-sequence order of the subtasks.

#### 9.1.2 DPR: Determining Procedure Rules

Rather than decomposing the task explicitly into an ordered set of more-or-less explicit subtasks, the pair speaks about rules or guidelines intended to help devise or execute subsequent work continually. These rules often aim at

simplification, for instance by excluding certain types of work step (such as writing tests) or by deciding to provisionally ignore particular complications from design consideration.

### 9.1.3 EXS: Expanding a *step* into a *strategy*

What was originally a *step* or *design* may be expanded, by the same speaker or the partner, into a *strategy* by adding an additional *step* to be performed and viewing both together in light of a higher-level goal. Often this takes the form of remarking that after the previous proposed action, only the additional activity X will be needed to reach the desirable state Y.

These types help a lot in identifying and understanding *strategy* utterances, but despite the coarseness of this classification it can be difficult to apply. We therefore add two additional properties of *strategy* utterances: The type of representation and range.

### 9.1.4 Extensional vs. intensional representation

A strategy can be verbalized in extensional manner, by mentioning actions separately and explicitly, or in intensional manner, by subsuming actions under a single abstraction that represents an action superconcept or a criterion.


Most commonly, OWP strategies and EXS strategies are extensional while DPR strategies are intensional, but neither is mandatory. For instance, the following utterance from Session ZB7 could be considered an intensionally expressed OWP *strategy* proposal: “Um, should we here, um, these instructions on executing, um, XPetstore, look it through and follow it?” Superficially, this looks like two steps: looking through instructions; following instructions. Semantically, however, the utterance suggests a longer sequence of steps (those given in the instructions) without explicating them. The above example could also be considered a DPR *strategy* utterance and this holds for all intensional OWP proposals we have seen so far.

### 9.1.5 Range

A *strategy* can have a mid-term or long-term planning horizon. A long-term *strategy* pertains to all of the rest of the current session or beyond. A mid-term *strategy* pertains only to some shorter stretch of intended work. There is no such thing as a short-term strategy. This would either be a *step* (that is, considered atomic) or we would call it mid-term – even though its actual execution might take less time than a lengthy *step*. It is often hard to tell the range of DPR strategies.

### 9.1.6 Mixed types

Note that these properties do not allow classification in all cases. A single *strategy* utterance may have extensional as well as intensional parts, there are ambiguities between intensional OWP and DPR utterances, and the range may be hard to determine, in particular for DPR utterances.

For the base layer, this is not a problem because the properties serve only to understand the nature of possible *strategy* utterances better. They are primary characterizing attributes. Future research, however, may want to address these topics. In that case, our considerations here may serve as a starting point or be ignored. Here are some (mostly isolated) examples of *strategy* utterances. 

Example 9.1: Examples of *strategy* utterances from all three sessions.

(a) **ZB7:** Z19.*ask\_strategy*

*“How do we start?”*

At the start of the session, Z19 asks this very open question, which sounds more like *propose\_step* than *propose\_strategy*. But since the partner answers with a strategy proposal, the utterance effectively turns into one of this class.

(b) **ZB7:** Z20.*propose\_strategy* (OWP, long-term)

*“I would ignore the EmailSpy for now. And modify the the XPetstore only.”*

Reply to (a). Z20 proposes two steps and their order, which covers the whole of the session.

(c) **ZB7:** Z20.*propose\_strategy* (DPR, mid-term)

*“Know what we do? We use our existing EmailSpy for debugging.”*

The driver proposes a testing approach that involves using a class the pair had developed in a previous session for a different purpose but that will be helpful for detecting possible failures.

(d) **CA2:** C5.*propose\_strategy* (DPR, mid-term)

*“And I won’t much adapt any old tests from any old people.”*

The driver formulates a radical avoidance strategy for coping with multiple subtasks he deems too difficult and/or too large, namely to not solve any of them at all.

(e) **BA1:** B1.*propose\_strategy* (EXS, mid-term)

*“And then we only need to insert this thing in all spots where (friends change).”*

B1 extends a *step* proposed by his partner by another step and implies the two together will reach a substantial goal, which makes it a *propose\_strategy*.

## 9.2 *strategy* concepts and their properties

As shown in Figure 3.1, we have observed *propose\_strategy*, *agree\_strategy*, *decide\_strategy*, *disagree\_strategy*, *challenge\_strategy*, *amend\_strategy*, and *ask\_strategy*.

### 9.2.1 Proposal mode

Besides the properties explained in the previous section, the proposal modes introduced for *step* in Section 4.2.1 apply to *strategy* as well and awareness of the modes may help identify *strategy* utterances correctly. For instance, proposals (b) to (e) from Example 9.1 all have mode PI. Keep in mind that the actual formulations may be rather indirect. We have not yet observed a *propose\_strategy* in mode OE.

### 9.2.2 Proposals with alternatives

As all *propose* phenomena, strategy proposals may specify several alternatives. Take this example from Session CA2: *“Then we should do that before or after. But not at the same time.”* This talks about when to perform certain restructurings in the code. It formulates two alternative possibilities and excludes a third; see Example 9.3 for its context.

### 9.2.3 *decide\_strategy* vs. *agree\_strategy*

The same discussion applies as in Section 4.2.4.

### 9.2.4 Secondary issues

A *propose\_strategy* may include aspects not directly related to solving the task at hand, e.g. related to session breaks (for whatever reason).

Example 9.2: Episode from Session CA2 (12:58:55–12:59:38) in which a strategy is discussed for a number of dialog steps.

(1) C5.*explain\_knowledge* + C5.*propose\_strategy*

*“But our (.) stand-up starts in two minutes anyway. And when we go to lunch then. That means we have time to think it (~~).”*



C5 remarks on a constraint and then rudimentarily formulates a strategy. “it” is the content of the pair’s previous discussion.

(2) *C2.explain\_standard of knowledge*

*“You want to have lunch right after the stand-up and stop this, stop this now?”*

C2 interrupts C5 to validate his understanding.

(3) *C5.propose\_strategy*

*“I want to have lunch right after the stand-up.”*

C5 interrupts C2 (and finishes before him!) to repeat and clarify a part of his previous proposal. Section 9.3.2 will explain why this is not a *propose\_todo*. Section 21.6.3 will explain how to handle repetitions.

(4) *C5.amend\_strategy*

*“No, I want to continue after lunch (~would) with this work task.”*

Right after C2 finishes, C5 details his proposal. (The pair then discusses a minor matter we skip here.)

(5) *C2.agree\_strategy*

*“OK, I would have liked I guess (.) to continue a bit. But OK.”*

C2 reluctantly acquiesces to the proposal.

(6) *C5.explain\_state*

*“As for checking in and so on, I think we are at a point where we can take a break.”*

C5 explains where he sees the work with respect to the goal. This is more an assessment than a proposal.

(7) *C5.agree\_state*

*“Of course, that’s true.”*

This utterance agrees to the proposal as much as to the assessment.

### 9.2.5 Forms of *amend\_strategy*

An *amend\_strategy* utterance adds detail to an existing proposal either by adding one or more steps to a proposal (typically of type OWP, such as in Example 8.2 (1)) or by making existing steps (for types OWS and EXS) or rules (for type DPR) more concrete or more detailed.

An example for the latter occurred shortly after the episode of Example 9.2. C5 elaborates his lunch proposal by saying



*“To go on here (.) I think it’s good to reflect for half an hour in your head how really (..) to go the way. Or should we better take your cautious approach?”*

### 9.2.6 Distinguishing proposals: *amend*, *challenge*, *propose*

Utterances in discussions of extensional OWP strategies often refer to individual pieces of the strategy only, rather than the whole. This makes it hard to discriminate *amend* from *challenge* from *propose*. The following heuristics may help:

- If the speaker signals to basically agree with the strategy and only adds or details individual pieces or aspects, use *amend\_strategy*.
- If the speaker refuses important pieces of the strategy and replaces these constructively, use *challenge\_strategy*. Without replacement, this amounts to refusing the strategy overall and one should annotate *disagree\_strategy*.
- If the speaker singles out one piece and elaborates it (often by turning it into multiple smaller pieces) while ignoring the remaining pieces, there are two possibilities: If the elaborated piece appears to replace all of the previous *strategy*, this is *challenge\_strategy*. If the remaining pieces appear to keep their previous role, it is *amend\_strategy*.

In practice, however, it may be hard to decide whether pieces are really ignored or merely not mentioned, and there may be neither symptoms of agreement nor symptoms of disagreement.

An instance of this problem is Example 8.2 which is preceded by a two-step OWP proposal: 1. add method, 2. review dialog. C2 suggests another step to be done first (check in) and the remainder discusses whether yet another step (try GUI) should be added still before that. In the end, however, it is unclear whether the original steps are still part of what would by now be a four-piece plan (1. try GUI, 2. check in, 3. add method, 4. review dialog) or only the new steps remain (1. try GUI, 2. check in). If the first strategy is fully executed, it can be considered intact, but it may never come to that, for many reasons.




The base layer cannot solve this problem, but its concepts help to notice such phenomena and provide a starting point for researching their meaning and relevance.


### 9.2.7 *ask\_strategy*

If (and only if) a query for a strategy does not imply a proposal, we should annotate *ask\_strategy*. As the term “strategy” is hardly ever used in such questions (and in strategy discussion in general), it will often be unclear whether the question aims at a *strategy* or rather at a *step*; see again Example 9.1 (a). We will discuss this discrimination further in Section 9.3.7.

### 9.2.8 *agree\_strategy*

The *agree\_strategy* utterances we have seen were unambiguous; Example 9.2 (5) was the closest we saw to an ambiguous one. They tended to be quite short (e.g. “OK”) rather than longer, although longer ones occur as well, e.g. “OK. *Good. Let’s do that.*” 

### 9.2.9 *disagree\_strategy*

We have seen only one instance of *disagree\_strategy*: In Session CA2, at one point C2 proposes “*In principle, we could for now only change TableModel and leave FeatureProxy as it is. We could.*” (DPR: a rule to avoid touching FeatureProxy for some time) and the partner remarks 

*“I don’t think we need to change the FeatureProxy.”* 

If this is correct, the strategy loses its basis, so the utterance should be considered a *disagree\_strategy*. If we assume the speaker is not quite sure whether FeatureProxy needs change or not, we should annotate *propose\_hypothesis* as well; see Chapter 13, in particular Section 13.2.2.

## 9.3 Discrimination from similar concepts

### 9.3.1 *\*\_strategy* vs. *explain\_knowledge/explain\_finding*

Strategy proposals contain and represent knowledge. As a general rule in the base layer for strategy proposals and their discussion (*agree\_strategy, challenge\_strategy, amend\_strategy, disagree\_strategy, decide\_strategy*), *strategy* takes annotation precedence over both *knowledge* and *finding*. Justifications (if present) are annotated separately (that is, in addition to the *strategy* concept) and then use either *explain\_knowledge* or *explain\_finding*. (This holds likewise for justifications of other proposals or evaluations, most importantly for *design* and *step*).

### 9.3.2 *propose\_strategy* vs. *propose\_todo*

A *propose\_todo* utterance remarks that *one* step needs to be done at *some* (usually undetermined) later time. In contrast, a *propose\_strategy* of type OWP or EXS

embeds at least two steps in a definite ordering and usually also suggests to start that work now.

### 9.3.3 *propose\_strategy* vs. *propose\_step*

In Section 9.1 we have defined strategy as an idea for planful action. This idea is formed in a creative act that combines prior knowledge, assumptions, and recent insights with an understanding of the goal hierarchy, the pair's capabilities and limitations, and the current situation. The idea aims at producing *advantage*: a solution for how to achieve a higher-level goal at all or achieve it more easily than previous ideas. Reaching a higher-level goal is beyond a step-sized goal insofar it is an emergent property arising from multiple successful steps in combination.

In contrast, a *step* is considered (by the speaker) something simple: it has no interesting structure and appears given rather than inventive. So it might seem as if the discrimination of *step* and *strategy* ought to be easy – and it often is, except when the following complications intervene.

#### 9.3.3.1 Recycled strategies

The creative act required for inventing a strategy needs not have happened right before: The strategy's idea may be a reused one, previously invented by the speaker or even a third party (think of practices such as test-driven development). This is a potentially important difference, but as the creative act as such is not often observable, the base layer prescribes to encode proposals of recycled strategies just like fresh ones: as *propose\_strategy*.

#### 9.3.3.2 *step* with forward reference

Some proposals of work steps appear to imply that choosing this step will make subsequent work easier. If that subsequent work and its connection to the step are explained, such utterances are *propose\_strategy*.


If, however, the reference is vague (e.g. a mere “*Let's first...*”), we call this a *step with strategic character* and annotate *propose\_step*. Beware of hasty judgment: Utterances such as “*Shall we tackle the simple cases first?*” will often have implications that reach much farther, because they formulate a rule and are therefore in fact DPR-type *propose\_strategy*.

#### 9.3.3.3 *steps* aiming at advantage

If the benefit intended by a proposal is evident, that does not mean it must be a *strategy*, because the benefit may not be advantage in the above sense of reaching a higher-level goal. Most *steps* with strategic character will have

evident benefit as well, but they are still atomic and hence still steps; see Example 6.1 (b).

### 9.3.3.4 Multi-part proposals not forming a strategy


An utterance such as “*Let’s start the program so we can test.*” may look like *propose\_strategy* at first: One might think there are two steps (run, test) mentioned here and they are integrated into a whole, so this could be a strategy? However, there is no inventive act here, no advantage needed to be devised, so this is simply *propose\_step*. Furthermore, the illocution of the utterance (the speaker knows he is talking to a professional programmer!) is only “*Let’s test*” so there really is no pair of steps at all. 

### 9.3.3.5 The creative act is invisible

The inventive act is a nice theoretical criterion for a strategy, but is not operationalizable: First, you cannot observe the actual creative act at all and only sometimes will get to see symptoms of it. Second, the inventive act may not even have happened in the present session; see Section 9.3.3.1 above.

### 9.3.3.6 Lowly creative acts

The creative act may be quite simple, relying far more on simply paying attention to constraints rather than on producing a new idea (as in Example 9.2 (1)).

This creates borderline cases in which *step* and *strategy* become a matter of taste.  Researchers may develop additional heuristics suitable for the data or their research question in this respect, but should spend most of their energy on discriminations that are *important* for their work.

## 9.3.4 *propose\_strategy* vs. *propose\_design*

In Section 4.3.3 we have discussed that design proposals often imply (or come along with or take the form of) step proposals. The inverse happens for *strategy*: Some pieces of a (typically OWP) *propose\_strategy* may in fact be design proposals. In such cases, the respective sub-utterances should additionally be annotated with *propose\_design* or *amend\_design*. The same rule may then also apply during the subsequent discussion of the strategy proposal.

Worse, the discussion may turn from a strategy discussion entirely into pure design discussion, which is important for the session and so should be reflected in the annotations by stopping to use *strategy* concepts and using only *design* concepts. We have not seen such a case so far.

### 9.3.5 *agree\_strategy* vs. *agree\_knowledge*

If P1 makes a strategy proposal and provides justification and then P2 agrees monosyllabically, it is unclear whether the agreement pertains to the proposal, only parts of the proposal, the justification, or several of these. You should normally just annotate *agree\_strategy*, except

- if the next few dialog steps show it was in fact an *agree\_knowledge* for the justification only or
- ✍ • if knowledge transfer is a focus of your investigation.

In the latter case, a double annotation may be the safest route. See Example 9.3 for a complicated case in which the proposal was actually a *challenge\_strategy* and the justification comes separately.

Example 9.3: Episode from Session CA2 (12:58:29–12:58:52) containing an ambiguous affirmation: Utterance 8 may pertain to utterance 5 (unlikely) or 7 or both. C5 leans forward towards the display and sometimes turns toward C2; C2 leans back in his chair.

(1) *C2.propose\_strategy*

*“Then we should do that before or after. But not at the same time.”*

Two alternatives are now on the table, plus a third marked as bad.

(2) *C2.propose\_hypothesis* + *C2.explain\_knowledge*

*“Cause I fear that for this work task that we will exceed the time anyway, (.) because it will be fairly complicated.”*

After four seconds of silence, C2 makes a prediction of how fast the further process will go and provides a justification.

(3) *C2.decide\_strategy*

*“Maybe one does it after?”*

After another four seconds of silence, C2 suggests (in mode LO) a decision himself.

(4) *C2.explain\_standard of knowledge*

*“But I’m not sure myself either.”*

Another two seconds later he emphasizes he does not trust his own judgment in this matter; see Chapter 14.

(5) *C5.challenge\_strategy*

"I would do it (before)."

C5 prefers the other alternative.

(6) *C2.agree\_strategy*

"Hmm, yea, I (!...!)"

C2 half-heartedly changes his mind.

(7) *C5.explain\_knowledge*

"Because if we plan it for later, than, then, er (.), then it won't happen."

C5 justifies his preference with experience.

(8) *C2.agree\_knowledge/C2.agree\_strategy*


"OK?"

The intonation clearly shows doubt. We cannot decide whether this just repeats the previous affirmation or expresses (half-)agreement with the most recent explanation.

### 9.3.6 *ask\_strategy* vs. *ask\_knowledge*

Any open question specifically for a strategy will be annotated specifically with *ask\_strategy*, not generically with *ask\_knowledge*.

### 9.3.7 *ask\_strategy* vs. *ask\_step*

We have now claimed several times that we can often decide between *ask\_strategy* and *ask\_step* only by considering the answer, not by looking at only the question itself. Unfortunately, this is still oversimplifying the matter, because the respondent may have misunderstood the question or may even have intentionally misinterpreted it. Further context both before and after the utterance can often disambiguate such cases, but if the asker sits through a misunderstanding quietly, we may never know it. 





## Process-oriented concepts: *state*

### 10.1 Topic of *state* concepts

The *state* concepts represent utterances regarding the status or degree of completion of working through the steps formulated or implied by a strategy. The relationship between *strategy* and *state* is equivalent to the relation between *step* and *completion* (Chapter 7). Like with the step referred to by a *completion* utterance, the strategy referred to by a *state* utterance needs not have been discussed or mentioned before, because the pair may be jointly following an “obvious”, tacit strategy, the driver may have had one in mind for a while without mentioning it (for whatever reason), or a sequence of steps may be recognized as a strategy only after executing some of them. Therefore, *state* utterances pertain to any evaluation of process progress above the step level.

Example 10.1: Some *state* utterances.

(a) ZB7: Z20.*explain\_state* + Z20.*disagree\_step*

“OK. (.) Now we have (!...!) This part is finished. This is (!!...!!) (.) Yesyes. No, no. We have changed the Topic. We did that earlier. Now we have finished the Spy. This here still works as it did. Nothing has changed for it.”

(interleaved with (b)). After a successful test of the previous changes, driver Z20 checks the progress in the requirements document and explains it, pointing (Z19 has joined looking). Z19 interrupts him (as shown in (b)), but Z20 quickly disagrees and proceeds explaining before Z19 even starts his second sentence. Both of them shortly speak in parallel and eventually Z20 disagrees with the proposed step as well.

(b) ZB7: Z19.*challenge\_state* + Z19.*propose\_step*

“No, no, wait. Not quite. We still need to transfer this part.”

(interleaved with (a)). Z19 interrupts the driver, pointing to an item in the document. The overall logical dialog sequence of this complicated example is Z20.*explain\_state*, Z19.*challenge\_state*, Z19.*propose\_step*, Z20.*disagree\_step*, Z20.*explain\_state* (continued).

(c) **BA1:** B1.*explain\_state*

“Yes, now we are (guess) finished, huh?”

After a set of important related changes (taking 30 minutes; Example 6.5 is an outtake) and their discussion, the driver states completion. The set of changes had not been formulated as a strategy beforehand, they appeared to follow one from the other in an ad-hoc manner.

(d) **CA2:** C5.*explain\_state*

“As for checking in and so on, I think we are at a point where we can take a break.”

see Example 9.2 (6)

(e) **CA2:** C5.*agree\_state*

“Of course, that’s true.”

answer to (d), see Example 9.2 (7).

## 10.2 *state* concepts and their properties

As shown in Figure 3.1, we have so far observed *explain\_state*, *agree\_state*, and *challenge\_state*.

### 10.2.1 Short *agree* utterances

Like other types of agreement utterances, *agree\_state* statements tend to be short (“Yes.”).

### 10.2.2 Lack of reference to a *strategy*

As mentioned above, a *state* utterance does not necessarily relate to a previous *strategy* utterance, let alone explicitly relate to it. Rather, it can relate to any coherent sequence of work steps that as a whole would be considered non-atomic, whether explicitly annotated as a *strategy* or not. So far, no single *explain\_state* utterance we have seen ever referred to a *strategy* explicitly.

### 10.2.3 Partial disagreement

A *challenge\_state* will often challenge the “is completed” assessment of one subtask (of several) only and may then come along with a *propose\_step* (or perhaps *propose\_todo*). Both are relevant, so a double annotation of *challenge* and *propose* should be used in such cases.

## 10.3 Discrimination from similar concepts

### 10.3.1 *explain\_state* vs. *explain\_completion*

The discrimination between *explain\_state* and *explain\_completion* rests on the discrimination between *strategy* (or unspoken strategies) and *step* (or unspoken steps).

### 10.3.2 *explain\_state* vs. *explain\_finding*

This discrimination will be explained in Chapter 12.



## Universal concepts: What is “knowledge”?

We have already used the term knowledge many times, but so far have been vague about what it actually means in the base layer. We will explain this more concretely now.

### 11.1 On knowledge

Classic **epistemology**<sup>1</sup> (going back to **Plato**<sup>2</sup>), considers whether knowledge could be viewed to be the same as “**justified true belief**<sup>3</sup>”. This would mean knowledge to be related to a person and the statement “*I know X*” considered to be correct if the following three conditions all hold<sup>4</sup>:

- X is true
- I am convinced that X is true (belief)
- I have good reasons to believe that X is true (justified)

For instance if I am about to toss a coin and state “*I know that this coin will show head*” I may believe what I say and it may later turn out to be true, but the belief is not justified and so it was not knowledge.

---

<sup>1</sup><http://en.wikipedia.org/w/index.php?title=Epistemology&oldid=565995152>

<sup>2</sup><http://en.wikipedia.org/w/index.php?title=Plato&oldid=566836861>

<sup>3</sup>[http://en.wikipedia.org/w/index.php?title=Justified\\_true\\_belief&oldid=559691602](http://en.wikipedia.org/w/index.php?title=Justified_true_belief&oldid=559691602)


<sup>4</sup>The definition has many known limitations, but they have little relevance for our discussion here. For our purposes, the *justified true belief* notion is a useful starting point.


For our purposes, however, this definition of knowledge does not work well, because each of the above conditions is problematic:

- The exact belief is usually not stated (only a rough approximation).
- The researcher can often not decide if a belief is true or not.
- The justification is often left unsaid and none may exist.

## 11.2 The base concepts’ notion of knowledge

As a way out we draw the following consequences for the base concepts: First, base-concept-“knowledge” requires belief, but it may be incorrect or unjustified or both. We still call it knowledge, because most of the time more information is simply not available to the researcher. Even assumptions and hypotheses are beliefs in this sense: we think of them as having a prepended clause “I am not sure but I think that...”.


 Second, although all of the utterances addressed by the P&P concepts are based on something the speaker believes, this something is not considered knowledge by the base concept set: We have decided to make *separate* concepts of the P&P phenomena because we believe they are particularly important for the course of a pair programming session and we have decided not to additionally annotate the knowledge embedded in such utterances because we want to keep the annotation process lean and simple; see Section 21.4.

 Third, any pair programmer has many many beliefs that are not covered by P&P concepts and will also not be considered knowledge in the sense of the base concept set: For the reasons discussed in Section 2.3, only verbalized beliefs count as knowledge.


This notion of knowledge will in fact be split up into several concept classes for representing utterances stating different types of belief that can be expected to have specific interesting roles in the pair programming process:

- *finding* about recent insights (Chapter 12),
- *hypothesis* about partial beliefs (Chapter 13),
- *standard of knowledge* about what or how much the speaker knows or does not know (Chapter 14),
- *gap in knowledge* about something relevant that both pair members do not know (Chapter 15), and


- *knowledge* about everything else not covered so far (Chapter 16). Think of *knowledge* as the final `else` clause in a long if-else-if chain. This definition implies we have now as many as four different notions of knowledge that appear in this book
  1. The vague, common-sense notion of knowledge.
  2. The more specific term knowledge as explained in the present section.
  3. The still more specific term knowledge as the thing underlying an utterance belonging to the *knowledge* concept class (in contrast to *finding*, etc.).
  4. The concept class *knowledge* for the respective utterances.

When you find the word *knowledge*, it will refer to meaning 4; when you find the word *knowledge*, the context will make it clear which of the other meanings is the intended one, typically either 2 or 3. 

The concepts from these concept classes are called *universal concepts* because they can be applied in discussion of product and process, not just one of these. Remember that the HHI base concepts only conceptualize verbalizations, so they do not speak of knowledge that is not talked about. The rather peculiar *activity* concept class (Chapter 17) is also universal.

To keep the number of concept classes low, the potentially very important discrimination between fresh knowledge (*finding*) and previously existing knowledge (*knowledge*) is currently not made for the *hypothesis*, *standard of knowledge*, and *gap in knowledge* concepts. Future research may need to change this, in particular with respect to *hypothesis*. 

### 11.3 Priority rules for assigning knowledge concepts

Whenever an initiative (as opposed to reactive) utterance is such that we could annotate it as *explain\_knowledge*, we will do so only as a last resort. The researcher will ideally be familiar enough with the various knowledge-related concepts and the corresponding phenomena to recognize them directly. If that does not work out in a particular case, one can walk down the following priority chain and use the the first concept class that applies: 

1. *propose\_design/requirement/step/strategy/todo, explain\_completion/state, remember\_requirement*. The speaker believes the embedded knowledge to be true or potentially true (that is, good enough to be worth a discussion).
2. *propose\_hypothesis*. The speaker believes the knowledge to be likely-but-not-certainly true.


3. *explain\_gap in knowledge*. The speaker believes the knowledge to be true. The same holds for all concepts further down the chain.
4. *explain\_standard of knowledge*.
5. *explain\_finding*.
6. *explain\_knowledge*.

This priority list is simplified insofar as (a) *ask* concepts are not included and (b) annotating more than one concept is sometimes appropriate (double annotation).



## Universal concepts: *finding*

### 12.1 Topic and typology of *finding* concepts

As explained in Chapters 4 to 10, an insight that is verbalized only as a proposal for the structure of product or process or as an assessment of work state (or their discussion) will not be annotated as *finding* at all. This is because the base layer considers the proposal as such as more interesting than its origin from an insight (a point of view your subsequent research may wish to change). 

Of the remaining insights, as discussed in Section 11.3, any insight that falls into one of the concept classes *hypothesis*, *gap in knowledge*, or *standard of knowledge* will also not normally be annotated as *finding*. Utterances that rely on an insight for justifying a proposal are one possible type of *finding*. Overall, the *finding* class sits just before the final catch-all class *knowledge* as the next-to-last, quite general concept class.

But what is a finding anyway? And how can we reliably recognize one, internal as it is, if only observations of actual behavior must be taken into account?

A *finding* is any result of a thought process that was found at least to a relevant part as a recent insight or consequence of an insight. “Recent” means recent during the current pair programming session and will typically lie within the past minute. “Insight” is opposed to remembering.

- For an insight be annotated as a *finding*, it has to be verbalized as if it was an undoubted truth.
- Replies of the type “Ah, now I understand” that merely signal the speaker has understood the partner’s explanation of something do not count as *finding* (rather as *explain\_standard of knowledge*).

- An insight that is verbalized once when it is fresh and again during later episodes of the session will be considered a *finding* the first time and *knowledge* later on.
- Spontaneous verbalizations of prior knowledge (*explain\_knowledge* utterances that neither have a preceding *ask* nor a proposal context) presumably occur when the speaker had the insight that such explanation is likely useful. Such insights are not annotated as *finding*, only the *explain\_knowledge* is annotated; see Chapter 16.

There is no complete, algorithmic method for recognizing verbalizations of findings. Rather, we present several finding verbalization types (which we will imprecisely call *finding* types); each defines a set of characteristics from which an utterance should be classified as *finding*:


1. P: perceived event
2. D: discovered issue (subtypes DU, DC, DO)
3. T: thought (subtypes TB, TU, TC)

### 12.1.1 *finding* type P: perceived event

An utterance has *finding* type P if it verbalizes the recent perception (typically seeing or hearing) of events in the pair's work environment, excluding utterances of the partner. A P *finding* utterance may direct the partner's attention towards something, may include an interpretation or evaluation of the event (e.g. as welcome or unwelcome), or may potentially have still other roles.

### 12.1.2 *finding* type D: discovered issue

A *finding* has type D if it talks about passive information being observed. This is typically the verbalization of potential defects, problems, irregularities, or other relevant issues of the observed information or recognizable via the observed information. A general adequacy assessment or evaluation of a new or existing part of an artifact also counts as a D *finding*.

 Type-D *findings* usually require prior knowledge, but as usual the base layer suggests avoiding double annotations here unless your research question requires otherwise.

*finding* type D has three subtypes:

***finding* type DU: uncatalyzed discovered issue.** The information observed is in an artifact, the finding pertains to development artifacts, and the finding was not triggered or advanced by something the partner said or did or some other event in the environment.

**finding type DC: catalyzed discovered issue.** Ditto, but the finding *was* triggered or advanced by something the partner said or did or some other event in the environment.

**finding type DO: observation.** The information is not in an artifact or the finding pertains to something outside of artifacts (such as the configuration of development tools).

Example 12.1: Episode from Session CA2 (12:42:37–12:42:57) with several D-type findings. Driver C2 has opened class `FeatureAttributeConfigurationProxy` (which inherits from `AbstractFeatureAttributeConfiguration`) and the pair reviews it. The findings pertain to a potential defect in the code.

(1) C5.explain\_finding

“The get (..) (missing)”

A DU finding. After the pair had looked at the code of the class (which all fit on the screen), C5 remarks the absence of a certain getter method.

(2) C5.amend\_finding

“It gets that from the Abstract.”

A DU finding. After two seconds of silence, C5 extends or elaborates his finding. He points to the class’ inheritance clause and apparently wants to make the point that the method is inherited. He sounds like it should not be.

(3) C2.amend\_finding

“It mustn’t take the Abstract.”

A DC finding. Interrupting him, the partner completes that thought: It should not be inherited.

(4) C5.agree\_finding

“Yup.”

Immediate agreement.

(5) C5.propose\_design

“You must overwrite it.”

C5 suggests to implement their own local `get` method.

(6) C2.mumble\_sth + C2.propose\_step

“It should never (!...!). Yes. What stuff is there in the Abstract? Let’s look.”

C2 proposes to first review the full set of methods inherited from the superclass.

#### (7) C5.challenge\_design

“In a sense, it shouldn’t inherit from Abstract really. Or we must not have this in Abstract. One or the other.”

C5 disagrees with overwriting the method and proposes to clean up the whole inheritance construction instead.

### 12.1.3 *finding* type T: thought

A *finding* of type T is an utterance that verbalizes the results of thinking or sudden (more precisely: apparently sudden) ideas and insights, possibly without a recent triggering event or recently observed information. *finding* type T has three subtypes:

***finding* type TB: betterment.** Utterances that correct (or find as incorrect) a speaker’s own previous *knowledge* or *finding* utterance without external events that appear to have caused that insight.

***finding* type TU: uncatalyzed idea.** Utterances that verbalize spontaneous ideas or insights that are not betterments and that occur without external events that appear to have helped create that idea or insight.

***finding* type TC: catalyzed idea.** Utterances that verbalize spontaneous ideas or insights that are not betterments and that were preceded by an external event that appears to have catalyzed that idea or insight: The insight must be such that it required taking into account additional information beyond that coming from the event and leading thought in a different direction than that perhaps suggested by the event.

### 12.1.4 Priority rules for checking *finding* types

The *finding* types are part of the base layer (as primary characterizing attributes of the *finding* concepts), but are not stand-alone concepts within the base concept set. The purpose of the *finding* types, much like the proposal modes of Section 4.2.1 or the strategy types of Section 9.1, is to help with the decision whether to apply *finding* at all. For the *finding* types, this even takes the form of an if-else-if chain, much like for the knowledge-related concepts overall (Section 11.3) as follows:

- Assume you have worked through the Section 11.3 priority rules down

to (including) *gap in information* without finding an appropriate concept class. So the utterance ought to be either of class *finding* or *knowledge*.


- Does it verbalize (and perhaps evaluate) a concurrent event? Then annotate a type-P *finding*.
- Does it verbalize an insight or evaluation relating to information currently being viewed? Then annotate a type-D *finding*.
- Does it appear to verbalize a spontaneous insight or cogitation result? Then annotate a type-T *finding*.
- Otherwise either postulate a new *finding* type or annotate a *knowledge* concept.


Note that the *finding* types are not orthogonal; they overlap. This is because they only serve to decide what is a *finding* utterance, not to canonically classify any *finding* utterance. Feel free to add your own (and even modify the existing) *finding* types where appropriate during your research.


Despite this help, reliably recognizing *finding* utterances tends to be difficult, because by their very nature findings have strong unverballed components.

### 12.1.5 *finding* type indicators and examples

For examples of *finding* phenomena see Examples 3.1, 6.4, and 12.1 above, the compressed examples just below and Example 12.2 further down. The compressed examples serve to explain common symptoms and indicators from which to recognize the finding verbalization type of *explain\_finding* utterances.

*finding* type P applies to verbalizations of recent computer outputs. Example from Session BA1: B1 has started an Eclipse search. While it is still running, B2 comments “Two matches.” 

*finding* type P applies to evaluations of test run results. Example from Session CA2: After some code changes, the pair runs the application and manually tests it. None of them says anything. After several steps, the driver concludes “There!” 

*finding* type DU applies to summaries regarding the quality of a part of an artifact the speaker has visibly just reviewed (see *verify\_sth* in Section 19.4). Example from Session CA2: The driver announces he will inspect a certain method and runs down its lines with the mouse pointer while doing so. Near the end he states “Yes, fits.” 

*finding* type DU applies to utterances about some unusual property of a part of an artifact the speaker has visibly just reviewed. Example from Session CA2: While reviewing a method and running down its lines with the

☞ mouse pointer, the speaker reaches a variable declaration and says *“This is an IFeatureProxiesTableModel, not a normal one”*.

*finding* type DU applies to utterances that effectively state *“I found what I was looking for”* at the end of the search for an item with some property. Example from Session CA2: The driver wants to navigate to a method to be changed, but neither remembers its name nor the classname, so he looks through the code of several plausible classes before he utters *“Ah, there. (..) Exactly. (..) That is it, I think”*.

☞ *finding* type DC applies when the speaker explicitly points out a defect after the partner made a remark that apparently made the speaker recognize the problem. Example from Session BA1: The driver attempts to decipher the meaning of an existing if condition and does not succeed. The observer provides partial explanations that make the speaker recognize that his only problem was that the code comment explaining the condition (that he himself happens to have written in an earlier session) was wrong. He says *“Then I wrote the wrong comment, only.”*.

☞ *finding* type DO verbalizes aspects of information freshly shown on the screen that relates to the work environment itself, not the program artifacts nor the events from test runs, etc. Example from Session BA1: Driver B1 opens the list of installed plugins in his IDE to check the status of the PHPEclipse plugin. No entry of this name exists and he asks rhetorically *“What happened to my PHPEclipse that originally I had installed here?”*

☞ *finding* type TB applies to self-criticism regarding changes the speaker had previously made on her own initiative. Example from Session BA1: The driver asks why two particular statements are not grouped together and then groups them without waiting for an answer. He ponders his change for five seconds (the partner still passive) before he says *“Well, it’s not really nice like this”*.

☞ *finding* type TC applies to the following example from Session BA1: The driver adds a structured comment to a PHP function. As he writes *“@return int”*, the observer makes a “no” sound and the driver immediately states *“There we have our problem”*. The function returns int at one point and boolean at another. The observer was aware of the boolean part; his utterance is *disagree\_activity* (see Chapter 17) and a P-type *explain\_finding*. This made the driver (who was previously more aware of the int part, but not exclusively so) recognize the problem and state it as a TC-type *explain\_finding*.

☞ *finding* type TC applies to a conclusion drawn from a previous knowledge utterance of the partner. Example from Session CA2: Observer C2 suggests to use the Alt-Shift-C key combination, but C5 explains that that would do something different (having to do with constants). C2 checks this against his own keystrokes knowledge and concludes *“Then you have changed that”*; see Examples 16.2 and 16.4 for details.

Any *finding* type may apply if the speaker emphasizes an insight explicitly by linguistic means such as “Now I understand...”, “Oh, wait...”, “Aaaahhh!”, etc.

Examples for this from Session **BA1**: The driver comments on the name of a function he has been editing “*I just recognize that’s total bullshit, that name*”. Elsewhere he says “*Aahh, we don’t want to reduce traffic really, of course, we want to save write accesses to the remote side. That’s the whole trick: That they don’t update something that is up-to-date anyway*”. At still another time the observer says “*Oh, Account!*”. From their context, we can recognize these as DU, TC, and P *finding* utterances, respectively.

Example 12.2: Episode from Session **BA1** with many *findings*. A number of code changes have been performed so that the PHP script should only return those database entries modified in a given time interval. For testing, the pair had enforced a particular time-difference by hard-coding, then removed that code again just before this episode. All utterances but one come from B1.

(1) B1.*propose\_step*

*“And now we’d said we could run a little simulation now.”*

B1 suggests to do a realistic test call from the browser and performs it right away.

(2) B1.*explain\_finding*

*“Now we return files (.). That means the last change (.) was after the query.”*

*finding* type P: He verbalizes the result that appears in the browser (a list of strings each containing one hexadecimal result value) and interprets it as having a particular property.

(3) B1.*propose\_step*

*“Stop, wait. After the last query?”*

A rhetorical question after several seconds of silence. Its meaning is “*Let us consider whether my interpretation was indeed correct.*”

(4) B1.*ask\_knowledge*

*“What do we hand over really as timestamp[last\_request]?”*

B1 has switched back from the browser into the IDE where the *if*-condition is still marked that determines the return value. It is unclear whether his question asks for the current value of the variable or for its general meaning.

**(5) B2.explain\_knowledge**

*“Well, that’s where the (~remote side) hands over to us (.) its Timestamp (..) when it last (~modified) the (~FriendslistId).”*

B2 explains the semantics of the variable without hesitation.

**(6) B1.agree\_knowledge + B1.ask\_knowledge**

*“Right, but what is our script doing now?”*

B1 agrees immediately; he has apparently known this before. He asks about the behavior in the current specific case.

**(7) B1.explain\_finding**

*“I gave it some arbitrary Timestamp (;;; I’m surprised that it worked with -1000, +1000.”*

After four seconds of silence, B1 formulates the insight that the `Timestamp` he submitted to the test run was not chosen in any particular manner and so the script should *not* have returned the expected results like it did. The screen still shows the IDE initially and B1 switches back to the browser while speaking and marks the relevant part of the URL. The insight may have arisen earlier: before step (3). The numbers -1000, +1000 refer to previously hard-coded values that were actually -100, +100 and had been removed before.

Even with the many explanations we provide, the content of the example is difficult to understand, which illustrates how much context understanding may be needed for annotating even the base concepts properly.

## 12.2 *finding* concepts and their properties

As shown in Figure 3.2, we have observed *explain\_finding*, *agree\_finding*, *disagree\_finding*, *amend\_finding*, and *challenge\_finding*.

Many properties of *finding* utterances have already been explained in the previous section. We will not repeat those here, only discuss more specialized ones.

### 12.2.1 Aggregation of utterances

If an utterance contains what could be viewed as several (but related) findings, we still annotate only one *explain\_finding* to avoid unnecessary decision difficulty. You might want to drop this rule if findings are a focus of your investigation. If the findings appear unrelated, the utterance should be considered two utterances and annotated separately.



### 12.2.2 Repeated statements

If the same finding is apparently verbalized more than once in similar or paraphrased form, we consider the following cases depending on who repeats and when:

- Same speaker shortly afterwards: Annotate the second utterance with *explain\_finding* again.
- Different speaker shortly afterwards: Decide which of the following is the most likely meaning of the utterance:
  - “OK, I understand your idea”; this would be annotated as *explain\_standard of knowledge*.
  - “OK, I understand your idea and think it is correct”; this would be annotated as *agree\_finding*.
  - The speaker had the same idea as the partner at essentially the same time; this would be annotated as another *explain\_finding*.
- Same or different speaker a long time later: If the utterance shows symptoms of a fresh finding again, it should again be annotated as *explain\_finding*. Otherwise, the previous finding should now be considered prior knowledge and be annotated as *explain\_knowledge*.

### 12.2.3 Thinking aloud

Some developers sometimes verbalize substantial parts of their thinking process in real time. For instance we found this long utterance in Session **BA1**:

*“First change zero (!...!) (..) The problem is if it returned the current Timestamp (!...!) (..) (Hm.) (.) We have a problem anyway that we keep the clocks in sync. (.) And if we take the Timestamp now it could happen that by a few seconds, milliseconds (!...!) (.....) No, seconds. That it lags behind by one, two seconds and (.) that we send something unnecessarily.”*



The utterance is far from contiguous (it has long pauses), but it is coherent and uninterrupted and is hence considered a single utterance rather than several. Unless some of its parts have to be annotated as *explain\_standard of knowledge* (see the discussion in Chapter 14) or *explain\_knowledge* (see the discussion in Chapter 14), we will annotate such cases simply as *explain\_finding*.

### 12.2.4 Revoking and replacing *findings*

Just like we saw speakers amend or revoke their own *design* proposal (see Section 4.2.6), we saw speakers revoke their own *finding* and replace it by another. Such utterances should be annotated as *challenge\_finding*. Do not be confused if the new version states prior knowledge: The fresh insight is that this is (subjectively) correct. See also the discussion in Section 21.6.5.

### 12.2.5 “Additional” findings

It is often difficult to decide whether an utterance is “merely” an addition or detailing of a previous finding (with or without elements of a new insight, but which should be annotated as *amend\_finding*) or an entirely new one (and should be annotated as *explain\_finding*). As in many similar places in the base concept set, we err on the side of reducing complexity and will always annotate such cases as *amend\_finding*, no matter which speaker. Subtle cases arise in the context of *knowledge* and will be discussed in Chapter 16.3.

Example 12.3: Episode from Session BA1 (14:44:21–14:44:52) in which a speaker amends his own previous *finding*. The episode also shows how to detect proposals (*design* in this case) in *finding* contexts.

#### (1) B1.*explain\_finding*

“I just recognize that’s total bullshit, that name.”

*finding* type DU: B1 has added a comment on function `registerFriendsLastChange` and now evaluates the function name.

#### (2) B2.*ask\_knowledge*

“Why?”

The observer promptly wants an explanation.

#### (3) B1.*propose\_design*

“Either `registerFriendsChange (!...!)`”

Almost interrupting the partner (and not answering his question), B1 proposes how the function should be named. This is an elaboration of the finding, but also a *propose\_design* which takes annotation precedence. He starts to offer two versions but stops after the first (perhaps due to another insight, but this is not relevant at this point).

#### (4) B1.*amend\_finding*

“Ah, (.) you know, (`~FriendsLastChange`) is total nonsense somehow so ‘cause we register, somehow, that this event occurred.”

After three seconds of silence, B1 elaborates on his insight. “Ah” signals another finding, but its close relation to the initial one makes us annotate it as *amend*, not *explain*.

(5) B1.challenge\_design

“What do you think of (...) UpdateFriendsLastChangeTime?”

Without a pause, B1 makes another naming suggestion different from his (incomplete) previous one. He starts performing the renaming even as he speaks.

(6) B2.agree\_design

“Wonderful.”

B2 accepts the second proposal.

(7) B1.amend\_finding

“‘cause that’s the nice thing about it is (...) (~that it’s different here everywhere). Here we have it and here we update it. And that is clean I guess.”

Although B2 has not asked, B1 immediately explains his reasoning. Both times the “here” is supported by pointing the mouse. This sounds more like an elaboration than like a new insight, so we annotate it as *amend\_finding*.

(8) B2.agree\_finding

“Good.”

This could refer to many things so we consider it as referring to the temporally closest of them.

Example 12.4: Episode from Session BA1 (14:23:59–14:24:38) in which a speaker amends the partner’s *finding*. It also serves as an example of how even difficult acoustic conditions do not always prevent proper annotation.

(1) B1.propose\_step

“Er, I’d say, we go copy everything here bad-ass.”

The pair wants to implement the function `getFriendsLastChange` which currently has an empty body. B1 suggests to start from a copy of the whole body of function `getFriendsIDs`. He scrolls to the top of that body while he speaks.

(2) B2.propose\_design + B2.explain\_finding

*"The SQL we don't need anymore. Because we only have an a Memcache query."*

After one second, B2 remarks that the first statements of the body will not be needed and explains why. Meanwhile, B1 starts copying the body.

(3) B1.*agree\_design*/B1.*agree\_finding*

*"Exactly."*

B1 immediately agrees in full; apparently, this was not new information for him.

(4) B1.*propose\_step*

*"I'll delete it in a second."*

Three seconds later, the copying is still going on, B1 explains how he intends to handle the issue.

(5) B1.*explain\_finding*

*"What are these TABs here?"*

B1 reports that he has stumbled over some incorrect source code formatting. Could as well have been annotated as *ask\_knowledge*. The utterance has no consequences in the episode.

(6) B1.*explain\_finding*

*"If not dollar id?!"*

After finishing the copying ten seconds later, B1 reads a line of code from the screen: `if (!$id)`. The pronunciation sounds like he has found an irregularity, not like an actual question.

(7) B2.*mumble\_sth*

*"Return array, ne, (~), (null)"*

This appears to talk about the loop body: `return array();`. The middle part is incomprehensible because B1 talks at the same time.

(8) B1.*amend\_finding*

*"(crap)"*

B1's adequacy assessment for the `if` clause.

(9) B2.*amend\_finding*

*"(~) Dollar id so-and-so (~)"*

Much of the utterance is incomprehensible, but the understandable part clearly references the previous *explain\_finding* and the partner's reaction shows he has understood. We therefore annotate this utterance as *amend\_finding*.

(10) B1.*agree\_finding*

"Yes, exactly."


B1 has apparently understood without problem and immediately agrees.

### 12.2.6 Justifications of proposals

As discussed in previous chapters, when a rationale or justification is provided along with a proposal, this will usually be annotated as *explain\_knowledge*. Sometimes, however, this knowledge will be fresh and then *explain\_finding* is the appropriate concept. The rules for discriminating these two in this particular context will be discussed in Chapter 16.


### 12.2.7 Justifications of *findings*

If a *finding* and an accompanying explanation or justification are stated in a single utterance, without a longer pause and without an intermediate utterance from the partner, they are annotated as one single *explain\_finding* (or *amend\_finding* or *challenge\_finding*), because it is impossible to decide whether they represent a single insight or not.

We usually ignore whether or not prior knowledge is part of the explanation; specialized studies may want to change this decision. 

### 12.2.8 *disagree\_finding, challenge\_finding*

Just like for other objects, the partner may reject a *finding* with either *disagree* or *challenge*.

*disagree\_finding*: The speaker does not provide information beyond the rejection itself. Actual cases of this infrequent behavior may take curious forms. For instance in Session CA2, C5 had said "*It needs to do a rebuild – it is rebuilding right now*" (*explain\_finding*), which C2 rejects without explanation by saying "*That has nothing to do with the taskbar not working*" (*disagree\_finding*); see Example 19.4 for more context. This reply is just as illogical as it sounds, the underlying problem being that C2's previous complaint about the taskbar had been completely missed by C5. The rules of annotating base concepts know nothing of such mishaps, so we simply annotate *disagree\_finding*. 

*challenge\_finding*: The speaker suggests either a different finding or a reason why the partner's finding is likely incorrect. The former can either lead into

agreement or into controversy. The latter can either lead into controversy or into a state where the pair is left with no valid finding at all, as in the following example (we will come back to this topic during the discussion of *challenge\_knowledge* in Section 16.2.9):

Example 12.5: Episode from Session ZB7 (16:15:39–16:15:48) in which the speaker rejects his partner’s *finding*. The annotation does not reflect whether the rejection involves prior knowledge.

(1) Z19.explain\_finding

“No, with the lib wasn’t right, there had to be ‘dot dot’ there.”

The pair works through the file `build.properties` and the build error messages on the Eclipse console. The observer suggests that the line `lib.dir=${basedir}/lib` should in fact be `lib.dir=${basedir}/../lib`

(2) Z20.challenge\_finding

“That cannot be. The lib is lying here, too.”

The driver immediately replies why this cannot be true, but nevertheless makes the suggested modification. This behavior points out a boundary of the base layer: it does not express such inconsistency between speaking and action.

(3) Z19.agree\_finding

“Yes, right. Then it wasn’t the problem.”

The observer withdraws his original *finding*.

The rejection is not always explicit, it can also be implicit in a *propose* in which case we perform a double annotation as in the following example:

Example 12.6: Episode from Session CA2 (12:39:27–12:39:52) in which the speaker rejects his partner’s *finding* implicitly by proposing something that would not be needed if he agreed.

(1) C2.explain\_finding

“Good.”

C2 has started a test run to see if a certain attribute table works as intended. He accesses the menu, displays the table, then displays the attributes, then makes this utterance which confirms the test as successful.

(2) C5.propose\_step/challenge\_finding

“Do (..) please change the values (..) and go there again.”

C5 suggests that modifying the attribute values should be tested as well. This is primarily a *propose\_step*, but implicitly also refuses the partner's "Good" conclusion as premature. The *challenge\_finding* (as opposed to *disagree*) takes the view that the proposal explains what needs to be done before the finding can become justified.

(3) *C2.agree\_step/C2.agree\_finding*

"Um. (.) Yes."

C2 agrees immediately (almost interrupting C5's utterance). He modifies an attribute value, closes all attribute tables, and reopens them.

(4) *C5.explain\_finding*

"(OK.)"

The observer recognizes that the table still contains the previous value. His utterance is fairly quiet.

(5) *C2.explain\_finding*

"Okayyy, doesn't work."

A moment later the driver also states that something is wrong. We annotate this as *explain\_finding* rather than *agree\_finding* because C5's utterance was quiet and C2 stared at the screen transfixedly at the time and has presumably not heard it at all.

(6) *C5.propose\_step*

"(Let's look.)"

The observer asks (but quietly) to see the code.

(7) *C2.propose\_step*

"Theeeeeen we should look why."

Two seconds later, the driver proposes the same. Again he has apparently not heard his partner.

### 12.2.9 Reasons for agreement

As in the other concept classes, *agree\_finding* utterances tend to be short. In some cases it is helpful to consider three possible situations in which the speaker expresses his or her agreement:

- known: The agreement may express "I knew this already since some time" (in the sense of prior knowledge as opposed to a finding). See for instance

the first *agree\_finding* in Example 12.4. In principle, such utterances are *explain\_standard of knowledge*, but we give the existing *finding* context priority unless there is a rather explicit declamation of a standard of knowledge.



- understood: The agreement may express “*I did not know this, but I understand what you say and now I think so, too*”. For instance, in Session BA1, B2 explains “*You haven’t passed any parameters*” to which B1 replies with a happy “*Right!*”.



- me-too: The agreement may express “*I just had the same insight*”. For instance, again in Session BA1, B1 explains “*The problem is when you stop the script here (.) then (.) the request will not get a response. (.) Or though then it will send its standard header and also (~not) send anything*” to which B2 replies “*I suppose so*” (which is *not* a *hypothesis* utterance because we are in a *finding* context already).



Like the proposal modes in Section 4.2.1, the strategy types in Section 9.1, and the *finding* types in Section 12.1, these agreement types are primary characterizing attributes but are not themselves stand-alone members of the base concept set and you need not be able to annotate them consistently; they serve only to make it easier to understand whether a particular utterance is an *agree\_finding* or not. We have not found a similar discrimination necessary for refusals so far.

### 12.2.10 Doubt

Agreements to *findings* may involve visible or audible doubt: The speaker is only half convinced. We will discuss this issue in Section 21.6.4.

### 12.2.11 *ask\_finding*?



There is no base concept *ask\_finding*, because the asker just needs information and will usually not care whether the underlying knowledge is new or old. (Exceptions are possible, for instance mentors might ask questions that aim at triggering findings.) Therefore, expect to see episodes in which *ask\_knowledge* is followed by *explain\_finding* or a similar pattern as in Example 12.7.

Example 12.7: Episode from Session BA1 (15:17:14–15:17:42) which is the closest we ever came to introducing the concept *ask\_finding*. However, strictly speaking the asker cannot know (and likely will not care) whether the requested knowledge is old or new, so we stick with the existing *ask\_knowledge* instead. The example also illustrates *amend\_finding*.

(1) B1.*explain\_finding*



*“The last Request here, (;) last and here, er, change and thus (!...!) Yes, right. Yes, now we get a friends list.”*

The driver thinks about what a certain if statement will produce in the situation currently considered. He takes notes on a slip of paper and partially articulates his thoughts. As a result, he formulates the insight that the set of friends of a certain person will be computed.

**(2) B2.challenge\_finding**

*“No, we don’t get a friends list.”*

B2 immediately contradicts B1 by postulating the opposite.

**(3) B1.challenge\_finding**

*“Yes, we do get one.”*

B1 repeats his original claim. (The reaction shows that B2.challenge\_finding is more appropriate than B2.disagree\_finding above; see Section 21.7.3.)

**(4) B2.ask\_knowledge**

*“Why? If (!...!!)”*

B2 demands a justification (and B1 interrupts the question early). As the justification of an insight is likely part of that insight or another insight, this could have been called ask\_finding but for the reasons explained above we have not introduced that concept.

**(5) B1.amend\_finding**

*“100 seconds after their last query we have (.) 100 seconds after their last query we have changed something (..). So we have changed something after they last asked. So they now get a fresh answer.”*

B1 explains his insight by providing details.

## 12.3 Discrimination from similar concepts

Discriminating *finding* concepts from product-oriented and process-oriented concepts is guided by the principle that the latter classes take priority as explained in Section 11.3 and the principle that knowledge transmitted along with process/product concept utterances is not annotated separately (only justifications are) as explained at the end of Section 3.6.

The remaining issues are discussed in the subsections following below.

### 12.3.1 *finding* vs. other universal concepts

Most ideas for discriminating *finding* from *knowledge* were already explained in Section 12.1 above and the rest will follow in Chapter 16 about the *knowledge* class. Discriminating *finding* from the other classes of universal concepts will be discussed only in the respective chapters.

### 12.3.2 *explain\_finding* vs. *propose\_design*

See Section 3.7, Section 4.2.3, and Section 12.3.3.

### 12.3.3 *explain\_finding* vs. \*\_*step*

For *propose\_step*, see Section 6.2.1. If a speaker explicitly marks an insight as part of a *step* utterance, this is annotated as an additional *explain\_finding*. We have seen such behavior when the speaker, after a proposal had been formulated, has perceived something that suggests the proposal was unsuitable (as in Example 6.1). The same rule should be applied to the other product-oriented and process-oriented concepts, but we have not seen such phenomena so far.

### 12.3.4 *explain\_finding* vs. *explain\_completion* or *explain\_state*

*explain\_completion* and *explain\_state* are both special cases of *explain\_finding* and therefore take annotation precedence.

## Universal concepts: *hypothesis*

### 13.1 Topic of *hypothesis* concepts

A *hypothesis* in the sense of the base concepts is an utterance relating to a conjecture, assumption, or hypothesis that the speaker observably doubts to some degree. It may be that the speaker expects it (or sometimes its opposite) to be true more likely than not or that no particular conviction is present. It can be new (a doubted finding) or old (uncertain knowledge). Actual verification of the hypothesis needs not be practical for the pair.

We have seen the following three types of *hypothesis* utterances, but more may exist. For examples see Section 13.1.4 which lists the types of issues that *hypothesis* utterances tend to talk about.

#### 13.1.1 Uncertain knowledge

A *hypothesis* of type 'doubted' pertains to something that could (and often should) be part of the speaker's existing knowledge but that is available only partly, making the speaker doubt it. For instance, relevant detail may be lacking, the speaker may be only half-convinced of the correctness of his or her memory, or the knowledge is altogether vague. Such a *hypothesis* is not specific to the context of the current session.

#### 13.1.2 Hard-to-verify assumptions

A *hypothesis* of type 'hard-to-verify' pertains to something that either takes a lot of effort to check (beyond what is practical in the current session) or that the speaker does not know how to check or that can only be verified by waiting a long time (beyond the length of the current session) to see how the world develops.

### 13.1.3 Readily verifiable conjectures

A *hypothesis* of type ‘can-check’ pertains to something that the pair is able to check within the current session. It is not important if they actually do.

### 13.1.4 Issue types addressed by hypotheses

We have seen the following types of topics (issue types) occur in *hypothesis* utterances, but more may exist.


**Program execution properties.** Hypotheses about what happened in a recent program execution or what may or will happen in a future one. For example, a few seconds after a `NullPointerException` has terminated a run in Session **ZB7**, Z20 comments: *“You know what happened? (...) I guess (no) (.) I don’t know but I guess (...) that the receive didn’t receive anything after 10 seconds.”* (type can-check). And in Session **BA1**, after the pair has modified a call argument, the observer speculates about the effect *“Is bigger. (So we) should get an exit, not a friends list, right?”* (type can-check). Such utterances can have any hypothesis type but will often tend towards can-check.

**Environment properties.** Hypotheses regarding properties of the various development tools used, the networking environment, etc. For example, while the pair investigates configuration issues in Session **ZB7**, Z19 comments *“Does, er, (!...!) I think XDoclet butchers that (...) perhaps?”* (type can-check). And in Session **BA1**, after a call to the `wget` utility failed, the observer has an idea why and asks *“Firewall?”* (type can-check). Such utterances can have any hypothesis type but will often tend towards can-check.

**Program artifact properties.** Hypotheses regarding properties of the program that are formulated in terms of the program text and related artifacts. For example, after a test run yielded the wrong result in Session **BA1**, B2 utters *“Safe::id\_to\_code. Probably some old bracketing error”* (type can-check). And in Session **CA2**, before moving a class to a different package, C2 proposes a justification *“This is probably not used here anymore”* (type is unknown). Such utterances can have any hypothesis type.

**Process course.** Hypotheses formulated about properties of past or future development activities. For example, after the n-th failed change-build-test cycle Z20 says in **ZB7** *“Perhaps we should have made a real undeploy”* (type can-check). And in Session **CA2**, C2 justifies a strategy proposal by saying *“Cause I fear that for this work task that we will exceed the time anyway, (.) because it will be fairly complicated”* (type can be considered can-check or hard-to-verify). Such utterances can have any hypothesis type.

**Conventions.** Hypotheses about general conventions (often regarding formatting or naming) that the pair is expected to follow. For example, after C5 has started filling the comment field during an SVN commit operation with

“cad-507” in Session CA2, C2 interjects “*I think the ‘cad’ is upcased or so?*” (type doubted). Such utterances will usually have hypothesis type doubted. 




## 13.2 hypothesis concepts and their properties


As shown in Figure 3.2, we have observed *propose\_hypothesis*, *agree\_hypothesis*, *disagree\_hypothesis*, *challenge\_hypothesis*, and *amend\_hypothesis*.



### 13.2.1 propose\_hypothesis

*propose\_hypothesis* represents utterances in which the speaker states how something *could* be explained or what properties something *might* possess. For this character, the verb *propose* appears more appropriate than *explain*, so this is what we use.

### 13.2.2 agree\_hypothesis, disagree\_hypothesis, challenge\_hypothesis

*agree\_hypothesis* represents any utterance by which the speaker expresses that he or she believes the proposed hypothesis is correct more likely than not. For example in Session CA2, C5 explains where he thinks the pair can obtain a required task ID by saying “*Well I think <Developer> has it in his Opera*” to which C2 agrees with a quiet “(OK)”. If the utterance also explains why the speaker agrees, this is expressed by annotating *explain\_knowledge* or *explain\_finding* in addition. For example, in the episode from Session ZB7 we know from above, Z20 hypothesizes “*You know what happened? (...) I guess (no) (...) I don’t know but I guess (...) that the receive didn’t receive anything after 10 seconds*” and Z19 replies “*Yes, it took a while, right*”, which we annotate as *agree\_hypothesis+explain\_finding*.   
  


Note that this style of annotation loses the relationship between agreement and explanation, so you might want to extend the modeling if that aspect is important for your particular study. 

Likewise, *disagree\_hypothesis* represents any utterance by which the speaker expresses that he or she believes it is more likely than not that the proposed hypothesis is incorrect. If the utterance also explains why, this is again expressed by annotating *explain\_knowledge* or *explain\_finding* in addition. For example in Session ZB7, Z19 appears to believe that the failing of the previously run test most likely occurred because not all required configuration settings had been made by the pair; he is leafing through a configuration documentation. He vaguely states his hypothesis as “*Yes, er, to get (...) to get that running we need to do certain things and something is missing of that*”, but Z20 disagrees by saying “*But we DID do all that*” (*disagree\_hypothesis/explain\_finding*).   


Had the speaker disagreed and offered an alternative hypothesis, this should have been annotated as *challenge\_hypothesis*.

### 13.2.3 Conditional agreement

Sometimes the speaker says something of the type “If we find condition X to hold, then I agree with your hypothesis”.

For instance in the example of Section 13.1.4, Z20 says “*Perhaps we should have made a real undeploy*” and Z19 replies “*If it still doesn’t work now, we can do it*”. We consider such conditions that do not modify the hypothesis as such to be part of the agreement and annotate the above utterance as *agree\_hypothesis/propose\_step*.

### 13.2.4 Revoking or replacing one’s own hypothesis

Like for *explain\_finding* (see Section 12.2.4), we found cases of *explain\_hypothesis* where the speaker takes it back a short time later. Sometimes, the speaker simply appears to lose confidence in the former idea; for example in Session CA2, C2 proposes “*Because it could be that it’s just this one AttributeConfiguration that we don’t have added yet*” and then adds “*But (.) I don’t think so*” (*disagree\_hypothesis*).

In other cases, a “better” idea appears (from discussion or simply from further thinking) and the hypothesis is replaced by a supposedly superior one; for example in Session CA2, C5 first proposes “*That should, we must, the demo down here <\*points to a class\*> (.) it ought to be in there, the EditColumnAttribute*” and then has a 12-second discussion with his partner, during which the class is not opened, and concludes “*Stop, no, then it must be up here <\*points to a different class\*>, right?*”. Such cases are annotated as *challenge\_hypothesis*.

### 13.2.5 *amend\_hypothesis*: One hypothesis or several?

If a subsequent utterance refers to a previously proposed hypothesis, and is not simply agreement or disagreement, it is not always clear whether it proposes a new hypothesis or refines the previous one. If it proposes a new hypothesis, that might be an additional one (*propose\_hypothesis*) or an alternative one (*challenge\_hypothesis*). If it refines the previous hypothesis, it should be annotated as *amend\_hypothesis*.

We use the following criterion for deciding when to encode with *amend*: If the original proposal can be paraphrased in the form “If A, then B”, then we use *amend* if the new utterance widens the condition (“If A or A2, then B”), narrows down the condition (“If A and A2, then B”), loosens the consequence (“If A, then B or B2”), or extends the consequence (“If A, then B and B2”).

Such utterances do not appear to be frequent; we have not seen any single instance of them. The only reason that *amend\_hypothesis* is part of the base concepts nevertheless is Example 13.1 which, with a little stretching of the above definition of *amend\_hypothesis*, could be considered to contain one.

Example 13.1: Episode from Session ZB7 in which driver Z20 states hypotheses and supporting justifications. One justification is another hypothesis, the other comes from existing knowledge.

(1) Z20.propose\_hypothesis

*"I think these dot dot there are wrong. (..) And it could be that I had set them on my machine."*

While discussing the reasons for the failure of a previous build process, the pair looks at a configuration entry of the form `conf.dir = $basedir/./conf`. The driver conjectures that the 'go to parent directory' part of the path is wrong (H1). He adds another conjecture H2 that this setting was correct on his machine (from which it was taken) but is inappropriate for the setup of the present machine. The second hypothesis serves to support the first and we decided to consider them one single hypothesis in our annotation.

(2) Z20.mumble\_sth

*"I, er (!...!)"*

The driver removes the './' part from three such paths in the configuration file.

(3) Z19.ask\_knowledge

*"Are these all files so, not?"*

Z19 asks if corresponding changes need to be made elsewhere.

(4) Z20.propose\_hypothesis

*"Hm, hmmm. No, that is only (.) here it should be proble(.).matic."*

Z20 does not appear to be sure; the reply is another conjecture (H3).

(5) Z20.explain\_knowledge+Z20.propose\_hypothesis

*"Cause I had the project different before. The basedir I think is the Eclipse project."*

Z20 justifies his former hypothesis. The first part is existing knowledge, the second part is another hypothesis (H4). We view H4 as an additional hypothesis here, but if we would construe the first statement above to have the shape "If H2, then H1", then we could take the present one as refining it into "If H2 and H4, then H1" and accordingly annotate it with *amend\_hypothesis*.

(6) Z19.explain\_standard of knowledge+Z19.agree\_hypothesis

“Ah, now I see. You are right.”

The second part formulates agreement – it is not fully clear to what. The first part does not sound like a very local statement, so it makes sense to assume the agreement covers more than only H4, which is another argument why *amend\_hypothesis* might be an appropriate annotation above.

### 13.2.6 Justification of hypotheses

Not only agreements and disagreements (as in Section 13.2.2) but also the hypothesis proposals themselves often come with a justification. Depending on its content, this justification should be annotated separately as either *explain\_knowledge*, *explain\_finding*, or *propose\_hypothesis*. Do not confuse parts of the hypothesis (such as conditional clauses) with justifications.

### 13.2.7 Justification by hypotheses

A hypothesis may also serve as justification for any other proposal (not just a hypothesis), as well as as justification for various types of *explain* and other utterances.

## 13.3 Discrimination from similar concepts

Most interesting is the discrimination from *finding* and *knowledge* concepts. The former is discussed below, the latter will be discussed in Chapter 16.3.

### 13.3.1 *propose\_hypothesis* vs. *explain\_finding*

A hypothesis will often be of recent creation rather than being part of existing knowledge and then represents an insight (finding). So when do we annotate *propose\_hypothesis* rather than *explain\_finding* in such cases? We have defined that the main criterion is incomplete conviction of the speaker: A finding is considered true, a hypothesis is considered uncertain. But how to discriminate those cases?

Indicators for the hypothesis-ness of an utterance are words such as think, hope, guess, expect (and others) or the use of question form. But neither do all *hypothesis* utterances exhibit those features nor do all *finding* utterances lack them – “think” in particular is highly ambiguous:




“Ahh, I think it has this (!...!) it does not accept this as a PHP project.”

In this quote from Session BA1, the insight character is obvious, but the degree of conviction is not.



The only solution is judging, from the context before and after the utterance, whether the speaker appears to be (at the time of the utterance) convinced of the truth of the assumption or not. In the above case, our analysis decided yes and we hence annotated *explain\_finding*.

Studies that pay major attention to validation-of-knowledge processes in pair programming will definitely need to create finer instruments for diagnosing and expressing what is going on in the session than the base layer provides. 



## Universal concepts: *standard of knowledge*

### 14.1 Topic of *standard of knowledge* concepts


The *standard of knowledge* concepts represent utterances in which the speaker queries the partner's or explains his or her own *level of* currently available knowledge: Whether a particular piece of knowledge is consciously present or how much knowledge about a certain topic is available (or missing). The utterance does not aim at communicating the content knowledge itself.

These utterances talk about the knowledge of a single pair member, not the pair as a whole (see *gap in knowledge*, Chapter 15, for the latter). The *standard of knowledge* concepts exist because we found they serve important roles in the pair programming process; *explain\_standard of knowledge* typically prepares or rejects or acknowledges a knowledge transfer. The following subsections explain these three different *standard of knowledge* utterance types.

#### 14.1.1 PT: Preparing knowledge transfer

The speaker states that he or she lacks certain knowledge (and perhaps explains what that knowledge would be about), implying (or stating) that this knowledge would need to be transferred to him or her. It is not important whether that transfer then actually happens or not or even whether it would be feasible. There are three subtypes:

**PTd, for decision:** The speaker explains to what extent he or she possesses knowledge sufficient for decisions to be made soon.

For example in Session CA2, C2 has just stated that he thinks a Facade should be introduced into the program to hide some details and then continues "What 

*the Facade looks like I don't know. That I don't yet, I don't (.) yet see".*

**PTe, for execution:** The speaker explains to what extent he or she possesses knowledge sufficient for performing certain activities (often a step) required soon.

☞ For example in Session **CA2**, the pair needs to determine the official task ID of their current work goal and driver C2 says “(*~Could*) you just look? I have no idea how I would (!...!). Don't know by heart”. The solution suggested explicitly in this utterance, that the better-informed partner next take the driver role, is often tacitly applied to resolve such situations.

**PTo, other:** This is the catch-all type for knowledge transfer triggers that do not immediately pertain to decisions or actions.

☞ For example in Session **BA1** during a sequence of connected simple changes, B1 states that he momentarily lost and then presumably re-established his orientation: “*translate each friend's id to ids: We've been there before. I'm really not sure right now where we are. (~Ah, here)*”. At some other point he says about a complicated if statement: “*I find it all quite confusing as well. I always make myself a sketch each time to understand it (~at all).*” (*explain\_standard of knowledge+explain\_knowledge*)

### 14.1.2 RT: Refusing knowledge transfer

☞ The speaker negatively answers a query for information (*ask\_knowledge*) or for a proposal (e.g. *ask\_step*) by stating that he lacks the necessary knowledge for fulfilling the request. For example in Session **CA2**, C5 asks what argument to pass to a method parameter: “*That needs A11, probably?*”. C2 replies “*No idea what that thing does*”.

### 14.1.3 AT: Acknowledging knowledge transfer


☞ The speaker states whether or to what degree a previous knowledge transfer (*explain\_knowledge* or *explain\_finding*) was successful or what its result was. For example in Session **CA2**, after C5's explanation of the reasons for some changes he had made, C2 states “*OK, I don't get it, but OK. I haven't understood, but OK*” to acknowledge the knowledge transfer as unsuccessful, yet terminate the episode nevertheless. Elsewhere in **CA2**, C5 explains where to find a certain class: “*The VirtualAttribute is here in pro <\*points on screen\*>*”, which C2 first repeats, sounding astonished, then acknowledges as understood: “*In <\*packageName\*>.pro is the VirtualAttribute? OK*”. Type AT includes the case that the speaker states to have possessed the respective knowledge even before the transfer.

## 14.2 standard of knowledge concepts and their properties

As shown in Figure 3.2, we have observed *ask\_standard of knowledge* and *explain\_standard of knowledge*.

### 14.2.1 *ask\_standard of knowledge*

The speaker sometimes queries the partner regarding how much he or she knows about something. If the immediate goal of such questions is more obtaining an understanding of the constraints under which the session is performed rather than obtaining knowledge for directly pursuing the session's goals, then we encode them as *ask\_standard of knowledge*.

For example in Session BA1, driver B1 wants to know how well his partner understands the PHP script next to be reworked: *"You don't know the script at all, do you?"*. Interestingly, B2's reply is more complex than expected: *"Yes, I looked at it a little while ago and it's not actually complicated"* (*explain\_standard of knowledge+explain\_knowledge*). (To understand why *explain\_knowledge* appears as well, see Section 16.2.1.) 

### 14.2.2 AT with paraphrasing

In order to state and make sure that transferred knowledge has properly been received and understood, the speaker will sometimes paraphrase the knowledge in his or her own words, sometimes adding aspects that were not explicit before (sometimes even completing a cut-off or trailed-off utterance of the partner); see Example 14.1 (5+6). Such utterances are still considered purely *explain\_standard of knowledge*.

Example 14.1: Episode from Session BA1 (14:14:31–14:14:51) in which explained knowledge is paraphrased by extension to signal understanding.

#### (1) B2.*ask\_knowledge*

*"Can timestamp\_last\_change greater zero happen? (.) Er, er, less than zero, or equals zero for all I care. (.) Under this <\*points to screen\*> condition."*

B2 asks about the value of a variable.

#### (2) B1.*explain\_knowledge*

*"Hmmm, that is our requirement so to say that we have of this function (!!...!!)"*

B1 starts to answer but gets interrupted.

## (3) B2.mumble\_sth

“Should we not that there, so that not that there, I mean (!...!)”

B2 does not finish; it remains unclear what he intends to say.

## (4) B1.explain\_knowledge

“...that it returns zero, em.”

B1 ignores the interjection and continues his explanation.

## (5) B2.agree\_knowledge

“OK, hmhm.”

B2 agrees immediately.

## (6) B2.explain\_standard of knowledge

“If it doesn't, if it fails, somehow.”

B2 complements B1's explanation by a condition for the zero return. B1 knows that condition but had not mentioned it. B2's utterance states how he understood the explanation and what he assumes to know now.

## (7) B1.agree\_knowledge

“Exactly.”

This confirms the correctness of the standard of knowledge that B2 claims to have achieved, not the knowledge content, so it should be something like *attest\_standard of knowledge*, but we have decided against such a super-specialized and rare concept, using the much more generic *agree\_knowledge* instead.

Note that paraphrasing of recently transferred knowledge also occurs as part of proposals and should then not be annotated as *explain\_standard of knowledge*. See Example 9.2 where both uses of paraphrasing occur right after one another in utterances (2) and (3).

### 14.2.3 standard of knowledge in the making


We have discussed for *finding* (in Section 12.2.3) that sometimes speakers verbalize some of the thinking process that leads to a finding before the finding itself.



Something similar may happen before a speaker diagnoses his or her standard of knowledge, as in this long utterance of B1 in Session BA1: “Really great would be, if we could return something that always (!...!) (..) hm (...) Wait-a-sec. That

*always (!...!) (.....) Now I (~somehow) lost the thread.*" Such utterances should be annotated with *explain\_standard of knowledge*.

#### 14.2.4 *explain\_standard of knowledge* may be findings



A *explain\_standard of knowledge* utterance may result from a fresh insight: a finding. Because of the priority rules stated in Section 11.3, it is nevertheless simply annotated as *explain\_standard of knowledge*. This is particularly and undoubtedly true if the utterance does not actually explain the finding but only announces it, as in "*I have an idea*" from CA2. 

#### 14.2.5 Limited-knowledge proposals

As we have discussed in the respective chapters on the product-oriented and process-oriented concepts, a proposal is still a proposal if the speaker reveals that he or she is not fully convinced of the proposal.


However, the base layer's principle of encoding primary intentions (Section 2.3.2) demands that if the speaker apparently intends to inform the partner about a low or high standard of knowledge that underlay the proposal, this should be annotated separately as *explain\_standard of knowledge*. However, minor doubts are common and so this rule would lead to frequent double annotations if applied generally. Therefore, we apply it only if the standard of knowledge is expressed explicitly and separately – as in utterances (3)+(4) of Example 9.3 taken together.

#### 14.2.6 Implicit statements

Sometimes the standard of knowledge is not described explicitly and rather an interest in certain information is stated instead. Examples: "*I'd be interested in that myself*" (C2 in CA2, *explain\_standard of knowledge*) or "*I would like to understand it!*" (B1 in BA1, *explain\_standard of knowledge* and *propose\_step*, B1 starts investigating the respective source code right away).  

#### 14.2.7 Backward-looking statements

*explain\_standard of knowledge* utterances are not always about the present, they may also pertain to the past and sometimes even exclusively to the past (that is, the standard of knowledge has since improved).

For instance in Session CA2, after deleting a line of code C2 remarks "*Hadn't understood it anyway*" to declare a previous (and now irrelevant) lack of understanding of the meaning of the deleted code line. 

### 14.2.8 Signaling ongoing thinking

If a speaker reveals that he or she is currently thinking about something, this should be annotated with *explain\_standard of knowledge*.

For example in Session BA1, B1 explains the consequence of a previously proposed change as “Ehm, that would make it optional” to which B2 replies “That’s what I’m considering”.

## 14.3 Discrimination from similar concepts

We discuss various discriminations of only the concept *explain\_standard of knowledge* to concepts from the *finding* and *hypothesis* and the product-oriented and process-oriented concept classes. With one exception, the discrimination from *knowledge* concepts however will be discussed in Chapter 16.

### 14.3.1 *explain\_standard of knowledge* vs. *ask\_knowledge*

Any *explain\_standard of knowledge* that diagnoses a less-than-perfect knowledge level can also be interpreted as a request to transfer the missing knowledge: *ask\_knowledge*. So what should we annotate? Relying on the priority rules (Section 11.3), we would always chose *explain\_standard of knowledge*, because it is more specific. But another general rule should in fact take precedence here: The encoding of primary intentions (Section 2.3.2).

If the primary goal of the utterance appears to be obtaining knowledge by making the partner explain it, *ask\_knowledge* is the right concept. But if the primary goal is to simply inform the partner of the speaker’s knowledge-related situation, we use *explain\_standard of knowledge*. Both concepts will be used together only if we find both goal types to be equally strong, which ought to be rare.

### 14.3.2 *explain\_standard of knowledge* vs. *agree\_finding* or *disagree\_finding*

Any *agree\_finding* can be interpreted as an *explain\_standard of knowledge*, but based on the principle of encoding intra-dialog relationships (Section 2.3.6), the former should take precedence to make the episode structure more visible; see Example 12.5. This does not preclude an additional annotation with *explain\_standard of knowledge* if sufficient emphasis of this aspect is present in the utterance nor an exclusive annotation with *explain\_standard of knowledge* if the utterance is neither agreement nor disagreement but rather a statement of insufficient judgment capability.

The same holds for *disagree\_finding*.




### 14.3.3 *explain\_standard of knowledge vs. explain\_finding*


See Section 14.2.4.

### 14.3.4 *explain\_standard of knowledge vs. agree/disagree for a proposal*

We discussed in Section 14.2.5 that a proposal may lack complete conviction and said that such cases may or may not deserve additional annotation with *explain\_standard of knowledge*. The same idea holds for *agree* and *disagree* utterances (regarding proposals) that reveal a sufficiently unreliable underlying standard of knowledge.

### 14.3.5 *explain\_standard of knowledge vs. propose\_hypothesis*

Formulating a statement as a hypothesis always also provides a glimpse of the current standard of knowledge – do not let this fool you. Normally, the priority rules (Section 11.3) suggest that in such cases *propose\_hypothesis*, being the more specific concept, should take annotation precedence; see most of the examples in Chapter 13. 

For understanding the overall session, though, it might actually be most productive to consider the specific nature and origin of the ambiguity in a particular case. For example, in the statement “*Maybe I have even written some myself, I’m not sure*” the first part is clearly *propose\_hypothesis* while the second part could be considered *explain\_standard of knowledge*. However, you might come to the conclusion that in this particular case it is merely a re-emphasis of the “*Maybe*” at the front. 


In other cases, the principle of encoding primary intentions (Section 2.3.2) might suggest otherwise and should then be given priority: If signaling the insecurity appears more important than the actual content of the hypothesis, *explain\_standard of knowledge* should (also) be used.




## Universal concepts: *gap in knowledge*

### 15.1 Topic of *gap in knowledge* concepts

During our research, we found several cases where it appeared that moments in which the pair recognizes (and states) that *both* partners are lacking the same relevant piece of knowledge may be of particular importance for the further course of the session. We decided to create a separate concept class for this special case of *standard of knowledge*, called *gap in knowledge*.

For example in Session ZB7, driver Z20 summarizes the joint standard of knowledge as “*We don’t know whether our changes to the file (..) make it through to JBoss*” (*explain\_gap in knowledge*) and the observer confirms: “*Exactly*” (*agree\_gap in knowledge*). This is one of the two types of *explain\_gap in knowledge* utterance we have seen: the situation summary, where the underlying knowledge deficits have already been verbalized before and the utterance mainly puts them in a nutshell. 

The other type is asymmetric: The partner has (explicitly or possibly implicitly) declared a certain standard of knowledge and the speaker now declares his own by way of explaining a joint gap in knowledge. Here is an example of the ‘implicit’ subtype from Session ZB7: Z20 summarizes the previous discussion (about how to obtain a Topic object) as

*“The question is: Who provides us with a Topic? <\*Z19: JBoss!\*> JBoss, yes. JBoss provides it. Why does it do that? In which (.) file is it stated? And is that file generated by XDoclet or not?”* 

(*ask\_knowledge*, or more precisely: Z20.*ask\_knowledge*, Z19.*explain\_knowledge*,

Z20.*agree\_knowledge*, Z20.*ask\_knowledge*). The pair looks at one another silently for about 3 seconds before Z19 states



“*Very good question. Can’t remember anything.*”

(*explain\_gap in knowledge*), which declares the joint technology knowledge of the pair to be too low.

## 15.2 *gap in knowledge* concepts and their properties

As shown in Figure 3.2, we have observed only *explain\_gap in knowledge* and *agree\_gap in knowledge*, but obviously at least the respective *disagree*, *challenge*, and *amend* concepts can readily be added as soon as a corresponding utterance is encountered.

### 15.2.1 *explain\_gap in knowledge*

The base layer requires for an *explain\_gap in knowledge* phenomenon that the speaker not merely diagnoses a gap in joint knowledge but also apparently assumes that this gap is considerably relevant for the further course of the session.

## 15.3 Discrimination from similar concepts

We only discuss discrimination from *standard of knowledge* here (and from *propose\_step*) but postpone discrimination from *knowledge* to the respective Chapter 16.

### 15.3.1 *explain\_gap in knowledge* vs. *explain\_standard of knowledge*

Most *explain\_gap in knowledge* utterances can be considered to also fulfill the conditions for a *explain\_standard of knowledge*. According to the priority rules (Section 11.3), only *explain\_gap in knowledge* should be used and *explain\_standard of knowledge* should not be used in such cases.

Specialized studies may for instance want to vary the ‘speaker assumes relevance’ condition mentioned in Section 15.2.1 such that the gap must also *objectively* be considerably relevant. Such studies will also need to make the notion of ‘considerable relevance’ more concrete.


### 15.3.2 *agree\_gap in knowledge* vs. *agree\_standard of knowledge*

According to the principle of encoding intra-dialog relationships (Section 2.3.6), agreements to a previous *explain\_gap in knowledge* will always be annotated

with *agree\_gap in knowledge*, never with *agree\_standard of knowledge*, even if the speaker only talks of him or herself.

If for an agreement to a previous *explain\_standard of knowledge* you feel tempted to use *agree\_gap in knowledge* because the speaker extends the previous utterance's scope from the partner to the pair, then *explain\_gap in knowledge* is probably the right annotation instead.

### 15.3.3 *explain\_gap in knowledge vs. propose\_step*

An utterance such as "*Now we need to find out: (.) Where does that value come from?*" (BA1) will usually be a clear *propose\_step*. However, if (as in this case) it can appropriately be paraphrased as "*We should take time to think because we both do not know where the value comes from*" and recognizing that "*both do not know*" is a relevant insight, then *explain\_gap in knowledge* should be annotated in addition. Such paraphrasing is often helpful for finding the right annotation. 



## Universal concepts: *knowledge*

### 16.1 Topic of *knowledge* concepts

Now would be a fine time to re-read Section 3.6, which introduced the notion of 'knowledge' used in the base concept set, and Chapter 11.2, which explains it in some more detail. We repeat the key points here:

- *knowledge* concepts represent verbalizations of (some forms of) knowledge, not the knowledge itself and also not implicit forms of knowledge transfer.
- Most kinds of knowledge verbalization are addressed by other, more specialized concept classes (such as *finding* or *standard of knowledge*), not the *knowledge* class.
- The *knowledge* concept class serves as a kind of catch-all last resort, much like an else-clause, to accommodate utterances not captured by the other concept classes<sup>1</sup> (Section 11.3).
- *knowledge* utterances are truthful (the speakers believe them to be true) but not necessarily true.

The latter two conditions result in a large variety of phenomena being represented as *knowledge* in the base concept set. For instance the knowledge underlying a *knowledge* utterance may stem from a variety of *areas*, such as

- generic programming knowledge (such as design knowledge or technology details knowledge),

---

<sup>1</sup>Not all utterances: *activity* utterances remain.

- domain knowledge (problem domain),
- product-oriented project knowledge about properties of the program system being developed,
- process-oriented project knowledge about previous events, activities, and decisions regarding the development process not covered by P&P concepts,
- and others.

Also, there are various different *roles* of *knowledge* utterances (see again Section 2.3.2 on illocutionary acts and Section 2.3.2 on encoding primary intentions) and correspondingly different types of knowledge could be discriminated. Most prominently, many *knowledge* utterances explain the reasoning underlying a proposal and hence serve to justify the proposal. Such justifications could have been expressed by a verb *justify*, so that for instance *propose\_design* would often be followed by a *justify\_design* of the same speaker. Or consider the following scene from Session CA2: Driver C2 throws his arms in the air and exclaims “*I didn’t write that!*”. This is clearly a rejection of responsibility, which we could have considered important enough a phenomenon to allocate a separate concept class for it. But as has been discussed in Sections 3.6, 3.7, 4.2.3, and other spots, the base layer takes a different approach: It prescribes to simply use the generic (and somewhat vague) *explain\_knowledge* in those cases<sup>2</sup>.

In order to keep the size of the base concept set at bay, we also do not perform the area-based discriminations mentioned above or any other type of subcategorization and simply use *knowledge* for all of the variants. Since *knowledge* is an important concept class, this means that future studies will add their own concepts particularly often in this region, but note that it might be more convenient to add property concepts rather than splitting existing concept classes; see Chapter 22 for the discussion.

Example 16.1: Isolated (or mini-episode) examples of *ask\_knowledge* and *explain\_knowledge* utterances, all from Session BA1.

(a) B2.*ask\_knowledge*

“*But curl is on it?*”

Question whether the http utility program curl is installed on the development machine.

<sup>2</sup>And also for other types of justifications such as the “*I didn’t write that!*” that C5 claims with defensively lifted hands in CA2 after C2 criticized some code and which could have been expressed with a concept such as *explain\_responsibility*



**(b)** B1.explain\_knowledge

“Yes.”

Answer to question (a).

**(c)** B1.explain\_knowledge

“Well, we don’t call that. It has such a big overhead that I have left it out. Look, it’s commented out up here.”

B1 answers a question regarding the use of some existing functionality. The statement soon after turns out to be not entirely correct.

**(d)** B2.ask\_knowledge

“What do you do here? Here you check, er, whether, er, id\_code is a real key, real thingamabob?”

A question while displaying a particular piece of code.

**(e)** B1.explain\_knowledge

“Ahm, right. (.) That’s simply hex with 16 (.) digits.”

Answer to question (d).

**(f)** B1.explain\_knowledge

“friends\_ids. (.) There we have it; that’s why it did it. Writing comments razzes me. i d s and ids are easily confused, id\_codes you can’t confuse. And that function here *<\*highlights a call\*>* is called id\_to\_code anyway.”

Without having been asked, B1 explains names he has chosen prior to the session.

## 16.2 knowledge concepts and their properties

As shown in Figure 3.2, we have observed *explain\_knowledge*, *agree\_knowledge*, *disagree\_knowledge*, *challenge\_knowledge*, and *ask\_knowledge*.

### 16.2.1 Evaluations and judgments

Most evaluations or judgments of artifacts, parts or aspects of artifacts, or other items such as conventions are findings and require a *finding* concept. Sometimes, however, the speaker reveals that the respective insight is not recent and then *explain\_knowledge* ought to be used instead; the final PTo example in Section 14.1.1 is such a case.

### 16.2.2 Unprompted knowledge transfer

Speakers sometimes *explain\_knowledge* without prior query from the partner, as in Example 16.1 (f). Although this will often appear to be triggered by an insight (“*My partner needs this knowledge now.*”), no *finding* concept is annotated unless the insight is verbalized; usually, only *explain\_knowledge* will be used.

### 16.2.3 Rhetorical questions

Presumably-rhetorical questions for knowledge will not usually be annotated as *ask\_knowledge*. Rather, their illocutionary act needs to be determined: They may be the verbalization of an insight or knowledge (as in the DO example in Section 12.1.5), lead-in to such verbalization, or implicit suggestions what to work on (then use *propose\_step*, see Example 12.2 (3)).

### 16.2.4 Aggregation of utterances

The same rules apply as discussed for *finding* in Section 12.2.1; see Example 16.1 (c) and (f).

### 16.2.5 *amend\_knowledge?*

In contrast to findings, which are held together reasonably well by the short period in which the insight was achieved (but note our nevertheless simplified handling of *amend\_finding* in Section 12.2.5), it is very difficult to determine a sensible granularity that defines what is considered one piece of knowledge versus another (in particular because *knowledge* contains so many different things that are not clearly kept apart). It would therefore be overly difficult to provide a consistent operationalization of *amend\_knowledge* (in contrast to just using another *explain\_knowledge*) and we have thus not included *amend\_knowledge* in the base concept set.

### 16.2.6 “Different” answers

Not every question is answered in the manner expected by the asker. For instance, the reply may instead explain why an answer is not actually needed as in Example 16.2. Such phenomena are not particularly frequent and are not addressed explicitly in the base concept set; we simply use *explain\_knowledge* here as well.

Example 16.2: Episode from Session CA2 (12:04:53–12:05:13) in which there is a difference of opinion regarding a knowledge transfer.

(1) C5.*ask\_knowledge*/C5.*propose\_step*

“Do you know the (.) how I access the function to change a method?”

Said while opening the context menu for method `setVirtualAttributes`.  
Section 6.3.3 discusses this double annotation.

(2) *C2.explain\_knowledge/C2.disagree\_step*

“That doesn’t get you anywhere.”

C2 answers by giving an assessment of the idea *underlying* the question.  
Section 6.3.6 discusses this double annotation.

(3) *C5.challenge\_knowledge+C5.propose\_design*

“It does. In the method, once opened, I can turn the `IVirtualColumn` into `I(!...!!)`”

C5 insists on his idea, explaining his immediate goal. C2 interrupts him.

(4) *C2.challenge\_knowledge/C2.disagree\_step/C2.disagree\_design+C2.explain\_knowledge*

“That will (!...!) That doesn’t buy you much. But Alt-Shift-C, do, do it with Alt-Shift-C.”

C2 dissents again, but suggests an approach we decided to be the answer to the original question.

(5) *C5.challenge\_knowledge*

“Alt-Shift-C should be constants.”

The driver explains that the key combination will call some other than the required functionality.

### 16.2.7 Modes of agreement

An *agree\_knowledge* utterance can be made in at least the following three different modes:

- known: The recipient knew this before, as in Example 12.2.
- understood: The recipient signals to have understood the explanation and to consider it correct, whether after an actual verification or otherwise. See Examples 14.1 and 16.3 (4). Such utterances often border on *explain\_standard of knowledge*; see Section 16.3.10.
- unchecked: The recipient signals to be willing to assume the explanation to be correct without making his or her own judgment about it. See Example 16.4 (1). Such utterances often have the character of pushing away additional explanations or of delegating responsibility (see also Example 16.3 (6)) or they are preliminary.

We consider these modes to form different agreement types here, that is, properties that characterize agreement utterances; compare to the respective types for *agree\_finding* (and their role) in Section 12.2.9.

Example 16.3: Episode from Session CA2 in which the partner corrects explained knowledge.

(1) C5.*propose\_design*

*“As it is now (;) we can do without this one.”*

The driver highlights one line of code and suggests deleting it.

(2) C2.*explain\_knowledge*+C2.*agree\_design*

*“That wasn’t right anyway.”*

The observer agrees and states the logic in that line to be incorrect.

(3) C5.*challenge\_knowledge*

*“It was right before. It was no longer after your change. Before it it was.”*

The driver dissents and explains when the line had become wrong (which was before the session).

(4) C2.*agree\_knowledge*+C2.*disagree\_knowledge*

*“Uhuh, OK. (...) Actually not.”*

C2 agrees, then reconsiders and disagrees.

(5) C5.*explain\_knowledge*

*“That was (!...!) The, the, the strategy then was such that via (.), er, AllColumnAttributes we’d get the virtual ones as well. So I had to reset them and upon no change set them again.”*

C5 elaborates further.

(6) C2.*explain\_standard of knowledge*

*“(OK, I don’t get it, but OK.) I haven’t understood, but OK.”*

C2 gives up his attempt to understand and pushes off further explanations.

Example 16.4: Episode from Session CA2 (12:05:16–12:05:35) which follows directly after Example 16.2.

(1) C2.*agree\_knowledge*

*“Okayyy”*

The observer agrees (using mode unchecked) to the *challenge\_knowledge* of Example 16.2 (5).

### (2) C2.explain\_finding

*"Then you have changed that."*

C2 now recognizes (a TC-type finding) that C5 must have modified the Alt-Shift-C key binding.

### (3) C5.amend\_finding

*"t is <\*<developer name\*>. And we have the (.) agreement here, y' know, that we've set it so."*

Although it comes from existing knowledge and although it justifies, this utterance is amending the finding.

### (4) C2.agree\_finding+C2.explain\_knowledge

*"Yes, sure, well, who, who likes it, who likes to change it, sure."*

C2 agrees to the amendment and then evaluates its content which the base layer treats as explaining existing knowledge.

### (5) C2.agree\_step/C2.explain\_knowledge

*"Then do (!...!). You need to point here <\*<points to method name\*> (..) then 'refactor', 'change method signature'."*


C2 answers the question from Example 16.2 (1), agreeing to the step by explaining how to perform it.

## 16.2.8 Indicating agreement vs. indicating attentiveness

See Section 4.2.7.

## 16.2.9 Opposition and controversy

As for most base concept classes, explained knowledge can be contradicted in two manners:

- *challenge\_knowledge*: The explanation is refused in a manner that provides additional knowledge which either subjectively corrects the knowledge as in Example 16.3 (3) or which at least shows that the previous explanation cannot be correct as in Example 16.2 (5). For the latter case, later studies may decide that it would be better to annotate *disagree\_knowledge+explain\_knowledge* rather than *challenge\_knowledge*. 

- *disagree\_knowledge*: The explanation is refused without providing additional knowledge, as in Example 16.3 (4). Such utterances are often short.

### 16.2.10 Disagreeing by agreeing to the opposite

If an *explain\_knowledge* states a binary property, e.g. “*X does not have property Y*” and the partner disagrees in “*Yes, it does!*”-style, this should be considered offering alternative knowledge and thus be annotated with *challenge\_knowledge*.

### 16.2.11 Opinions

The base concepts considers an utterance expressing an opinion to express a belief. It should therefore be annotated with *explain\_knowledge*.

### 16.2.12 Limited conviction

While an *explain\_knowledge* is always uttered with apparently complete conviction (otherwise it would be *propose\_hypothesis* instead), the base concepts do not require complete conviction for subsequent *agree* and *disagree* utterances, which explicitly or implicitly are allowed to reveal remaining doubt; see Example 9.3 (8). The example “*Really?*” from BA1 illustrates that this even results in cases where it is difficult to decide between annotating *agree\_knowledge* and annotating *disagree\_knowledge*.



### 16.2.13 *ask\_knowledge* is not always that

Many *explain\_knowledge* utterances are triggered by a previous question, as in Example 12.2 (4), see also (6). But such questions are to be annotated as *ask\_knowledge* only if they do not imply a proposal, as Example 12.2 (3) does, and also do not pertain to P&P concepts.

### 16.2.14 Questions including possible answers

Speakers sometimes phrase questions that include a hypothesis for the answer, such as Example 16.1 (4). If such a hypothesis appears to be present merely for clarity or other rhetorical purposes, we simply annotate *ask\_knowledge*. A *propose\_hypothesis* is added only if the hypothesis appears to be meant as such (perhaps as the primary aspect then); see Section 16.3.6.

### 16.2.15 Statement or question?

Even when fully exploiting the information available from context and intonation, it is sometimes hard to decide whether an utterance should be taken as a statement for explaining knowledge (*explain\_knowledge*) or as a question asking

for verification (*ask\_knowledge*). It may even be that neither is the case and the speaker is actually formulating an insight to be exploited (*explain\_finding*).

The best example we have is untranslatable into English because it rests on the use of the German word “ja” (normally “yes”) as a modal particle. With that word left in, the half-translated utterance of C2 from Session CA2 goes

*“The processing of the, the thread IDs stays [ja] the same, there is [ja] nothing (!...!)”*



C2 says this unprompted after he stated the pair could now proceed (after a successful test). C5 replies “Exactly”. The “ja” can mean “as you know”, which would lead to the meaning *explain\_knowledge*. Or it can mean “I think”, which would allow for *ask\_knowledge* or *explain\_finding*. Or it can mean “surprisingly”, which would suggest *explain\_finding*. The nature of *knowledge* as the fallback concept class suggests to annotate such ambiguous cases as *explain\_knowledge*.



## 16.3 Discrimination from similar concepts

Except for a very few bits and pieces, everything needed to correctly apply the *knowledge* concept class and to discriminate its instances from those of the previous classes has been said above. But because of the importance of *knowledge* utterances and the subtlety of the class (due to its catch-all character), we nevertheless discuss a large number of cases explicitly at least shortly here.

### 16.3.1 *explain\_knowledge* vs. *propose\_step*

A longer utterance containing a step proposal can involve three aspects: The actual step proposed, a justification of why it should be performed, and an additional explanation of how it could or should be performed. Only the first of these is annotated as *propose\_step*, the other two parts, if present, must each be annotated with an appropriate *knowledge*, *finding*, or perhaps *hypothesis* concept. Without a separate proposal, the otherwise exact same explanation *how* to do “it” can be considered the proposal.

### 16.3.2 *explain\_knowledge* vs. *propose\_design*

If design proposals are accompanied by a separate justification, separately annotate the latter as *explain\_knowledge* as discussed in Sections 3.7 and 4.2.3: *propose\_design+explain\_knowledge*. If the proposal is implicit in the *knowledge* utterance, use a double annotation *propose\_design/explain\_knowledge*; see the discussion in Section 4.3.2.

### 16.3.3 *explain\_knowledge* vs. *explain\_finding*

As we have discussed at length in Chapter 12, we use *finding* not only for utterances that contain findings explicitly, but in many other cases as well. Five remarks remain to be made:

1. If *finding* and *knowledge* appear equally likely, then *explain\_knowledge* should be used: we may lack any symptom at all that would let us favor *finding* over *knowledge* or vice versa, as in the following example from BA1 where B1 asks “*What will happen here when this friend here deletes himself <\*points to a paper note\*>? (.....) Then strictly speaking I have to, then strictly speaking I need to (~invalidate) mine, my friends list.*” and B2 answers without any hesitation

“*That it does, yes. It, you are friend (~too) of him who deletes himself. (.) And here in this function will be (.) for all (.) of his friends, including you, (.) the MemCache object will be (~deleted).*”

You may perceive the promptness of the reply to suggest existing knowledge as the content of the reply, but it may as well be a finding created during the five seconds of silence within the previous utterance of B1. We simply cannot decide the matter and will annotate *explain\_knowledge* as a result. Such problems are common in particular for (possible) findings of the idea types (TC and even more so TU).


2. Even if a linguistic *finding* symptom such as “Ah, ...” is present in the utterance, it may only signal an insight of the type “*I just recognize that you need the following information*”, but the base layer does not annotate such minor aspects of utterances. If the content of the subsequent utterance is existing knowledge, the whole utterance should be annotated with *explain\_knowledge* and only with *explain\_knowledge*.

3. It is normal that existing knowledge is required for having an insight. It is therefore also normal that such existing knowledge is verbalized as part of the verbalization of the insight. Such verbalizations should normally be annotated simply as *finding*, except if the speaker explicitly marks some part of it as existing knowledge, in which case a separate annotation or double annotation of *explain\_knowledge* is called for.

4. Explaining existing knowledge may itself be the trigger for an insight – a potentially very important phenomenon for understanding pair programming. If this happens, both parts should separately be annotated accordingly as after the self-interruption in the following B1 utterance from Session CA2:


“*The problem is, er, if you want to, er, use the (.) Columns, and in all too, you need to have the same attribute column (.) names. And to have that, you need to (!...!) Oh, no, stop, that's not a problem: We have them here.*”



5. The speaker's idea that existing knowledge should be explained will often be a finding. If this idea itself is not verbalized, however, the base layer suggests to annotate *explain\_knowledge* only. Since the idea may be important for the session, subsequent studies may decide to change this decision. 

Many of these remarks apply not merely to *explain\_knowledge* and *explain\_finding*, but likewise to *amend\_finding*, *challenge\_finding*, and *challenge\_knowledge*.

#### 16.3.4 *explain\_knowledge* vs. *amend\_finding*, *challenge\_finding*, *disagree\_finding*

Elaborations as part of *explain\_finding* as well as parts or all of *amend\_finding* or *challenge\_finding* utterances may consist of existing knowledge. In the base layer, this fact is not annotated; according to the episode principle (Example 2.3.6),  only the respective *finding* concept is used. Later studies may want to introduce properties to be annotated additionally in order not to lose the information about the origin of the knowledge in such cases.

Accordingly, a *disagree\_finding* elaborated by a justification based on existing knowledge will be annotated as *challenge\_finding*.


#### 16.3.5 *explain\_knowledge* vs. *agree\_design/disagree\_design*

Rather than to *agree* or *disagree* with a design proposal explicitly, the speaker may package his opinion in a knowledge utterance. As this may be a relevant phenomenon for pair programming, the base layer suggests to use double annotations (e.g. *disagree\_design/explain\_knowledge*) in such cases.

The same idea applies to all P&P proposals, but in our data we have observed such behavior only for *design*.

#### 16.3.6 *ask\_knowledge* vs. *propose\_hypothesis*

We have previously allowed that a *hypothesis* utterance may take the form of a question (as in the example below). We have also allowed that an *ask* utterance may include possible answers (as in Example 16.1 (4)). Therefore, there may be ambiguity whether an utterance is asking for knowledge or proposing a hypothesis. The base layer rule for these cases states to use *ask\_knowledge* iff it appears that the speaker believes the partner will be able to reliably answer the question (and use *propose\_hypothesis* otherwise).

Here is an example from BA1: After the pair changed an argument supplied to a PHP script, the observer wonders what the result will now be: "*Is greater. (~So we) should get an exit, not a (~friends list), or not?*" Here it is unlikely that the speaker assumes definite knowledge in his partner, in particular since that 

partner had attempted (and aborted) an explanation himself previously, so *propose\_hypothesis* is the right annotation.

In contrast in Example 16.1 (4), while the second part sounds a lot like *propose\_hypothesis*, the first part is a direct question to the partner, so *ask\_knowledge* it must be. Example 16.5 is another illustration for how similar these two concepts can be.

Example 16.5: Episode from Session CA2 (12:29:04–12:29:10) showing the similarity of *ask\_knowledge* and *propose\_hypothesis*.

(1) C2.*ask\_knowledge*

*“That’s how it went, with a minus, right?”*

The pair is filling an SVN commit message and wonders about the prescribed syntax for the task ID. C2 has typed ‘cad-509’ and has moved the cursor back to the minus sign.

(2) C5.*propose\_hypothesis*

*“I think the ‘cad’ is upcased or so?”*

As C2 is apparently unsure about the format, C5’s question must be a hypothesis.

### 16.3.7 *ask\_knowledge* vs. *explain\_finding*

Questions are typically the result of a more-or-less recent insight which told the speaker that he or she is lacking relevant information or understanding. Such a tacit underlying insight is not annotated unless the question must be considered rhetorical in which case the question is not annotated as such. A borderline case occurred in Session BA1, where the driver encounters tabulator characters in the source code and asks *“What are these TABs here?”*. *explain\_finding* and *ask\_knowledge* are equally adequate annotations for this utterance, see Example 12.4 for the context.



### 16.3.8 *explain\_knowledge* vs. *explain\_standard of knowledge*

As discussed in Section 14.2.2, if a speaker paraphrases knowledge previously explained by the partner, this will be annotated as *explain\_standard of knowledge* even if it includes additional bits of knowledge as in Example 16.6 (2). This rule was chosen for practicality. Its limit is reached when the parts can clearly be identified as separate instances in which case *explain\_knowledge* should be used for the respective part. Later studies may wish to find more refined rules in this area.



Example 16.6: Episode from Session CA2 (11:53:01–11:53:23) in which C2 agrees to an explanation by means of *explain\_standard of knowledge*.

(1) C5.*explain\_knowledge*

*“I said, I (...) I (.), er, work different from you. I (...) including along. I have included in the considerations that they from that getColumnAttributes may, that they are included there. That’s why they’re reset there now.”*

The observer “explains” the code he wrote before the session. He points to spots visible on the screen.

(2) C2.*agree\_knowledge*+C2.*explain\_standard of knowledge*

*“Um, then you can easily, then you can easily change it. (!!That had confused me!!)”*

C2 agrees by explaining how he understood the explanation. C5 interrupts him during the last part.

(3) C5.*explain\_knowledge*

*“ (!!Yes!!)”*

C5 states that C2 understood him correctly. (There is no specialized base concept for such utterances.)

### 16.3.9 *ask\_knowledge* vs. *explain\_standard of knowledge*

See Section 14.3.1.

#### 16.3.10 *agree\_knowledge* vs. *explain\_standard of knowledge*

An *agree\_knowledge* is sometimes ambiguous versus an AT-type *explain\_standard of knowledge*. In such cases we stick to the episodes principle (Section 2.3.6) and annotate *agree\_knowledge* unless both possible intentions are made somewhat explicit, in which case we either make a double annotation as in Example 16.6 (2) or split up the utterance as in Example 14.1 (5+6).



## Universal concepts: *activity*


### 17.1 The notion of facade concept class

The design pattern **Facade**<sup>1</sup> [7] defines a construct (usually a class) that provides an appropriately simplified view of a set of other constructs (such as a set of classes), omitting detail and focussing attention on the parts relevant for the purpose at hand.

The base layer borrows (rather loosely) from this idea to introduce the notion of a facade concept class. A *facade concept class* contains concepts that provide a simplified perspective on a part of a session. That part is

- somehow coherent for the purpose of the analysis,
- but consists of more than one utterance or of an utterance and something else (and often both).

Such facade concepts will typically not be used alone: they are designed to commonly be used as one part of a double annotation. The elements of any one facade concept class together may or may not provide an episode structure as the normal HHI concept classes do.

In the base layer, *activity* is the only facade class, but others might conceivably be introduced in later studies. 

### 17.2 Topic of *activity* concepts

The *activity* concepts represent utterances that comment on HEI or HCI activities (Chapter 19) currently or recently going on, for instance expressing

<sup>1</sup>[http://en.wikipedia.org/w/index.php?title=Facade\\_pattern&oldid=559454643](http://en.wikipedia.org/w/index.php?title=Facade_pattern&oldid=559454643)

agreement or disagreement with actual code changes just performed by the partner. They thus link the HHI dialog world to the HCI/HEI world.

The core concept is *think aloud\_activity*. Such utterances explain on the fly all or part of what the speaker is concurrently doing<sup>2</sup>. One *think aloud\_activity* addresses exactly one activity (but several *think aloud\_activity* annotations may be made for subsequent stretches of the same activity). The verbalization can be descriptive. Or it is explanatory, reflecting, or otherwise adding (whether consciously or not) information beyond what the partner could derive by pure observation otherwise. In particular, the verbalization may constitute one or more instances of HHI concepts. These will be annotated separately and the *think aloud\_activity* serves to capture their relationship to one instance of an HCI/HEI concept. For example in Session CA2, C2 is modifying a method:



```

<*starts modifying the method*>
It ought to be implemented like this. (;;;)
<*stops modifying the method*>
(.)
<*starts wandering up and down with the cursor*>
And it must not, definitely not, be part of Abstract. Or so I think.
<*stops wandering up and down with the cursor*>”

```

The activity being verbalized here is a *write\_sth* (see Section 19.1), the two HHI utterances are both *propose\_design*, and the *think aloud\_activity* relates those three parts to one another.



Note the cursor wandering may in fact be a form of **displacement activity**<sup>4</sup>. The base layer’s model of HCI/HEI activities is coarse, so it provides no concepts for modeling such behavior, but subsequent studies may want to introduce appropriate extensions.

A *think aloud\_activity* may start a bit before the activity or continue a bit after its end as explained in Figure 17.1. In any case, the base layer suggests a new activity always requires a new instance of *think aloud\_activity*.



The concept does not say anything about what the content of the verbalization is, how complete it is, or whether the resulting information or integration of the partner is a conscious goal or not. If the verbalization comes in several parts with pauses in between, multiple instances of *think aloud\_activity* may be used. These rules will be elaborated in Section 17.3.

A *think aloud\_activity* utterance typically addresses one or more of the following:

<sup>2</sup> Starting to verbalize is typically a spontaneous decision of the speaker, so this concept is different from what happens when using the **think aloud research method**<sup>3</sup> where the speaker is repeatedly requested to verbalize.

<sup>4</sup>[http://en.wikipedia.org/w/index.php?title=Displacement\\_activity&oldid=558153583#Use\\_in\\_science](http://en.wikipedia.org/w/index.php?title=Displacement_activity&oldid=558153583#Use_in_science)

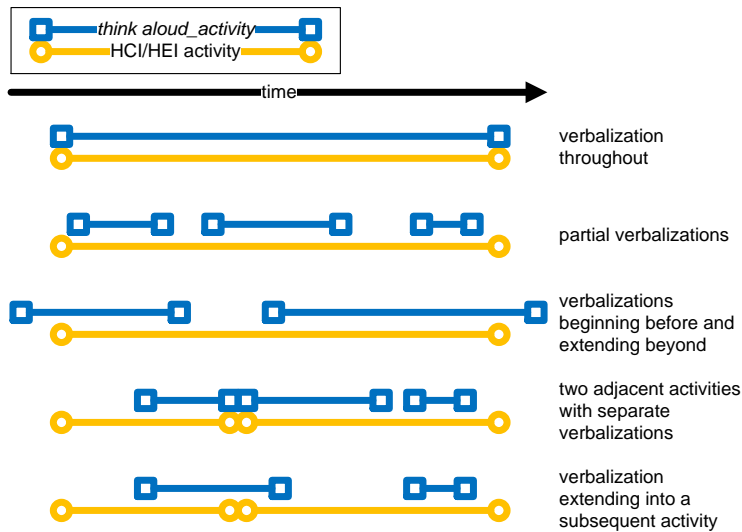



Figure 17.1: How activities may relate to *think aloud\_activity* utterances.

- TA1: What am I doing? This may include verbalizing typing input. Will often involve *propose\_design*.
- TA2: Why am I doing what I am (or soon will be) doing? Will often be *explain\_knowledge* (or *finding*).
- TA3: How am I doing what I am (or soon will be) doing? Will typically be *explain\_knowledge*.
- TA4: What decisions am I making? Will often be *propose\_design* or *propose\_step* (but also *amend* and *challenge* and sometimes others).
- TA5: What decisions underlie my activity? Often involves a *strategy*.
- TA6: What fresh insights am I having? Will usually be *finding* utterances.
- TA7: How do I interpret and evaluate feedback the computer is providing? Also *finding*.
- TA8: How sure am I to be doing the right thing? Tends to be *explain\_standard of knowledge*.
- TA9: How do I judge the quality of what I am doing? Tends to be *finding*.
- TA10: How far have I come in what I am doing? Typically *completion*.

- TA11: How does the infrastructure (development tools and runtime tools) influence what I am doing? Typically *knowledge* or *finding*.

Here are a number of examples from Session **BA1**. The mentioned HCI/HEI concepts will be introduced in Chapter **19**.


1. B1 modifies a URL in a shell script call and remarks on two differences to his usual workstation setting. B2 makes an interjection and B1 replies without stopping the editing. Length 33 seconds:

 “<\*start write\_sth\*>  
<\*TA11:\*> I need to adjust each time to not having an apple-key. (;)  
<\*TA4:\*>Here. We simply use localhost, I'd say. (;;;;) (And then it is) (!...!)  
<\*B2: You call it from the shell, don't you?\*>  
Yes. (;) <\*TA1: reads his inputs aloud\*> projects, um, <\*\*\*URL part\*\*>  
(..) trunc (.) API (;) <\*\*\*URL part\*\*> (~~)  
<\*end write\_sth\*>  
<\*TA11:\*>Mighty small, that screen.”

2. B1 has opened the Eclipse Preferences window and is expanding several tree nodes. Length 3 seconds:

 “<\*do\_sth is running\*>  
<\*TA2:\*> Hmmm. Fonts and Colors must be somewhere.  
<\*do\_sth continues\*>”

3. A short time later Fonts and Colors is found. B1 wants to make the editor font smaller to get better overview in the code. He eventually finds that his Courier font size change to the smallest value 10 did not help much. Length 21 seconds:

 “<\*do\_sth is running\*>  
<\*TA4:\*> Let's see. What does Courier have? (;;;;;;;;;;) Bah! (;)  
<\*TA10:\*> Not a big improvement.  
<\*TA6:\*> The problem is it cannot go much smaller.  
<\*end do\_sth\*>”


4. Elsewhere, B1 opens a file by double-click, but it opens in the external editor TextPad rather than the internal Eclipse editor. Another attempt via the context menu has the same result. The third attempt uses a different context menu entry. Length 27 seconds:



```

"<*start do_sth*>
(;;) <*TA5:*> OK, we don't do that with TextPad (;;) No way. (;;)
<*TA6:*> We need to tell it with what to open it. (;;)
<*B2: Can't you drag-and-drop? (~~)*>
Oh God. (;)
<*TA3:*> We need to use this: Open in Editor.
<*do_sth continues*>"

```




5. Still elsewhere, B1 has changed a configuration file and once again calls `wget http://dev-intern` in a shell. The call terminates with the message “connection refused”. Length 13 seconds:

```


"<*start verify_sth*>
(;;) <*TA4:*> Then we can try it again. (;;;)
<*TA7:*> OK, the connect still doesn't like us. <*end verify_sth*>"

```



## 17.3 activity concepts and their properties

As shown in Figure 3.2, we have observed *think aloud\_activity*, *agree\_activity*, *disagree\_activity*, *challenge\_activity*, *amend\_activity*, and *stop\_activity*. While *think aloud\_activity* refers to activity of the speaker, all others refer to activity of the partner; see Section 17.3.14 for corner cases.

A joint goal for most of the rules for using these concepts, in particular *think aloud\_activity*, is not to allow the granularity get too small. Subsequent studies may need to deviate from the rules wherever they blur the relevant phenomena too much, e.g. a study interested in the role of fluent versus halting speech. 

### 17.3.1 Granularity of *think aloud\_activity*

A core question for the *activity* class is to determine the beginning and end of a *think aloud\_activity*. We answer it by answering a whole sequence of subquestions as follows.

- When do we consider an HCI/HEI activity to begin or end? This will be discussed for each type of activity in Chapter 19. In short, the answer is to be found neither by looking at syntactical elements of the artifacts nor by looking at contiguous stretches of physical activity but rather by considering logical work steps.
- How to handle verbalizations beginning before or ending after the HCI/HEI activity (Figure 17.1)? If the protruding part is only a part of an utterance, it should usually be considered to be part of the *think aloud\_activity* iff the utterances are comments going along with the activity. Such prefixes and suffixes should be kept short, however. For instance in Example 5 above,

the *verify\_sth* could technically have been considered complete *before* the last utterance, but for understanding it is more helpful to extend the *think aloud\_activity* to fully include it.

- How many aspects relevant for the activity can be left unsaid before we stop calling it a *think aloud\_activity*? How many need to be verbalized at least? The number is irrelevant; only one aspect needs to be verbalized.

- How continuous and contiguous does a verbalization need to be (gaps and pauses, Figure 17.1)? Arbitrary gaps and pauses are acceptable as long as the same verbalization intent still appears to be at work. Only if the speaker appears to stop the verbalization entirely and then begins a new one will we annotate a new *think aloud\_activity*. The behavioral details of this difference are difficult to describe but are often not difficult to decide for an actual researcher.

- How to handle deviations from the main topic? As long as the deviation is semantically connected to the activity's topic at all, the *think aloud\_activity* is allowed to continue. For instance in Session BA1, there is a point during an editing step where the actor explains a syntactic improvement to the PHP language that he would like to see introduced.

- How to handle interjections from the partner? If the actor reacts to them casually or not at all, they are not considered to break the *think aloud\_activity*. If the actor reacts to them by shifting her attention focus, the *think aloud\_activity* ends. Subsequent studies may wish to change either of these decisions.

### 17.3.2 *think aloud\_activity* phenomena leading to questions

If the actor poses a question to the partner during a *think aloud\_activity* and continues verbalizing right afterwards, the question is not considered to break the *think aloud\_activity*. If there is a speaking pause after the question, the rules for gaps and pauses from Section 17.3.1 apply. Subsequent studies may want to introduce their own rules for this aspect.


### 17.3.3 HCI/HEI activities resulting from an utterance

*think aloud\_activity* represents utterances that accompany an HCI/HEI activity. In the converse case, an activity may accompany an utterance (most commonly pointing to or highlighting on the screen). *think aloud\_activity* does not apply to such situations.

Subsequent studies for which such auxiliary activities are important should introduce an appropriate class of facade concepts to express the relationship. An example might be a study interested in misunderstandings due to missing pointing and highlighting.

### 17.3.4 Disconnect of HCI/HEI activity and verbalization

If utterances of the actor during an activity have no discernible semantic relationship to the activity, *think aloud\_activity* is not applicable. If a *think aloud\_activity* is already ongoing, the utterance(s) should be treated like pauses and gaps (Section 17.3.1).

Subsequent studies may want to introduce a new facade class for such phenomena if they occur regularly. 

### 17.3.5 The partner commenting on activity vs. on verbalizations

Use *agree\_activity*, *disagree\_activity*, *challenge\_activity*, *amend\_activity*, and *stop\_activity* when the partner comments on what the actor is doing. They do not apply if the partner only comments on the actor's *verbalization* of what he or she is doing.

Note that many activities have so little physical "body" that activity and verbalization are hard to keep apart and so the above problem is less likely to occur.

### 17.3.6 *challenge\_activity*

The counter-proposal formulated in a *challenge\_activity* may pertain to process aspects of the activity (procedure, as in Example 17.1), to product aspects (content, as in Example 17.2), or to both.

Example 17.1: Episode from Session BA1 (13:46:49–13:47:01) containing a procedure-related *challenge\_activity*.

#### (1) B1.*propose\_hypothesis*

"Ah, maybe we must enable the chat."

B1 is already in an B1.*explore\_sth*. This utterance starts a B1.*think aloud\_activity*.

#### (2) B2.*agree\_hypothesis*

"Um, or so."

This does not break the B1.*think aloud\_activity*.

#### (3) B1.*propose\_step*

"That's a possibility. We need the auto\_prepend."

This continues both the B1.*explore\_sth* and the B1.*think aloud\_activity*: B1 moves the Eclipse Navigator View's highlight cursor to the file auto\_prepend. B1.*explore\_sth* ends. B1.*think aloud\_activity* ends.

**(4)** B2.challenge\_step

*"No, that's in the Conf."*

This constitutes a B2.challenge\_activity.

**(5)** B1.propose\_step

*"I'll first make a Working Set."*

B1 calls the Eclipse function Select Working Set (B1.do\_sth and B1.think aloud\_activity).

Example 17.2: Episode from Session **BA1** (14:30:39–14:30:47) containing a product-related challenge\_activity.

**(1)** B2.propose\_design+B2.explain\_completion

*"return 0, but otherwise; exactly."*

B1 is modifying a return statement (B1.write\_sth). B2 proposes something about its content (B2.amend\_activity), B1 continues editing, B2 finds his proposal to be implemented (B2.agree\_activity).

**(2)** B2.propose\_design

*"No, that was quite right."*

B1 selects a part of the return statement do delete it (B1.write\_sth continued), B2 requests to leave it as is (B2.challenge\_activity).

**(3)** B2.explain\_standard of knowledge

*"Aaah, you want to do it up there."*

B1 performs the deletion (B1.write\_sth continued), B2 recognizes his plan of action and agrees with it (B2.agree\_activity).

### 17.3.7 agree\_activity, disagree\_activity

Activity agreements are often very short utterances. See both a long and a short one in Example 17.2.

Activity refusals that come without a counter-proposal (disagree\_activity) are often very short utterances and just like challenge\_activity may pertain to process aspects or product aspects. For instance in **CA2** the driver is about to click 'Commit All' in an SVN commit operation when the observer says "Not All".



### 17.3.8 Comments before the fact

The disagreement in the above example was uttered before the deed it referred to had actually begun. This was possible because the speaker could see the preparations and infer the intention. Such behavior is possible for *agree*, *disagree*, *challenge*, *amend*, and even *stop\_activity*.

### 17.3.9 Comments after the end

An *agree*, *disagree*, *challenge*, and *amend* may be uttered when the respective activity has already terminated. Such annotations are acceptable if the gap is sufficiently short and there are other symptoms as well that suggest the comment pertains to the activity itself (such as the act of searching) rather than only its result (such as the search hits found).

### 17.3.10 *amend\_activity* vs. *challenge\_activity*

When the partner comments on an activity constructively (that is in a proposal-like manner), it is not always easy to decide whether that includes an element of criticism (*challenge\_activity*) or not (*amend\_activity*). Example 17.3 illustrates such ambiguity.

The base layer suggests to use *amend\_activity* whenever no controversy is apparent in the utterance itself nor in the partner's reaction, and also use it when the actor had previously hesitated (and even if controversy ensues afterward). Subsequent studies particularly interested in such phenomena, however, will need to develop their own, much finer set of rules here.

Example 17.3: Episode from Session CA2 with high ambiguity. The pair has discussed how to modify a particular interface. The respective file is open.

#### (1) C5.mumble\_sth (semantics unclear)

"OK. (;;;;) That would be (!...!)"

The OK may signal a start of something (C5.think\_aloud\_activity begins). C5 navigates to method setVirtualColumns in the Eclipse package explorer and hovers there (C5.explore\_sth).

#### (2) C2.propose\_design

"getVirtualAttributes then maybe."

C2 suggests what to modify. This should be C2.amend\_activity if the researcher has not seen a different action plan in the activity of C5 and should be C2.challenge\_activity otherwise. C5 promptly starts editing the method name (C5.write\_sth).

### 17.3.11 *stop\_activity*

If a speaker disagrees with an activity that has already started but not yet ended, and disagrees with the whole of it rather than only some aspect, we do not use *disagree\_activity* but rather the specialized *stop\_activity*: An explicit appeal to entirely terminate the current activity.



For example in Session CA2, the driver is testing a class by means of the JDemo GUI testing framework, when observer C5 says “OK, but *what we did has more influence on the Action (.) than on (.) the GUI. So (.) that means, I'd rather go from <application name>.*” This is an C5.*explain\_knowledge* leading to a C5.*propse\_step*, the latter constituting a *stop\_activity*.

### 17.3.12 Interjections leading to activity change

Activity comments need not be *stop\_activity* to lead to a change of activity. Example 17.3 may be such a case (but possibly the change would also have happened without the interjection).

### 17.3.13 *think aloud\_activity* by the “observer”

*think aloud\_activity* requires actorship and so most commonly is performed by the driver. In practice, however, the “observer” also may look something up in a book, make a sketch on paper, or do any of many other things that entitle him or her to *think aloud\_activity* as well.

### 17.3.14 Self-criticism

The verbalization occurring during an activity (*think aloud\_activity*) may include elements such as

- self-evaluations of the activity which have a positive result, reinforce the activity, and would be annotated as *agree\_activity* if they came from the partner;
- self-evaluations with negative result that flag some aspect as deficient and would be annotated as *disagree\_activity* if they came from the partner;
- self-evaluations with negative result that find the whole activity to be the wrong idea and that would be annotated as *stop\_activity* if they came from the partner;
- mental notes to do something else as well that would be annotated as *amend\_activity* if they came from the partner;
- self-corrections of approach that would be annotated as *challenge\_activity* if they came from the partner.

Such an annotation style would turn the currently straightforward *think aloud\_*  
*activity* annotations into something much more complicated and therefore the  
base layer suggests not to make them.

Subsequent studies particularly interested in reflection phenomena will need   
to change this decision and possibly also further refine the concepts used.





## Universal concepts: Miscellaneous

There are two HHI concepts that have only little semantic content: *mumble\_sth* and *say\_off topic*. They only serve to mark spots in the session where some element of the dialog is largely ignored by the annotation.

### 18.1 *mumble\_sth*

*mumble\_sth* is used for utterances that cannot be annotated otherwise because they are incomprehensible, whether for acoustic or phonetic reasons (which depending on recording quality and background noise may not be rare at all) or for semantic ones. Semantic reasons may be incompleteness (often cut-off or trailing-off utterances) or plain gibberish.

Note that what appears to be *mumble\_sth* for the researcher may well be understandable for the pair partner, even the gibberish. If the partner reacts to the utterance, the reaction can often be used to reconstruct the illocution (if not the wording), and then the concept appropriate for that illocution should be annotated even if the utterance itself could not be understood; use *mumble\_sth* only if there is no way to reconstruct the meaning. Be aware that this approach can occasionally lead to misinterpretations; see the discussion in Section 21.7.3.

### 18.2 *say\_off topic*

*say\_off topic* represents utterances on topics that have nothing to do with the goal of the session such as matters of private life or events in the wider work environment. For instance in Session CA2, C5 makes a remark regarding the fact that the session is being recorded: “*I wonder what they will think, who don't know what we are doing here*”. We should mention that such remarks and in



fact any indication whatsoever that the pair is still aware of being recorded is generally rare after the starting moments of a session throughout our data.


## **Part III**

# **Other concepts**


... in which we define the concepts that aim at non-verbal interaction employing the computer, paper and pencil, the index finger, and possibly other useful equipment.



## The HCI/HEI concepts

An annotation with an HCI/HEI concept represents an interaction of a pair member with the computer (HCI, human-computer interaction) or the remaining environment other than the partner (HEI, human-environment interaction). There are currently only eight simple interaction types or purposes (*write, search, explore, verify, read, sketch, show, and do*), the concepts for which all use the same, generic object *sth* (short for *something*) only. This leads to a coarse modeling of HCI/HEI activities, which is appropriate because the base layer puts its focus on utterances. The modeling is just enough to serve as the nucleus of a fuller modeling of HCI/HEI issues. Later studies interested in action (rather than only in dialog) will have to introduce a additional concepts. 

The *maximum* granularity of an HCI/HEI annotation is called one *activity entity*. An activity entity is a sequence of actions guided by one short-term goal. The goal can be explicit (e.g. defined by a previous *design* or *step* episode) or tacit; the sequence can be expressed by one HCI/HEI annotation or several; the base layer does not model the activity entities themselves.

The above granularity notion can lead to near-arbitrary segmentation for tacit goals, cannot cope well with goals that change during execution, and hardly gets a grip on the way in which the execution may occur in multiple, unctiguous stretches of time. 

The notion of activity entity is central to understanding the HCI/HEI concepts: Although these concepts often represent physical activities, they are not meant to characterize these physical activities *as such*. Rather, a respective annotation should cover the intellectual process that underlies the observable physical activity (if any) and holds together its parts, see Example 19.1. This is difficult to do properly, given our behavioristic ideal (Section 2.3.4), but we at least attempt a reasonable approximation.

Here is a short overview of what the verbs mean (in the order of their subsequent description):

- *write*: editing product (or product-related) computer artifacts
- *search*: manual or automated search for well-defined target items
- *explore*: investigate and analyze in or among artifacts
- *verify*: checking the suitability of the modifications performed on artifacts
- *read*: reading aloud information shown on the screen or in paper documents
- *sketch*: graphical sketching on paper or by computer
- *show*: pointing to elements or sections of physical or non-physical artifacts
- *do*: everything else not covered by the above concepts

## 19.1 *write\_sth*

*write\_sth* addresses writing activity in product artifacts (such as program code) or product-related artifacts (such as configuration or documentation files).

Writing activity should not be confused with typing. While typing is a necessary part of it, activity connected to the typing is also part of the writing activity, such as proofreading, correcting, near-range screen navigation, and in particular thinking (before typing, in between, and even after). The visibility of several of these parts is limited, so we will need to use context indicators and apply our best judgment. For instance in the first example on page 168, after typing and speaking the word “projects”, the actor hesitates for about a second before further typing, meanwhile saying “um”, and later makes two more typing pauses during which he remains silent. Neither of these, however, should be considered to terminate the *write\_sth* because from a semantic point of view it all clearly belongs together (which may not appear obvious in the transcript provided, but was reasonably obvious for us in the actual session recording).

The editing aspect of *write\_sth* needs not be fulfilled by direct keyboard input; semi-automated mechanisms such as search-and-replace or the application of a refactoring operation also count.

A *write\_sth* ends when the goal is reached or substantially changed, when a break occurs, when the driver changes, and when some other HCI/HEI concept needs to be annotated for this actor. It does *not* automatically end when HHI concepts need to be annotated or when an HCI/HEI concept is annotated for the partner. Here is a complex case of *write\_sth*:

Example 19.1: Episode from Session BA1 (starting 13:55:24) in which a lot of dialog occurs during a single *write\_sth*. There was a *B1.agree\_design* just previously.

(1) *B1.explain\_finding*, 8 seconds

*"That's then total <\*starts editing\*> independent of the (..) implementation. As long as they're independent (.), or not?"*

**Start think aloud\_activity, start write\_sth.** B1 explains why the design proposal is good. The "or not?" is rhetorical.

(2) *B2.agree\_finding*, 1 second

*"Hm."*

B2 agrees immediately.

(3) *B1.agree\_design*, 1 second

*"We can try."*

This refers to the original proposal. B1 does not stop editing.

(4) *B1.explain\_finding*, 3 seconds

*"<\*stops editing\*> Er, what are we (!...!) Have we even (!...!) Ah, no, our script is running (!...!). Right. OK."*

After a very short pause, B1 thinks aloud and arrives at an insight.

(5) *B1.amend\_design*, 2 seconds

*"And one needs to be id\_to\_code now."*

After a short pause, B1 refines the proposal: One of the two *code\_to\_id* calls must become *id\_to\_code* instead.

(6) *B1.explain\_finding*, 13 seconds

*"We have a code: id\_to\_code (!...!) <\*continues editing\*>"*

**think aloud\_activity ends** after the utterance. B1 makes further changes to the same line. He then switches into the browser to test the change; this **ends the write\_sth**.

## 19.2 *search\_sth*

*search\_sth* represents the search for well-defined items in digital (HCI) or physical (HEI) documents, by manual or automated mechanisms, in an efficient or inefficient manner, to either locate something or to check its existence.

We call an item well-defined if the actor appears to believe to know the item well enough to be able to specify it explicitly by an identifier, a file name plus line number (typically from an error message), a regular expression, etc. Automatic search includes the time of waiting for its results.

Example 19.2: Episode from Session BA1 showing a two-part *search\_sth*.

(1) B1.*propose\_step*, 2 seconds

*“Let’s look where that’s used in here.”*

B1 proposes to search for calls to a method; he points to a method name (*show\_sth*), then starts the search using the Eclipse search function (**starts B1.search\_sth/B2.search\_sth**).

(2) B1.*explain\_finding*, 4 seconds

*“Workspace (!...!) (;) Workspace is (~)”*

**Start B1.think aloud\_activity.** The Eclipse search dialog is open, the search scope is to be selected as either Workspace (default, currently selected) or Working Set. B1 starts the search as is. The search reports “Scanning file 1 of 2348”.

(3) B1.*say\_off topic*, 1 second

*“Muffins afterwards!”*

Unrelated remark after 2 seconds.

(4) B2.*say\_off topic*, 3 seconds

*“First work, then reward.”*

B2 replies laughingly. The search is still running.

(5) B2.*explain\_finding*, 2 seconds

*“We have 2350 files? Wow.”*

After two seconds, B2 takes notice of the “Scanning [...]” display.

(6) B1.*explain\_finding*+B1.*amend\_step*, 6 seconds

*“Eeeer, yes. That’s total nonsense of course. We could search in our Working Set. I thought I hadn’t checked out anything else, but (!...!)”*


After another second, B1 recognizes that the search scope was ill-chosen (finding type TC). While speaking, he stops the search. The utterance **ends the B1.think aloud\_activity**. He starts the search again, this time using Working Set scope. The pair waits for its results quietly. The results appear, **ending the search\_sth**.




### 19.3 *explore\_sth*

Exploring something means to investigate or probe a set of digital or physical data with respect to items that are not well-defined (contrast with *search\_sth*). The goal is either to determine whether such an item exists at all (such as a method providing a certain functionality), or to locate the (or such an) item, or to understand a set of items better (such as looking for ideas what to do next or how).

Exploring may look through the content of one file, the content of several files, the names of files, information provided by the IDE (such as lists of method names), the results of program runs (including compilation error messages, but see *verify\_sth* in Section 19.4), the content of a database (including IDE settings and so on), etc. Such activity may or may not have physical symptoms such as scrolling.

For instance in Session ZB7, the current driver Z20 suggests to look “*what methods the Subscriber offers*” and immediately starts scrolling through the TopicSubscribe JavaDoc webpage and then the MessageConsumer JavaDoc webpage (*explore\_sth*); he verbalizes the names of several methods of potential interest. 

The exploration may be superficial or thorough. Quiet reading (and even pure thinking) of material counts as exploration and reading aloud to oneself counts as exploration as well. Several exploration goals covered in one contiguous exploration procedure count as a single *explore\_sth*.

In contrast, reading aloud to the partner is *read\_sth* (Section 19.5). Checking the outcome of previous work is *verify\_sth*; *explore\_sth* does not apply to items created during the same session (Section 19.4) if these items are part of artifacts (as opposed to execution results). Discriminating *explore\_sth* from a manual *search\_sth* can be difficult. Even recognizing an *explore\_sth* at all may be difficult because no physical activity may be obvious; only context information can help. 

Example 19.3: Episode from Session BA1 beginning 13:42:11 and showing an *explore\_sth*. B1 had previously issued a `wget http://localhost` command line call (*do\_sth*); the result analysis has sufficient structure to turn it into *explore\_sth*.

(1) B1.*explain\_finding*+B1.*ask\_knowledge*, 1 second

“Hey. Hm. What’s cooking there?”

The console displays “Can’t resolve hostname ‘localhost’. Connection refused”. Starts B1.*explore\_sth* and B1.*think\_aloud\_activity*.

(2) B2.*propose\_hypothesis*, 1 second

“Firewall?”

**Starts B2.explore\_sth.** **B2.think aloud\_activity:** Apparently B2 is exploring, too.

(3) B1.explain\_finding, 1 second

“Can’t resolve hostname ‘localhost’.”

B1 considers this the interesting part. **Ends B1.think aloud\_activity.**

(4) B2.propose\_hypothesis, 2 seconds

“Maybe only the (~~)?”

**B2.think aloud\_activity:** A new idea (understandable for the partner) after 2 seconds of silence.

(5) B1.agree\_hypothesis

“(Yes.)”

**Start B1.think aloud\_activity:** Agreement after a pause.

(6) B1.explain\_finding, 1 second

“(~) is OK.”

The utterance **ends B1.think aloud\_activity and also B1.explore\_sth/B2.explore\_sth.** B1 proceeds with a new proposal and switches from the console into the browser.

*explore\_sth* represents exploration that stands for itself. If the activity occurs within or at the end of another activity and can be considered part of that other activity, only a concept for that other activity should be annotated. For example, a *verify\_sth* may consist of running a test and then examining the test’s output. The latter, by itself, would qualify as *explore\_sth*, but since from the point of view of the *verify\_sth* it is an integral part of the verification process, no *explore\_sth* should be annotated.


## 19.4 *verify\_sth*


*verify\_sth* represents activities for verifying or validating results of the pair’s previous work done in the same session. The checking can be manual or automated and can be review-based, analysis-based, testing-based, or mixed-mode. It includes the case of making sure that all process steps have been performed (as opposed to checking the product). It does not matter whether the actor expects to find correctness or rather incorrectness.

A *verify\_sth* may be announced by a *propose\_step* such as “Let’s look”. If it is not and also has no *think aloud\_activity*, it may be hard to detect. It can contain

actions such as reading and paging through code, program outputs, log files, data displays, and other representations as well as starting compile, build, deploy, or program run processes, operating programs, and many others.

*verify\_sth* often starts with a preparatory step, such as starting the program to be tested or opening a file to be reviewed. Implementing automated tests is not *verify\_sth*. The proofreading usually done as part of *write\_sth* is *write\_sth*, not *verify\_sth* (but semantic checking after a *write\_sth* is *verify\_sth*). Corrections (including commenting) performed during a *verify\_sth* terminate that activity and are *write\_sth*.

If the verification of the pair's work-results directly leads to the investigation of other material, it may be unclear whether this should be considered to end the *verify\_sth* and start an *explore\_sth* or rather be considered to continue a complex case of *verify\_sth*. 

If a test leads into a code review, it is useful to annotate two separate *verify\_sth*. If the pair does not verbalize the result of the *verify\_sth*, it can be difficult to decide when it ends. Like for *explore\_sth*, identifying when the partner is or is not taking part in the *verify\_sth* can be difficult and is largely an open problem. Subsequent studies need to decide whether to modify some of these criteria. 

Example 19.4: Episode from Session CA2 (12:25:39–12:26:23) in which C2 performs two HCI/HEI activities at the same time: a *do\_sth* during a *verify\_sth*. The pair had previously modified some code.

(1) C2.*propose\_step*, 1 second

"OK, let's go."

C2 proposes to test the application and clicks 'Run' to start it. **Starts C2.verify\_sth**. Is a C2.*think aloud\_activity*.

(2) C2.*ask\_knowledge* (or perhaps C2.*explain\_finding*), 2 seconds

"Huh, (.) our <\*>application name\*> (.) why can't we get at it?"

C2 moves the mouse to access the invisible Windows taskbar in order to see whether an instance of the program is already running, but the taskbar does not appear: C2.*do\_sth*. C2.*think aloud\_activity* (a separate one, because it refers to a new activity entity).

(3) C5.*explain\_finding*, 3 seconds

"It needs to do a rebuild – it is rebuilding right now"

While he speaks, the window of the new run appears at center-screen.

(4) C2.*disagree\_finding*, seconds

*"That has nothing to do with the taskbar not working."*

C2 disagrees and opens the Eclipse Console view.

(5) C5.*ask\_knowledge*, 1 second

*"What taskbar?"*

C5 had not previously understood what was C2's problem.

(6) C2.*mumble\_sth*, 1 second

*"(~Hm. Oh.)"*

Meaning unclear.

(7) C5.*ask\_knowledge*, 1 second

*"(~What ta) (!!...!!)"*

C5 asks again but gets interrupted.

(8) C2.*mumble\_sth*, 1 second

*"(~)"*

Incomprehensible interjection.

(9) C5.*ask\_knowledge*, 1 second

*"Is, is down?"*

C5 asks whether the taskbar is (routinely) hidden.

(10) C2.*explain\_knowledge*, 2 seconds

*"No, I don't have one. (.) Just look."*

C2 explains he considers the taskbar gone, not just hidden.

(11) C5.*explain\_knowledge*, 5 seconds

*"But it appeared for me a moment ago."*

C5 grabs the mouse that C2 is holding (C5.*become\_driver*, **starts** C5.*do\_sth*, C5.*think aloud\_activity*). He moves it down to the edge of the screen, but no taskbar appears. (**end of** C5.*do\_sth*).

(12) C2.*explain\_finding*, 2 seconds

*"We have an <application name> still running."*

It is unclear how C2 knows. Meanwhile, the startup of the new run finishes, is ready for testing.

(13) 6 seconds

–

C5 starts testing the newly-run application: **Starts C5.verify\_sth.**

(14) C5.explain\_finding, 1 second

“OK”

C5.think aloud\_activity: C5 finds the application to work correctly. **End of the C5.verify\_sth started in entry (13).**

(15) C2.agree\_finding, 1 second

“Fine.”

C2.think aloud\_activity: C2 agrees. **End of the C2.verify\_sth started in entry (1).**

(16) C5.propose\_step, 1 second

“Now we can check in.”


This suggestion how to proceed makes sure the verify is over.

## 19.5 *read\_sth*

By *read\_sth* one should annotate those (parts of) activity entities where the actor intentionally reads aloud (and verbatim) information shown on the screen, in a printed document, etc. – whether to communicate the information or only to direct attention (the partner’s or one’s own) to it. In contrast, casual reading (e.g. during a *verify\_sth*) that has more the character of speaking to oneself is not *read\_sth*. In HHI terms, *read\_sth* will often be *explain\_finding* of type D.

## 19.6 *sketch\_sth*

The creation of sketches, whether on paper, on a whiteboard, on a computer, or elsewhere is annotated as *sketch\_sth*. This is typically performed to clarify thoughts, whether the actor’s own, the partner’s, or both, and whether the clarification is directed more to the partner, the actor, or both.

Sketching is a potentially very interesting part of pair programming and subsequent studies interested in it should probably discriminate the above-mentioned cases. 

## 19.7 *show\_sth*

Pointing to certain information in a physical or non-physical artifact is annotated as *show\_sth* if it is the primary activity. The pointing can happen by

any means such as the mouse cursor, highlighting via a mouse-generated or keyboard-generated text selection, a pen, a finger, and conceivably also a nod (if specific), etc.

For instance, a developer explaining a stretch of code may point to identifiers or control constructs as she goes. Again, it is the underlying goal that defines the activity, so multiple pointings go together into a single *show\_sth* if they contribute to the same goal. Pointing may also serve to provide emphasis as in Example 10.1 (2).



In contrast, if the pointing is only part of a different primary activity, such as *explore\_sth* or *verify\_sth*, where the reader may for instance follow the current focus with the cursor, no separate *show\_sth* should be annotated to keep the annotation economical.

## 19.8 *do\_sth*

Anything that is not addressed by any of the concepts introduced previously can be annotated as *do\_sth*. In principle, it could be used for almost any physical movement at all or at least for minor computer actions such as clicking a button – but obviously that would as such hardly reflect a goal as base layer activities should and so should probably not be annotated separately.



In contrast, a good example of *do\_sth* could be the cleaning-up of the IDE workspace by closing multiple editor windows. Depending on the goal of your study, there might be many such meaningful actions that fall under *do\_sth* in the base layer – but a *do\_sth* annotation provides too little meaning.

So please consider *do\_sth* to be a kind of beacon: Whenever you feel tempted to use it, also consider whether you should leave out that annotation as most likely unimportant or invent a more specific concept to represent the phenomenon more meaningfully. Using *do\_sth* is advisable only if neither is the case.

## 19.9 On drivers, observers, and co-action

In order to understand pair programming by qualitative analysis, it is obviously important to record who is the actor of each event being conceptualized. For a speech act represented by an HHI concept instance, this is trivial. In contrast, while the HCI/HEI concepts presented above are all based on an observable (at least in principle) physical act as well, it is the underlying goal that really counts and it may be far from obvious whether P1, P2, or both are pursuing it.

HEI and even HCI activities can be performed not only by the person currently in possession of the keyboard (the “driver”) but also by the other (the


“observer”)<sup>1</sup> In order to understand the phenomena in the session, we should ask ourselves who is the actor (or actors) for each HCI/HEI annotation and assume that co-action, both partners sharing the activity intellectually, may be common.

This is not easy even for the physical act itself: The observable physical act is clear and obvious for some concepts (such as for *write\_sth*, *sketch\_sth*, and *show\_sth*, unless hidden by limitations of the session recording) but often subtle for others (in particular *verify\_sth*). And even a partner not participating in a physical computer-operating act may well co-perform the corresponding intellectual act.

The base layer suggests the following criteria as a starting point for assigning HCI/HEI actorship: Unless there is specific evidence against it . . .

- . . . consider the main physically active pair member as an actor. Where operating the computer is involved, this will be the driver, but where fingers, pencils, or eyes suffice as the active infrastructure, it can also be the observer.
- . . . consider apparently concentrated gazing at the monitor (if it can be diagnosed) to be an important indicator (in favor or against actorship, depending on the situation).
- . . . consider each person who proposed the activity or participated in its discussion as an actor.
- . . . consider each person verbalizing part of the activity as an actor.
- . . . consider each person asking or answering questions pertaining to the activity during the activity as an actor.

Not only the participation itself in some activity may be hard to diagnose, its exact beginning and end may be as well (see also Section 21.5).

Overall, the base layer cannot provide sufficient criteria for reliable assignment of HCI/HEI activities to the right actors. However, approximate correctness of the annotation may be sufficient for many purposes. 

---

<sup>1</sup>We have even seen cases where the “observer” obtained full control, telling the “driver” what to do on a keystroke-by-keystroke basis (“*Down, Down, Down, Return*”).





## Supplementary concepts


The base layer provides a few very rough additional concepts that may be useful on some occasions.

### 20.1 *become\_driver*

The actor takes control of the keyboard and mouse. Cases such as a pair member taking the mouse only are not covered.

### 20.2 *work in parallel\_sth*

Both partners are independently working on different activities. This simplistic concept can either be used without an actor (*work in parallel\_sth*) or with pairs of annotations ( $P1.work\ in\ parallel\_sth + P2.work\ in\ parallel\_sth$ ), allowing more granular use.

For studies interested in such behavior, in particular when analyzing distributed pair programming based on tools such as **Saros**<sup>1</sup> that allow the pair to also each use their computer independently, much more refined ways of describing the degree of coupling of the two activity strands will be needed. 

### 20.3 *work alone\_sth*

The actor continues work without any partner at all because the partner left the workplace.

---

<sup>1</sup>[http://en.wikipedia.org/w/index.php?title=Saros\\_\(software\)&oldid=566544288](http://en.wikipedia.org/w/index.php?title=Saros_(software)&oldid=566544288)

## 20.4 *wait\_for\_sth*

The “actor” suspends all activity until a certain expected event occurs, such as the results arriving after a long-running search (Section 19.2) or test. Typically used together with concepts such as *search\_sth* or *verify\_sth*.

## 20.5 *react\_to\_interrupt*

Marks a point where the pair stops its work due to an external event such as a phone call coming in. Such an annotation explains why no further annotation follows for some time afterwards.

## Part IV

# Using the base concepts

...in which we summarize how to use the base concepts as they are (Chapter 21), how to adjust them to your particular study (Chapter 22), and how to introduce your own layer(s) of concepts in addition (Chapter 23).




## Guidelines for annotating

If you have read the concept descriptions in Chapters 4 to 17 attentively, you have seen not only many concept-selection rules, but also a number of hints on how to operationalize the annotation process. Both are fragmented over many pages, so we summarize them in this chapter.

### 21.1 How to pick appropriate HHI concepts

The many details of each concept as presented in the previous chapters notwithstanding, here are the *principles* of annotating appropriate HHI concepts to utterances:

- Your goal is to determine the primary illocution of an utterance. The length of utterances is determined by what still belongs to the same illocution. 
- Describe the utterance's possible illocution in your own words. Make sure you notice if the speech act is indirect so you can clearly tell apart the secondary (obvious) illocution from the primary (actual) illocution and use the primary one. There are many more different illocutions than there are different verbs in the base concepts so you will usually need to make a subsumption step to determine which base concept (combination of a verb and an object) fits best with the overall content of the utterance. Best fit will often depend on specifics of the current pair programming session context. There are many examples throughout the book in which this problem is discussed; start for instance in Section 4.2.1.
- If it is an initiative utterance, annotate the most specific base concept that applies. (Roughly speaking, specificity decreases from chapter to chapter. See also Section 11.3.)

- If it is a reactive utterance, observe the episodes principle (Section 2.3.6).
- If no base concept appears to apply properly, consult Chapter 22.
- If more than one base concept applies, consult Section 21.4.

While using this procedure, be aware of two inherent limitations: First, in the best possible world, each concept would be defined by stating all of its necessary and sufficient conditions. This goes back as far as Aristotle and while it works well in mathematics, it **does not apply well**<sup>1</sup> in the world of human language and action. Thus, the manner in which we define the base concepts is more akin to the idea of Wittgenstein's **Familienähnlichkeit**<sup>2</sup> (or other kinds of **prototype theory**<sup>3</sup>) combined with some amount of **ostensive definition**<sup>4</sup>.



As a result, the base concepts on the one hand may not precisely cover a particular given case exactly, but on the other hand are open for modification to repair that problem. Mindless application of fixed rules is definitely *not* a manner of working with the base concepts that is likely to create useful results.

Second, not only the base concepts are limited, the observer is limited as well: There is no way you (or anyone, including the pair members themselves) could guarantee to have determined the “right” illocution for a given utterance.

From our pragmatist point of view, both of these restrictions are not overly disturbing, because perfection is not required, only usefulness is.

## 21.2 How to pick appropriate HCI/HEI concepts

The criterion for determining an appropriate HHI concept for an utterance is the utterance's illocutionary act. We use an analogous criterion for the HCI/HEI concepts: What does the activity do (in terms of its intention, goal, purpose) in the context of the session?



For instance, scrolling down through a file might be the execution of a *verify\_sth*, a manual mode of *search\_sth*, some part of an *explore\_sth*, and perhaps something else entirely (*do\_sth*). In particular, it might be a **displacement activity**<sup>5</sup>. As mentioned in Section 17.2 (see the possible example there), displacement activities are not modeled in the base layer but one may want to model them in subsequent studies.

<sup>1</sup>[http://en.wikipedia.org/w/index.php?title=Concept&oldid=567169993#Notable\\_theories\\_on\\_the\\_structure\\_of\\_concepts](http://en.wikipedia.org/w/index.php?title=Concept&oldid=567169993#Notable_theories_on_the_structure_of_concepts)

<sup>2</sup>[http://en.wikipedia.org/w/index.php?title=Family\\_resemblance&oldid=565012381](http://en.wikipedia.org/w/index.php?title=Family_resemblance&oldid=565012381)

<sup>3</sup>[http://en.wikipedia.org/w/index.php?title=Prototype\\_theory&oldid=540675817](http://en.wikipedia.org/w/index.php?title=Prototype_theory&oldid=540675817)

<sup>4</sup>[http://en.wikipedia.org/w/index.php?title=Ostensive\\_definition&oldid=544692051](http://en.wikipedia.org/w/index.php?title=Ostensive_definition&oldid=544692051)

<sup>5</sup>[http://en.wikipedia.org/w/index.php?title=Displacement\\_activity&oldid=558153583](http://en.wikipedia.org/w/index.php?title=Displacement_activity&oldid=558153583)

### 21.3 What to consider as context

Many times we have used the term *context* when discussing that an utterance can usually not be interpreted based on its wording and pronunciation alone and that further information needs to be used as well but we have never explained what context really means.

From a practical point of view, the applicable context is restricted to what is known to the researcher to a sufficient degree. In our setting, a relevant restriction will often be imposed by the scope of the current pair programming session. The following items may or may not be available to the researcher as part of the applicable context:

- The goal or task set for the session.<sup>6</sup>
- Everything said in the session so far (HHI).
- Everything done in the session so far (HCI, HEI).
- All information shown on the screen (and noticed by a pair member!) now or previously in the session.
- The personality, physical state, and mood of the pair members, their general and their short-term social relationship, etc.<sup>7</sup>

While the second and fourth of these are often reasonably well observable, the observability of the others tends to be severely restricted. The researcher's understanding of the actual context is thus limited.

### 21.4 When to use double HHI annotations

In order to keep the annotation process manageable, the base layer attempts to normally assign only a single HHI concept to each utterance. This principle is not always appropriate. You may occasionally need two (or possibly more) annotations to best describe the current phenomenon. Here is the list of such cases as known to us<sup>8</sup>:

1. Proposals (or agreements, disagreements, challenges) that are “packaged into” (that is, take the form of) *explain\_knowledge* or *explain\_finding* utterances. These are indirect speech acts in which both the secondary (*explain*) and the primary illocution are of major interest. In such cases, both concepts should be

---

<sup>6</sup>Such information can and should be gathered by other means outside the actual session recordings, such as an interview or questionnaire.

<sup>7</sup>Ditto.

<sup>8</sup>You may occasionally encounter additional ones – but if you think you do, re-read the annotation principles in order not to open the floodgates prematurely.

annotated; see Section 6.3.6, Example 16.2, Example 17.2. There are also cases in which it is ambiguous whether the *explain* is primary or secondary; they are handled in the same way. Also, instead of *explain* you may see *challenge* as in the *propose\_step/challenge\_finding* pair of Example 12.6.

2. Utterances containing subutterances<sup>9</sup> that need to be annotated differently but where splitting the utterance up is not possible or not wanted; see Section 10.2.3. For the common case of *finding* utterances that explain some existing knowledge as well, you will need to decide how to handle them in subsequent studies: double annotation for completeness or only the *finding* for simplicity.
3. Utterances that are ambiguous with respect to the base concepts. It does not matter whether the ambiguity appears to exist in the speaker's head as well (as in Section 9.3.5 and Example 16.6 (2)) or appears intended by the speaker (as in Section 6.3.5).
4. Indirect speech acts where the secondary illocution is of a type that is in the focus of your particular studies; see Section 9.3.5.
5. Design proposals made as part of strategy proposals.
6. Implicit action announcements as discussed in Section 21.6.1.
7. Any *activity* utterance that is not *only* that.

Any HCI/HEI annotation that happens to coincide with an HHI annotation is not a double annotation at all in this sense.

- Note that this list concerns only the base layer as such. Your own subsequent studies will likely deserve their own rules for the use of double annotations that are aligned with the research question and may have to modify the above list considerably.

## 21.5 How to segment utterances

For many types of qualitative study, the segmentation of the raw data is a crucial and difficult step and consumes a large fraction of the overall effort – yet we have not talked about this at all so far. Why not?

Because in the base layer's approach, asking for "how to" segment is not a question that has an explicit answer. It has no explicit answer because the choice of concept to use for an annotation and the choice of segment to which to assign that concept are determined at the same time and influence each other.

The following is the best approximation to "segmentation rules" for the base layer:

---

<sup>9</sup>Also, utterances the meaning of which contains such subutterances, i.e., that can be *paraphrased* such that the subutterances emerge.




- Segmentation is a holistic process concurrent with assigning the concepts.
- If you have decided to annotate the HHI concept X and your candidate segment covers a whole utterance, covers the respective X completely, but does not cover more than the X, then you have chosen a suitable segmentation.
- If your segment covers less than a whole utterance and the remaining part needs to be annotated with a different HHI concept, then you have also chosen a suitable segmentation.
- If your segment covers less than a whole utterance and the remaining part needs to be annotated with the same HHI concept, extend your utterance unless the remaining part is clearly a semantically separate and different instance of X.

Apply the same idea analogously to the HCI/HEI concepts as well (“utterance” becomes “activity”). During your annotation process, you should perform constant comparison and always consider not only the annotations themselves as preliminary, but the segments to be annotated as well – in particular (but not only) for HCI/HEI concepts.

## 21.6 How to handle specific phenomena

### 21.6.1 How to annotate implicit announcements

A question or other utterance may sometimes reveal that the speaker intends to do something although the intended action is not mentioned; see Example 16.2 (1). We call such a disclosure, whether conscious or unconscious, an *implicit action announcement*. Such announcements are not easy to detect, because in comparison to explicit ones they require the researcher to have a much better understanding of the session’s technical meaning. However, as the partner may detect that intention and this may influence the session’s course substantially, you should annotate the implicit *propose* whenever you detect one.

Depending on their research question, subsequent studies may have to define rules for limiting the effort put into detecting such implicit announcements in a well-defined way. 

### 21.6.2 How to annotate thematic shifts

One of the primary intentions during the creation of the base layer was making visible the structure of basic dialog episodes; the approach chosen for doing so was by marking intra-dialog relationships (Section 2.3.6). Via the notion

of initiative versus reactive verbs (Section 3.5), this basic approach led to the structure of concept classes as explained in the chapters above. Each such concept class essentially represents a topic type. An episode in the primitive sense of the base layer is basically a sequence of utterances regarding one particular instance of one such topic.

When using the base layer, it is therefore important to correctly identify the topic (concept class) at each point. The base layer provides help for doing this in various forms, starting with the basic definition of what holds together a concept class (such as “What is a design proposal?”, Section 4.1), over sometimes a few helper concepts that distinguish important subtypes (such as the various strategy types (Section 9.1) or hypothesis types (Section 13.1)), down to subtle additional rules for individual concepts such as the idea that an *explain\_knowledge* must not express any doubt (that would make it a *propose\_hypothesis*) but an *agree\_knowledge* may.

Nevertheless sticking to one concept class within an episode is not always the right thing to do: Sometimes, a reply does not react to the primary topic set by the previous utterance but on something else such as a subaspect of it. Such cases may constitute a change of topic which we call *thematic shift* (or simply *shift* for short). For instance in Example 21.1, B2’s reply neither pertains to B1’s design proposal nor to the hypothesis stated along with it. Rather, it only pertains to a finding that is implicit in the proposal:

Example 21.1: Episode from Session BA1 (14:57:00–14:57:19) in which an annotation is used only due to a thematic shift. For description and discussion of the content you will need to consult the whole of Section 21.6.2.

(1) B1.amend\_design/B1.explain\_finding+B1.propose\_hypothesis

“But then why don’t we also, er (.) kick out the one for this id <\*points to element\*>? (.) Or do they do that explicitly somewhere else again?”

“They” are the previous programmers of the code.

(2) B2.agree\_finding

“That’s a good point. That’s missing here.”

B2 says this after the pair has sat silent for 10 seconds.

In this episode, the pair is discussing a modification of a `for` loop that removes objects from a cache and that (like most of the surrounding code) was written by other programmers. B2 had previously suggested a particular deletion (*propose\_design*) and B1 now amends that by another. B1’s utterance has three aspects that can be paraphrased as follows:


- “This element needs to be deleted too.” (*explain\_finding*)
- “I suggest to do it right here.” (*propose\_design*)
- “But perhaps it’s already being done elsewhere.” (*propose\_hypothesis*)

The design proposal is the main aspect and would hence normally suppress the annotation of the finding (Section 4.2.1), because the finding is not stated explicitly. B2’s reply ignores the design proposal, ignores the hypothesis, and only refers to the implicit finding. B2’s utterance clearly should be annotated as *agree\_finding*. What the base layer suggests to do in addition is adding another code to the *previous* utterance that “prepares the ground” for the concept used for the reply: We annotate the finding aspect (that would normally have been merely implicit) in addition to the already-present design and hypothesis aspects.

This is a general rule and the whole point of this short chapter: If a thematic shift from class A to class B occurs in a reply, then

- annotate the reply by a B concept even if that breaks the normal episode structure, but
- also consider adding a suitable corresponding B concept to the *previous* utterance (the one to which the reply refers).


The addition should be made if the B concept is appropriate according to its own definition and had only been suppressed due to the definition of the A concept used and/or the A-versus-B discrimination rules. We call such annotations *shift-motivated annotations*. The addition should not be made if by itself the B concept is inappropriate for the previous utterance.

Later studies may additionally or alternatively want to introduce an explicit *shift* concept or property if that better suits their respective research goal. 

### 21.6.3 How to annotate repetitions

If the semantically (but not necessarily syntactically) same thing is said again by the same speaker, there are four cases:

An *immediate repetition* occurs within the same utterance. This is not annotated separately but is simply considered to be either emphasis or meaningless and lumped together with the first.


An *early repetition* occurs when the session context has not yet changed substantially. Its dialog role will usually be to make sure the original utterance is not lost or ignored. Such repetitions will be annotated with the same concept again. 


The fact of being a repetition is not modeled by the base layer; subsequent studies may want to introduce concepts for doing this.

An *early repetition* of an *explain\_finding* can be a special case: If the dialog topic has not progressed since the first time the finding was stated, the above early repetition rule applies; we just annotate *explain\_finding* again. If however, the dialog has touched a different topic in the meantime (such as the partner commenting on something else than the finding), the same content will then be annotated as *explain\_knowledge*, because the insight is now considered to have become part of existing knowledge. Do not apply this rule if you find clear symptoms that the speaker had forgotten his or her earlier insight (and not just its relevance!).

A *later repetition* is annotated as is most appropriate for its new context.


#### 21.6.4 How to annotate incomplete agreement or disagreement

 The agreement (or disagreement) with proposals and explanations is often less than one hundred percent. So far, we have seen two cases: Either the speaker is unsure of his or her own judgment (as in Section 4.2.10), or he or she agrees with only a part of the previous utterance and disagrees with some other (which may be common for strategies, as in Section 9.2.9). If a *challenge* annotation is not appropriate, the base layer annotation should then simply be an *agree* or a *disagree* – whichever sentiment appears to dominate.

 Subsequent studies, however, may need to introduce concepts for also annotating the doubt and/or partial criticism.

#### 21.6.5 How to annotate self-corrections


Utterances that extend (*amend*) or contradict (*disagree*, *challenge*) a previous utterance need not come from the partner; they may also come from the same speaker as in Example 4.2 (*amend\_design*), Example 6.3 (*amend\_step*), or as discussed in Section 12.2.4 (*challenge\_finding*). This is not unusual and may be important for the course of the session.

 The base layer does not express the fact of self-correction in any way, it is only (ambiguously) visible from the overall episode structure. Subsequent studies may wish to introduce concepts for annotating self-corrections as such explicitly.


#### 21.6.6 How to annotate justifications

One particular type of *explain\_knowledge* utterance is the justification of a proposal. From its frequency and importance alone, *justification* could have been a separate concept class. However, to justify something is a form of illocution

and in the base concept set illocutions are expressed by the verbs, not the objects.

Thus, there is no indication (in an annotation that uses only the base layer) of justifications and suitable concepts (probably just attributes) for modeling them may be one of the first additions that subsequent studies interested in these phenomena should make. 

## 21.7 Method hints

The guidelines in this book so far may be sufficient to define what the result of your annotations *should* look like (unless you chose to modify them as discussed in Chapter 22). However, as you may have guessed already from reading the examples, the session reality you will face can be very peculiar, leaving you perplexed. In such moments, the following tricks may help. They are not meant to be used in any particular order; rather, they are part of the annotator's toolkit much like the definitions of the concepts themselves. 

### 21.7.1 Step back

If there were a mantra of the base layer annotator, it would be “*Context, context, context!*”. So if you are having a hard time deciding which concept is the right one for the next utterance, it may be that (a) as a beginner you are not taking the context into account enough and are rather attempting to categorize an utterance based on only its language content, or (b) later on you are looking at context but cannot see the forest for the trees; you are considering the previous utterances and their annotations but still cannot make up your mind what the current one means.

In the latter case, step back and put on your software developer hat (as in Example 6.2): What is going on there in terms of the overall programming session? What is the current goal? What is understood and what is unclear? To whom?

### 21.7.2 Paraphrase

If that does not help and you are still torn between two concepts, it may help to use your context understanding for re-expressing the utterance in new wording.

For instance in Session BA1, B2 asks

*“Should we store both values or should (!...!). (..) Or is last\_change sufficient? Do we really need last\_request?”*



A first-time user of the base layer might immediately annotate this as *ask\_knowledge*, but let us assume you have meanwhile learned to mistrust the syntactical form and ask yourself whether maybe this ought to be viewed as *propose\_design*? How would you make the decision between the two? You use each of these possibilities and rephrase the essence the utterance would then have.

If it was indeed *propose\_design*, it ought to mean “*I suggest to use last\_change only and not use last\_request*”. In this case, this does not sound right, because not handing over *last\_request* to the script had been suggested previously, so the original utterance above would be a rather strange way of suggesting something closely related and is thus probably not intended to be a proposal.


Counter-check: If the utterance was indeed *ask\_knowledge*, it ought to mean “*I do not understand why we should store last\_change as well as last\_request*”. This meaning fits nicely into the context so it is likely the right interpretation and hence *ask\_knowledge* should be annotated. See also Example 12.2 (3).

### 21.7.3 Peek into the future

If paraphrasing also does not help, it is time to use your ultimate weapon: Your super-human power of seeing the future. An often very helpful indicator towards an appropriate annotation for an utterance is the reaction of the partner: If the partner apparently interpreted the utterance as X, maybe you should do so too rather than annotating Y? If the speaker subsequently protests, this may obviously be the wrong idea, so you should do a little further peeking to safeguard against that possibility. See the discussion in Section 9.3.7. The technique will be particularly valuable when you need to understand a misunderstanding where the respective utterance may appear totally illogical at first; see again Example 19.4.

Beware, however: The partner’s reaction may actually be a misunderstanding and the speaker simply too lazy, too tired, too shy, too slow, too inattentive, or any of a dozen other things to protest. In that case, the annotation resulting from peeking into the future would be wrong. It may be a tiny little bit wrong only, because the speaker may not have been fully clear of his or her own intentions to begin with. It may be wrong but without consequence, because that particular annotation difference does not make a difference with respect to your research question. But if your research question has anything to do with the precision or reliability of communication, with contingency or serendipity, or with shifts (Section 21.6.2) you rather want to detect and annotate ambiguity than making this kind of annotation mistake. So like all superheroes you should use your power with great care.

## Guidelines for modifying the base concept set

Whatever your research question is for a subsequent study that uses the base layer, the base concepts alone are not likely to get you an answer. You will usually require additional and/or different concepts to cover all your annotation needs in order to produce your insights. Remember that the base concepts should not be viewed as a coding scheme but rather as a starting point for your own understanding and modeling; see Section 2.1. 

For instance, the clean differentiation between *finding* and *knowledge* (here and in Chapter 16) and between them and the other HHI concepts (Chapters 4 to 10) required an extremely lengthy research process. We are convinced that the resulting concept class structure is very useful for pursuing a broad variety of research questions. Nevertheless, it may be ill-suited for your particular question. Feel free to kick parts or all of it overboard as soon as you have *precisely* understood why you are doing it and have developed some idea of what you will gain. (Be advised that you will likely not have a good idea what you will lose.)

We cover modifications and additions within the topic realm of the base concept set in this chapter and outside of it in the next chapter.

### 22.1 What makes a good concept set

A good set of concepts for analyzing pair programming should fulfil at least the following two quality criteria:

- Adequacy: It should allow to express all phenomena of interest adequately (that is, without too much distortion), with sufficient precision,

and in a matter that explains (“theoretical coding”, see Section 1.4.4) rather than merely describes.

- Parsimony: It should sufficiently limit the intellectual effort required during the annotation process so that the process remains manageable.

It is not easy to create an adequate set of concepts anyway, but it becomes very difficult if you strive for parsimony at the same time. There must neither be too many concepts nor must the concepts be overly diverse and arbitrary.

The base concept set has nice properties in this respect: The overall number of concepts is modest, the concepts are nicely grouped into concept classes, and even the individual concept classes are similarly structured and constantly re-use almost the same criteria for discriminating the concepts. Furthermore, the concepts are fairly generic and can be applied uniformly across studies on a diverse set of topics, thus connecting those studies to one another. This character should be maintained when modifying and extending the base concept set.

## 22.2 When to shift a boundary



If you find that for a particular phenomenon the base layer does explain how it should be annotated but that annotation would blur a distinction that is important for your research question, feel free to change that particular rule and shift the boundary between certain base concepts.

If that need arises for a detail rule that only discriminates one pair of concepts (from somewhere in a “Discrimination from similar concepts” section) the modification is usually unproblematic. Just make sure you thoroughly document (including rationale) all such deviations from the normal definition of the base concepts so that the meaning of your annotations can be reproduced. We have already marked many plausible candidates for such alterations in the respective places.


If, however, the modification need arises for a more general rule that discriminates one verb from another for all concept classes or one object from another for all its verbs, be aware that the modification may shatter the integrity of the base concept set as a whole if you are not careful. It is a good idea to sleep over such a decision, discuss it with somebody else (perhaps with us?) and then not do it – resort to an extension instead; see Chapter 23 and the remainder of the present chapter.

## 22.3 When to add a new property value

It may happen that you encounter phenomena you are inclined to consider, for instance, a *strategy* but that do not fit into any of the existing strategy types as




defined in Section 9.1.



In this case, after carefully checking that your phenomenon does *indeed* not fit into any of the existing types, you should not hesitate to add a new type to the list. This amounts to refining the base layer by completing the description of a primary characterizing attribute with respect to a case that had never occurred in our data. 

It is conceivable (although we do not think it is likely) that you will discover another primary characterizing attribute. In this case, you would not only add property values but rather a complete property along with its values.


## 22.4 When to add a concept and which

If you arrive at the end of the instructions of Section 21.1 and have not found a base concept that characterizes the illocution you have determined, the first step is asking yourself whether one of the universal concepts (e.g. a *knowledge* concept) can do the job. These concepts are collection pools for many different phenomena and yours may be among them if you consider the illocution less specifically. If none of them fits, continue as explained in the next paragraph. Do the same if one of them does fit but such annotation would lose a discrimination that is important for your research question.

The second step is considering new combinations of existing verbs and existing objects. For instance you might encounter an utterance that is clearly a *disagree*, applied just as clearly to a *todo*. No concept *disagree\_todo* exists in the base concept set, so what are you to do? You need not get the least bit nervous in such cases. The only reason why *disagree\_todo* is missing from the base concept set is that we have never seen it in our sessions. If it occurs in yours, nothing speaks against adding it to your concept set right away. Just make sure that over time you understand how to discriminate it from its neighbors in ambiguous cases. For instance, if the speaker provides a justification, the resulting *disagree\_todo+explain\_knowledge* might sometimes be similar to an *explain\_state*. If you communicate your results, feel free to consider your addition to be part of the base concept set, but make sure you properly explain the new concept(s) to your readers. 

The third and final step is considering to add new verbs or new objects. For instance, a number of times during the analysis of our sessions we encountered utterances in which the speaker stated where a certain relevant information could be found (such as in certain other parts of the code, certain project documentation, or certain public documentation). For instance in Session ZB7, Z19 says “*There was this page where you could look up what to do in the, um, JNDI directory*”. We considered the possibility of adding a concept such as *remember\_source of information* to represent such utterances. This concept would reuse an existing (if rare) verb, but introduce a new object. The resulting  

concept would be highly specialized, but that is also true of, for instance, *explain\_gap in knowledge*. So why did we include *explain\_gap in knowledge* in the base concepts but did not, eventually, include *remember\_source of information*? For two reasons: First, *gap in knowledge* concerns the subtle but important distinction of knowledge and meta-knowledge, while *source of information* is just another type of fact knowledge. Second, we felt that moments of *explain\_gap in knowledge* utterances are interesting, even crucial moments in a pair programming session, whereas *remember\_source of information* is merely another form of tapping into resources – which pairs do all the time.

 This does not mean that in your particular studies you should never add something like *remember\_source of information*, but we think that entirely new concepts should rarely be considered part of the base concept set but rather be considered part of a new layer. Separate layers will be discussed in Chapter 23.

## Guidelines for creating new concept sets

### 23.1 The idea of layers

As discussed in Section 1.4.3 (it might help to re-read that one-page section now), the purpose of layers is modularizing the overall research process: Rather than inventing each concept yourself, you reuse one or several sets of concepts that are well-worn, refined, delineated, documented, minimized, and validated.

If all goes well, this has a number of advantages:

- Even though you may work with a large number of concepts, you can find each concept easily because they come in clearly arranged topic blocks.
- For the same reason, you are less likely to misinterpret and abuse a concept.
- You save a lot of concept development work, but grounding is still in place, because constant comparison applies to reused concepts just like it does for your own.
- When you write up your results, you can do so more concisely, because you may be able to refer to this book rather than explain everything yourself in some places.
- If some other researcher concurrently reuses the same concepts as you, it will be much easier to compare and combine the results afterwards.

## 23.2 Granularity

To introduce successful concept layers for your own research questions, it may be required to explain phenomena of a much larger or much smaller granularity than those described by base concepts.

In some cases your research question may require (figuratively speaking) to pull out a microscope to look at the local details of particular utterances. In other cases, your research will want to understand larger-scale structures that comprise several utterances and help understanding the overall development of a pair programming session. For instance, some of the new concepts may need to talk about whole episodes (in the sense of Section 2.3.6) or about sets of such episodes.

## 23.3 Properties and property values

An example for a sort of finer-grained concept that may be useful for your layer are properties. Properties (and property values) relate to main concepts (and their annotation instances) somewhat like attributes (and their values) relate to classes (and their instances): A property value annotation provides detail on a particular instance of a concept.

Property concepts are a great way to add almost any detail you need regarding the phenomena of an existing concept (whether a base concept or other). In particular, adding such detail to existing concepts can be a useful intermediate step when working towards identifying new concepts: Once you have enriched the existing concept with enough detail, you may suddenly discover an expressive new concept.

With respect to the base concepts, one use of this technique will apply when your research question requires differentiating a single base concept into several narrower concepts, in particular:

- Discriminating different “flavors” of an object type.
- Discriminating different “flavors” of a verb type.

For instance if you are interested in understanding how pair programmers develop the design of their software, you will need to record much more detail about the design proposals, such as their granularity, the context in which they arise, their focus (data, logic, structure, etc.), their newness relative to previous proposals of the same speaker, their newness relative to previous proposals of the partner, their newness relative to existing code, the level of abstraction used to express them, the degree of detail given, and so on and so forth.

Also for verbs, there is no reason why for instance there could not be different types or modes of challenging. The base layer description has already mentioned two such verb subtypes for explanatory purposes: the agreement types for *agree\_finding* (Section 12.2.9) and for *agree\_knowledge* (Section 16.2.7) and the proposal modes (Section 4.2.1) that apply to many concepts.

## 23.4 Forming “nice” layers

Obviously the advantages of layering will be damaged if the layers are too fine-grained, too coarse-grained, or their topic not well-defined. But how to obtain a right-grained and well-defined layer?

One could think about the problem of forming good layers in terms known from software modularity: A nice layer would then have an interface that hides detail (meaning not all of the concepts will be explained in the article you eventually write about your study), it will have high coherence (meaning it focuses on a particular topic area), and it will strive for low coupling to other layers (meaning it does not meddle with the concepts of other layers more than necessary, rather reusing them as they are when possible).

However, we do not know yet whether this metaphor is helpful. It may actually stand in the way of focussing on what is most important, namely obtaining insight with respect to your research question. “Nice” layers should probably be considered a by-product of your pair programming research, not a goal in themselves.

## 23.5 Go!

If you are one of those people who have to read the end of a book first even if it is a non-fiction book: Congratulations, you have now done so. Please proceed to Section 1.5.3, choose one of the two suggestions described there, and start reading in that manner.

If you have arrived here while following the suggestions of Section 1.5.3<sup>1</sup>: Great, you are now ready to start your base-layer-driven research. Just go ahead and complete your base layer knowledge as you go. Feel free to get in contact with us any time to discuss interesting phenomena you see or decisions you face. Maybe you also want to chat with us when defining your precise research question? We will be glad to!

---

<sup>1</sup>If you have arrived here by reading this book cover-to-cover: Wow! You appear to be really keen on studying pair programming.



# Bibliography

- [1] E. Arisholm, H. Gallis, T. Dybå, and D. I. Sjøberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.
- [2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.
- [3] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley Professional, 2004.
- [4] S. Bryant, P. Romero, and B. du Boulay. Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies*, 2008.
- [5] L. Cao and P. Xu. Activity patterns of pair programming. In *Proc. of the 38th Annual Hawaii International Conf. on System Sciences (HICSS 2005)*, page 88a, Washington, DC, USA, 2005. IEEE Computer Society.
- [6] J. Chong and T. Hurlbutt. The social dynamics of pair programming. In *ICSE07: Proceedings of the 29th Int'l Conf. on Software Engineering*, pages 354–363, Washington, DC, USA, 2007. IEEE Computer Society.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [8] J. Hannay, T. Dybå, E. Arisholm, and D. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, 2009.
- [9] J. E. Hannay, E. Arisholm, H. Engvik, and D. I. K. Sjøberg. Effects of personality on pair programming. *IEEE Transactions on Software Engineering*, 36(1):61–80, Jan. 2010.
- [10] J. T. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, 1998.
- [11] S. Salinger. *Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung*. PhD thesis, Freie Universität Berlin, Fachbereich Mathematik und Informatik, 2013.

- [12] S. Salinger, L. Plonka, and L. Prechelt. A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 4(1):9–25, 2008.
- [13] S. Salinger, F. Zieris, and L. Prechelt. Liberating pair programming research from the oppressive driver/observer regime. In *Proc. 35th Intl. Conf. on Software Engineering (ICSE)*, pages 1201–1204. IEEE Press, 2013.
- [14] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. SAGE, 1990.
- [15] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Professional, 2002.
- [16] S. Wray. How pair programming really works. *IEEE Software*, 27(1):50–55, 2010.



# Index

## Symbols

\*\_design, 66

\*\_step, 79, 130

## A

activity, 31, 32, 40

activity, 40, 41, 46, 56, 67, 111, 151, 165, 169, 200

agree\_, 169, 171, 172, 174

amend\_, 169, 171–174

challenge\_, 169, 171–174

disagree\_, 118, 169, 171, 172, 174

stop\_, 169, 171, 173, 174

think aloud\_, 56, 166, 167, 169–175, 183–189

activity entity, 181

advantage, 100

agree, 41, 47–49, 64, 65, 67, 106, 145, 158, 161, 173, 204

agree\_activity, 169, 171, 172, 174

agree\_completion, 84

agree\_design, 60, 61, 64, 65, 81, 123, 124, 156, 161, 183

agree\_finding, 115, 120, 121, 123–128, 144, 156, 157, 183, 189, 202, 203, 213

agree\_gap in knowledge, 147–149

agree\_hypothesis, 133–135, 171, 186

agree\_knowledge, 102, 103, 120, 142, 148, 153, 155, 156, 158, 163, 202, 213

agree\_requirement, 70, 71

agree\_standard of knowledge, 148, 149

agree\_state, 97, 106

agree\_step, 75–78, 127, 157

agree\_strategy, 88, 96, 97, 99, 102, 103

agree\_todo, 88

agreement type, 127, 128, 155, 157, 213

known, 127, 155

me-too, 128

unchecked, 155, 157

understood, 128, 155

all is data, 27

amend, 41, 47–49, 64, 78, 98, 123, 134, 148, 167, 173, 204

amend\_activity, 169, 171–174

amend\_design, 60–64, 80, 84, 89, 90, 101, 183, 202, 204

amend\_finding, 54, 115, 120, 122–125, 128, 129, 154, 157, 161

amend\_hypothesis, 133–136

amend\_knowledge, 154

amend\_requirement, 72

amend\_step, 63, 75–78, 82, 184, 204

amend\_strategy, 33, 63, 87, 88, 96–99

annotation, 25, 30, 31, 37, 41, 46, 133, 182, 197, 199, *see also* double annotation

behavioristic ideal, 40

memo, 77

segmentation, 181

shift-motivated, 203

ask, 38, 41, 47–49, 52, 81, 112, 114, 161

ask\_design, 60, 65, 67, 82

ask\_finding, 128, 129

ask\_knowledge, 55, 66, 67, 81, 82, 103, 119, 120, 122, 124, 128, 129, 135, 140, 141, 144, 147, 148, 152–154, 158, 159, 161–163, 185, 187, 188, 206

ask\_standard of knowledge, 55, 141

ask\_step, 75, 79, 82, 103, 140

*ask\_strategy*, 95, 96, 99, 103

AT, 140, 141, 163

*attest\_standard of knowledge*, 142

axial coding, 25

## B

BA1, 55, 59, 60, 63–65, 69, 70, 75, 78, 80, 83, 85, 95, 106, 117–119, 121–123, 128, 132, 136, 140–144, 149, 152, 158, 160–162, 168, 170–172, 183–185, 202, 205

base concept, 23, 40

style of definition, 198

base concept set, 23, 27, 35, 39, 207,

*see also* coding scheme

and episodes, 40

base layer, 23, 24, 27, 30, 31, 37, 95, 126, 181, 200

and other layers, 211

base layer modification, 27, 30, 33, 35, 37, 39, 41, 48, 49, 53, 84, 85, 88, 95, 101, 102, 110, 111, 114, 125, 128, 133, 137, 148, 152, 157, 161, 162, 166, 169–171, 173, 175, 181, 187, 189, 190, 193, 198, 200, 201, 203–205, 207–210

*become\_driver*, 188, 193

## C

CA2, 59, 60, 63, 64, 66, 70, 75–77, 87, 89, 95, 96, 99, 102, 106, 115, 117, 118, 125, 126, 132–134, 139, 140, 143, 152, 154, 156, 159, 160, 162, 163, 166, 172–174, 177, 187

can-check, 132

category, 24

*challenge*, 41, 47–49, 57, 64, 78, 98, 107, 125, 148, 167, 173, 200, 204

*challenge\_activity*, 169, 171–174

*challenge\_completion*, 84

*challenge\_design*, 60, 62–64, 116, 123

*challenge\_finding*, 120, 122, 125–127, 129, 161, 200, 204

*challenge\_hypothesis*, 133, 134

*challenge\_knowledge*, 126, 153, 155–158, 161

*challenge\_requirement*, 70, 71

*challenge\_state*, 105–107

*challenge\_step*, 63, 75, 77, 80, 82, 172

*challenge\_strategy*, 63, 96, 98, 99, 102

code, 46

coding scheme, 28, 35, 35, 207

*completion*, 46, 47, 73, 83–85, 105, 167

*agree\_*, 84

*challenge\_*, 84

*disagree\_*, 84

*explain\_*, 56, 83–85, 107, 130, 172

*completion/state*

*explain\_*, 111

concept, 24, 25, 31

concept category, 36

HCI/HEI, 36

HHI, 36

concept class, 29, 36, 37, 202

facade concepts, 37

for HCI/HEI concepts, 36

concept name, 45

constant comparison, 25

constructive concept, 49

context, 25, 37, 38, 59, 61, 62, 76, 90, 120, 128, 137, 182, 197, 199, 205

## D

D, 114, 115, 117, 189

DC, 115, 118

*decide*, 41, 47–49, 64

*decide\_design*, 60, 64, 89

*decide\_step*, 75, 78

*decide\_strategy*, 96, 99, 102

*design*, 46, 56, 59, 60, 65–67, 73, 74, 76, 78, 79, 82, 85, 89–91, 94, 99, 101, 122, 161, 181

- \*\_, 66
- agree\_, 60, 61, 64, 65, 81, 123, 124, 156, 161, 183
- amend\_, 60–64, 80, 84, 89, 90, 101, 183, 202, 204
- ask\_, 60, 65, 67, 82
- challenge\_, 60, 62–64, 116, 123
- decide\_, 60, 64, 89
- disagree\_, 60, 64, 66, 155, 161
- explain\_, 56
- justify\_, 152
- propose\_, 53–56, 59–64, 66, 67, 71, 72, 75, 77, 79, 80, 84, 90, 91, 101, 115, 122, 123, 130, 152, 155, 156, 159, 166, 167, 172, 173, 202, 203, 206
- design/requirement/step/strategy/todo propose\_, 111
- dimensionalization, 25
- disagree, 41, 47–49, 57, 65, 67, 78, 80, 125, 127, 145, 148, 158, 161, 173, 204, 209
- disagree\_activity, 118, 169, 171, 172, 174
- disagree\_completion, 84
- disagree\_design, 60, 64, 66, 155, 161
- disagree\_finding, 120, 125, 129, 144, 161, 187
- disagree\_hypothesis, 133, 134
- disagree\_knowledge, 153, 156–158
- disagree\_requirement, 72
- disagree\_step, 33, 40, 75, 78, 82, 88, 105, 106, 155
- disagree\_strategy, 88, 96, 98, 99
- disagree\_todo, 209
- DO, 115, 118, 154
- do, 181, 182
- do\_sth, 168, 169, 172, 185, 187, 188, 190, 198
- doing, 36
- double annotation, 67, 79, 90, 102, 107, 112, 114, 126, 143, 159–161, 163, 165, 193, 199, 200
- doubted, 131, 133
- DPR, 93–95, 97, 99, 100
- driver
  - become\_, 188, 193
- DU, 114, 115, 117–119, 122
- E**
- early repetition, 203, 204
- episode, 40, 41, 62, 128, 201, *see also* example
  - shift, 203
- example, 55, 59–62, 65–66, 69–71, 74–81, 83–84, 87–89, 95–97, 102–103, 105–106, 115–120, 122–129, 134–136, 141–142, 152–157, 162–163, 171–173, 182–189, 202
  - for ostensive definition, 28
  - reference to, 28, 33, 41, 56, 60, 62–64, 67, 70, 71, 76–79, 81, 84, 85, 89, 90, 96–99, 101, 102, 106, 117, 125, 128, 130, 134, 141–144, 154–158, 161–163, 171–174, 181, 190, 200–202, 204–206
- explain, 47–49, 56, 57, 123, 133, 136, 199, 200
- explain\_completion, 56, 83–85, 107, 130, 172
- explain\_completion/state, 111
- explain\_design, 56
- explain\_finding, 55, 64–66, 72, 74–76, 78, 79, 82, 85, 89, 90, 99, 107, 112, 115, 117–128, 130, 133, 134, 136, 137, 140, 145, 157, 159–162, 183–189, 199, 202–204
- explain\_gap in knowledge, 112, 147–149, 210
- explain\_hypothesis, 134
- explain\_knowledge, 30, 33, 53–56, 64, 66, 67, 71, 72, 74, 76, 79, 82, 85, 87–90, 96, 99, 102, 103, 111, 112, 114, 120, 121, 125, 133, 135, 136, 140–142, 147,

152–163, 167, 174, 188, 199,  
202, 204, 209

*explain\_responsibility*, 152

*explain\_standard of knowledge*, 61, 97,  
102, 112, 113, 121, 128, 135,  
139–145, 148, 149, 155, 156,  
162, 163, 167, 172

*explain\_state*, 56, 97, 105–107, 130,  
209

*explain\_step*, 48

*explore*, 181, 182

*explore\_sth*, 171, 173, 185–187, 190,  
198

EXS, 93, 94, 97, 99

**F**

facade concept class, 165

facade concepts, 37

finding, 113

*finding*, 46, 52, 54–56, 66, 77, 79, 99,  
110, 111, 113–120, 122, 123,  
125, 126, 128–130, 136, 142,  
144, 151, 153, 154, 159–161,  
167, 168, 200, 207

*agree\_*, 115, 120, 121, 123–128,  
144, 156, 157, 183, 189, 202,  
203, 213

*amend\_*, 54, 115, 120, 122–125,  
128, 129, 154, 157, 161

*ask\_*, 128, 129

*challenge\_*, 120, 122, 125–127, 129,  
161, 200, 204

*disagree\_*, 120, 125, 129, 144, 161,  
187

*explain\_*, 55, 64–66, 72, 74–76,  
78, 79, 82, 85, 89, 90, 99,  
107, 112, 115, 117–128, 130,  
133, 134, 136, 137, 140, 145,  
157, 159–162, 183–189, 199,  
202–204

finding type, 114–119, 122, 154, 157,  
160, 184, 189

D, 114, 115, 117, 189

DC, 115, 118

DO, 115, 118, 154

DU, 114, 115, 117–119, 122

P, 114, 117–119

T, 116, 117

TB, 116, 118

TC, 116, 118, 119, 157, 160, 184

TU, 116, 160

*findings*, 125

forcing, 25, 25, 26, 27

## G

*gap in information*, 117

*gap in knowledge*, 46, 110, 111, 113,  
139, 147, 148, 210

*agree\_*, 147–149

*explain\_*, 112, 147–149, 210

*gap of knowledge*, 47

gibberish, 177

grounded theory, 26

grounding, 24, 25, 28, 39, 48, 53, 69

## H

hard-to-verify, 131, 132

*hypothesis*, 46, 54, 56, 110, 111, 113,  
128, 131–133, 136, 144, 159,  
161

*agree\_*, 133–135, 171, 186

*amend\_*, 133–136

*challenge\_*, 133, 134

*disagree\_*, 133, 134

*explain\_*, 134

*propose\_*, 99, 102, 111, 133–136,  
145, 158, 161, 162, 171, 185,  
186, 202, 203

hypothesis type, 131–133

can-check, 132

doubted, 131, 133

hard-to-verify, 131, 132

## I

illocution, 27, 31, 39, 49, 66, 152, 154,  
197, 209

and intention, 38

- behavioristic ideal, 40
  - for activities, 198
  - indirect speech act, 63
  - of justifications, 204
  - primary and secondary, 63, 82, 199
  - illocutionary act, 38
  - immediate repetition, 203
  - implicit action announcement, 201
  - initiative verb, 49
  - intention, *see* illocution
  - interrupt*
    - react to*\_, 194
  - issue type, 132
- J**
- justification, 204
  - justification*, 204
  - justify*, 152
  - justify\_design*, 152
- K**
- knowledge, 109
  - knowledge*, 46, 48, 52–54, 56, 64, 66, 67, 79, 99, 111, 113, 114, 116, 117, 122, 130, 136, 144, 148, 151–154, 159, 160, 168, 207, 209
  - agree*\_, 102, 103, 120, 142, 148, 153, 155, 156, 158, 163, 202, 213
  - amend*\_, 154
  - ask*\_, 55, 66, 67, 81, 82, 103, 119, 120, 122, 124, 128, 129, 135, 140, 141, 144, 147, 148, 152–154, 158, 159, 161–163, 185, 187, 188, 206
  - challenge*\_, 126, 153, 155–158, 161
  - disagree*\_, 153, 156–158
  - explain*\_, 30, 33, 53–56, 64, 66, 67, 71, 72, 74, 76, 79, 82, 85, 87–90, 96, 99, 102, 103, 111, 112, 114, 120, 121, 125, 133, 135, 136, 140–142, 147, 152–163, 167, 174, 188, 199, 202, 204, 209
- known, 127, 155
- L**
- later repetition, 204
  - layers, 211
  - LO, 63, 66, 76, 102
- M**
- me-too*, 128
  - memo, 25
  - mumble*, 47–49
  - mumble\_sth*, 49, 61, 62, 115, 124, 135, 142, 173, 177, 188
- O**
- object, 46, 209
  - OE, 63, 64, 96
  - off topic*, 47, 48
    - say*\_, 49, 177, 184
  - open coding, 25
  - OWP, 93–95, 97–99, 101
  - OWS, 97
- P**
- P, 114, 117–119
  - P&P concepts, 37, 50, 57, 110, 152, 158, 161
  - pair coding, 26
  - paraphrasing, 134, 141, 142, 149, 162, 205, 206
  - PI, 63, 64, 96
  - primary characterizing attribute, 28, 35, 116, 128, 202, 209, *see also* property
  - agreement type, 127, 128, 155, 157, 213
  - finding type, 114–119, 122, 154, 157, 160, 184, 189
  - hypothesis type, 131–133
  - proposal mode, 63, 64, 66, 76, 96, 102, 213

standard of knowledge utterance type, 139–141, 153, 163  
 strategy type, 93–95, 97–101  
 think aloud topic, 167–169  
 primary intention, 38  
 property, 25, 28, 90, 94, 132, 152, 161, 209, 212, *see also* primary characterizing attribute  
 proposal mode, 63, 64, 66, 76, 96, 102, 213  
     LO, 63, 66, 76, 102  
     OE, 63, 64, 96  
     PI, 63, 64, 96  
 propose, 38, 41, 47–49, 56, 57, 64, 72, 81, 96, 98, 107, 126, 133, 201  
 propose\_design, 53–56, 59–64, 66, 67, 71, 72, 75, 77, 79, 80, 84, 90, 91, 101, 115, 122, 123, 130, 152, 155, 156, 159, 166, 167, 172, 173, 202, 203, 206  
 propose\_design/requirement/step/strategy/todo, 111  
 propose\_hypothesis, 99, 102, 111, 133–136, 145, 158, 161, 162, 171, 185, 186, 202, 203  
 propose\_requirement, 69–72  
 propose\_step, 40, 56, 61, 63, 67, 75–82, 89, 90, 95, 100, 101, 105–107, 115, 119, 123, 124, 126, 127, 130, 134, 143, 148, 149, 154, 159, 167, 171, 172, 184, 186, 187, 189, 200  
 propose\_strategy, 56, 63, 90, 95–97, 99–102  
 propose\_todo, 67, 87–90, 97, 99, 107  
 proposed\_step, 80  
 propose\_step, 174  
 PT, 139  
 PTd, 139  
 PTe, 140  
 PTo, 140, 153

## R

react\_to\_interrupt, 194  
 reactive verb, 49  
 read, 181, 182  
 read\_sth, 185, 189  
 remember, 48, 49, 71  
 remember\_requirement, 70–72, 111  
 remember\_source of information, 209, 210  
 remind, 72  
 remind\_requirement, 71, 72  
 requirement, 47, 56, 69–71, 73, 74  
     agree\_, 70, 71  
     amend\_, 72  
     challenge\_, 70, 71  
     disagree\_, 72  
     propose\_, 69–72  
     remember\_, 70–72, 111  
     remind\_, 71, 72  
 requirements, 60, 69  
 responsibility  
     explain\_, 152  
 RT, 140

## S

say, 48, 49  
 say\_off topic, 49, 177, 184  
 saying, 36  
 search, 181, 182  
 search\_sth, 183–185, 194, 198  
 selective coding, 26  
 Session  
     BA1, 55, 59, 60, 63–65, 69, 70, 75, 78, 80, 83, 85, 95, 106, 117–119, 121–123, 128, 132, 136, 140–144, 149, 152, 158, 160–162, 168, 170–172, 183–185, 202, 205  
     CA2, 59, 60, 63, 64, 66, 70, 75–77, 87, 89, 95, 96, 99, 102, 106, 115, 117, 118, 125, 126, 132–134, 139, 140, 143, 152, 154, 156, 159, 160, 162, 163, 166, 172–174, 177, 187

- ZB7, 59, 79, 94, 95, 105, 126, 132,  
133, 135, 147, 185, 209
- shift, 202
- show, 181, 182
- show\_sth, 184, 189–191
- sketch, 181, 182
- sketch\_sth, 189, 191
- something, 47, 181
- source of information, 210  
remember\_, 209, 210
- standard of knowledge, 46, 47, 52, 54,  
110, 111, 113, 139, 141, 142,  
147, 148, 151  
agree\_, 148, 149  
ask\_, 55, 141  
attest\_, 142  
explain\_, 61, 97, 102, 112, 113,  
121, 128, 135, 139–145, 148,  
149, 155, 156, 162, 163, 167,  
172
- standard of knowledge utterance  
type, 139–141, 153, 163  
AT, 140, 141, 163  
PT, 139  
PTd, 139  
PTe, 140  
PTo, 140, 153  
RT, 140
- state, 46, 47, 73, 83, 85, 105, 106  
agree\_, 97, 106  
challenge\_, 105–107  
explain\_, 56, 97, 105–107, 130,  
209
- step, 40, 46, 47, 56, 63, 67, 73–76, 79,  
81–83, 85, 87–90, 93, 94, 96,  
99–101, 105, 107, 130, 181  
\*\_ , 79, 130  
agree\_, 75–78, 127, 157  
amend\_, 63, 75–78, 82, 184, 204  
ask\_, 75, 79, 82, 103, 140  
challenge\_, 63, 75, 77, 80, 82, 172  
decide\_, 75, 78  
disagree\_, 33, 40, 75, 78, 82, 88,  
105, 106, 155  
explain\_, 48  
propose\_, 40, 56, 61, 63, 67, 75–  
82, 89, 90, 95, 100, 101, 105–  
107, 115, 119, 123, 124, 126,  
127, 130, 134, 143, 148, 149,  
154, 159, 167, 171, 172, 184,  
186, 187, 189, 200  
proposed\_, 80  
propse\_, 174
- step with strategic character, 100
- sth, 36, 47, 48, 181  
do\_, 168, 169, 172, 185, 187, 188,  
190, 198  
explore\_, 171, 173, 185–187, 190,  
198  
mumble\_, 49, 61, 62, 115, 124,  
135, 142, 173, 177, 188  
read\_, 185, 189  
search\_, 183–185, 194, 198  
show\_, 184, 189–191  
sketch\_, 189, 191  
verify\_, 117, 169, 170, 185–187,  
189–191, 194, 198  
wait for\_, 194  
work alone\_, 193  
work in parallel\_, 193  
write\_, 166, 168, 172, 173, 182,  
183, 187, 191
- stop, 48, 49
- stop\_activity, 169, 171, 173, 174
- strategy, 35, 40, 47, 56, 63, 64, 67,  
73, 74, 76, 79, 83, 85, 87, 89,  
90, 93–96, 98–101, 105–107,  
167, 208  
agree\_, 88, 96, 97, 99, 102, 103  
amend\_, 33, 63, 87, 88, 96–99  
ask\_, 95, 96, 99, 103  
challenge\_, 63, 96, 98, 99, 102  
decide\_, 96, 99, 102  
disagree\_, 88, 96, 98, 99  
propose\_, 56, 63, 90, 95–97, 99–  
102
- strategy type, 93–95, 97–101  
DPR, 93–95, 97, 99, 100

EXS, 93, 94, 97, 99  
 OWP, 93–95, 97–99, 101  
 OWS, 97

## T

T, 116, 117  
 TA1, 167, 168  
 TA10, 167, 168  
 TA11, 168  
 TA2, 167, 168  
 TA3, 167, 169  
 TA4, 167–169  
 TA5, 167, 169  
 TA6, 167–169  
 TA7, 167, 169  
 TA8, 167  
 TA9, 167  
 TB, 116, 118  
 TC, 116, 118, 119, 157, 160, 184  
 thematic shift, 202  
 theoretical coding, 25  
 theoretical sampling, 25, 26  
 theoretical saturation, 26  
 theoretical sensitivity, 25  
 theory, 24  
 Think, 33, 90, 98, 103, 111, 113, 145,  
 158, 160, 162, 165, 170, 181,  
 185, 187, 190, 191, 197, 198,  
 204, 205  
*think aloud*, 48, 49, 56  
 think aloud topic, 167–169  
   TA1, 167, 168  
   TA10, 167, 168  
   TA11, 168  
   TA2, 167, 168  
   TA3, 167, 169  
   TA4, 167–169  
   TA5, 167, 169  
   TA6, 167–169  
   TA7, 167, 169  
   TA8, 167  
   TA9, 167  
*think aloud\_activity*, 56, 166, 167, 169–  
 175, 183–189

*todo*, 47, 56, 67, 73, 74, 79, 87–90, 209  
*agree\_*, 88  
*disagree\_*, 209  
*propose\_*, 67, 87–90, 97, 99, 107  
 topic, *see* episode  
 transcription markup, 31  
 TU, 116, 160

## U

unchecked, 155, 157  
 understood, 128, 155  
 universal concepts, 111  
 utterance, 31, 39, 40  
   on activity, 165  
   reply, 49  
   segmentation, 200  
   transcription markup, 31  
   vs. the thing itself, 40

## V

verb, 47, 49, 209  
   bivalent, 49  
   constructive, 49  
   initiative, 197  
   reactive, 198  
   unconstructive, 49  
*verify*, 181, 182  
*verify\_sth*, 117, 169, 170, 185–187,  
 189–191, 194, 198

## W

*wait for\_sth*, 194  
*work alone\_sth*, 193  
*work in parallel\_sth*, 193  
*write*, 181, 182  
*write\_sth*, 166, 168, 172, 173, 182,  
 183, 187, 191

## Z

ZB7, 59, 79, 94, 95, 105, 126, 132, 133,  
 135, 147, 185, 209



Stephan Salinger • Lutz Prechelt

## Understanding Pair Programming: The Base Layer

There has been and still is a lot of controversy on whether pair programming is a useful engineering technique – as if this would not strongly depend on the specific goals, task, and the pair's pair programming skill. Rather than providing still more bottom-line, quantitative results on pair programming, a research group at Freie Universität Berlin set out to decipher

- what is the actual process of pair programming and
- what is pair programming skill.

This book provides a set of concepts that serves as the infrastructure for studies of pair programming that focus on qualitative data analysis. It promises to connect the results of such studies to one another.

The book is oriented towards researchers only, not towards practitioners.

