CS6310 – Software Architecture and Design

Spencer Rugaber

Contents

P1L1 Introduction 1
- 01 – Welcome 1
- 02 – Introductions 1
- 03 – Course 1
- 04 – Objectives 2
- 05 – Course Structure 2
- 06 – Readings 2
- 07 – Assignments 2
- 08 – Projects Design Studies 2
- 09 – Policies 3
- 10 – Grading 4
- 11 – Class Participation 4
- 12 – Resources 4
- 13 – Conclusion 4

P1L2 Text Browser Exercise (Analysis) 5
- 01 – Introduction 5
- 02 – Text Browser Exercise 5
- 03 – GUI Elements Quiz 5
- 04 – GUI Elements Solution 5
- 05 – FileManager 5
- 06 – ViewPort 5
- 07 – ScrollBar 6
- 08 – Use Cases 6
- 09 – Use Cases Quiz 6
- 10 – Use Cases Quiz Solution 6
- 11 – Analysis Model 6
- 12 – Classes Quiz 7
- 13 – Classes Quiz Solution 7
- 14 – Operations 7
- 15 – Operations Quiz 7
- 16 – Operations Quiz Solution 7
- 17 – Visible Attributes Quiz 8
- 18 – Visible Attributes Quiz Solution 8
- 19 – FileManager 8
- 20 – Relationships 9
- 21 – Relationships Quiz 9
- 22 – Relationships Quiz Solution 9
- 23 – Number of Lines Quiz 9

24 – Number of Lines Quiz Solution 10
25 – LinesVisible Association 10
26 – Another Association Quiz 10
27 – Another Association Quiz Solution 10
28 – Explanation 11
29 – Displays Diagram 11
30 – Handle Association Quiz 11
31 – Handle Association Quiz Solution 11
32 – HandleProportion 11
33 – Subtleties 11
34 – Summary 12

P1L3 Design Concepts 13
01 – Design Concepts 13
02 – Terms Quiz 13
03 – Terms Quiz Solution 13
04 – Programming or Design Quiz 13
05 – Programming or Design Quiz Solution 13
06 – Software Design 14
07 – Design Notation 14
08 – Weather Quiz 14
09 – Weather Quiz Solution 14
10 – Approaches to Software Design 15
11 – Issues with Design 15
12 – Design Review Quiz 15
13 – Design Review Quiz Solution 15
14 – Design Validation 15
15 – Other Design Issues 16
16 – Design Documentation 16
17 – Documentation Quiz 16
18 – Documentation Quiz Solution 16
19 – Traditional Design Documentation 17
20 – Leonardo Objects 17
21 – Design Rationale 17
22 – Coupling and Cohesion 17
23 – Java Quiz 1 18
24 – Java Quiz 1 Solution 18
25 – Java Quiz 2 18
26 – Java Quiz 2 Solution 18
27 – Information Hiding 18
28 – Abstraction and Refinement 19
29 – Aesthetics 19
30 – Design Philosophy 19
31 – Metaphors Quiz 20
32 – Metaphors Quiz Solution 20
33 – Summary 20

P2L01 Review of UML 21
01 – Diagrams 21

02 – OMT 21
03 – UML 21
04 – Diagram Types 22
05 – Diagram Quiz 22
06 – Diagram Quiz Solution 22
07 – Class Models 22
08 – UML Classes 23
09 – UML Relationships 23
10 – Example Class Diagram 23
11 – Object Diagram 23
12 – Composite Structure Diagram 23
13 – Component Diagram 24
14 – Example Component Diagram 24
15 – Deployment Diagram 24
16 – Example Deployment Diagram 24
17 – Packages 24
18 – Example Class Diagram with Packages 24
19 – Example Package Diagram 24
20 – Profile Diagram 25
21 – Example Profile Diagram 25
22 – UML Structure Diagram Quiz 25
23 – UML Structure Diagram Quiz Solution 25
24 – UML Quiz 25
25 – UML Quiz Solution 26
26 – Behavior Diagrams 26
27 – Use Case Diagram 26
28 – Use Case Diagrams 26
29 – Example of Use Case Diagrams 26
30 – Individual Use Cases 26
31 – Tabular Version of Example 27
32 – Context Diagrams 27
33 – Example Context Diagrams 27
34 – Sequence Diagram 27
35 – Example Sequence Diagram 27
36 – Communication Diagram 27
37 – Example Communication Diagram 27
38 – Activity Diagram 28
39 – Example Activity Diagram 28
40 – Interaction Overview Diagram 28
41 – Timing Diagram 28
42 – State Diagrams 29
43 – Example State Machine Diagrams 29
44 – Behavior Diagram Quiz 29
45 – Behavior Diagram Quiz Solution 29
46 – Object Constraint Language 29
47 – Example OCL 30
48 – UML MetaModel 30
49 – Class Model of UML MetaModel 30

50 – Summary 30

P2L02 Object Oriented Analysis Exercise 31
01 – Analysis 31
02 – Object Oriented Analysis OOA 31
03 – Object Quiz 31
04 – Object Quiz Solution 31
05 – Object Oriented Analysis and Design 31
06 – OOA 32
07 – Steps in OOA 32
08 – Technique 32
09 – Step 1 Locate Nouns Quiz 32
10 – Step 1 Locate Nouns Quiz Solution 33
11 – Issues 33
12 – Step 2 Candidate Classes Quiz 33
13 – Step 2 Candidate Classes Quiz Solution 33
14 – Initial Class Model Diagram 34
15 – Caveats 34
16 – Step 3 Adjectives Quiz 34
17 – Step 3 Adjectives Quiz Solution 35
18 – Adjective Issues 35
19 – Updated Class Model 35
20 – Step 4 Operations 35
21 – Action Verbs Quiz 36
22 – Action Verbs Quiz Solution 36
23 – Operations in Class Quiz 36
24 – Operations in Class Quiz Solution 36
25 – Operations in Classes 37
26 – Operation Issues 37
27 – Step 5 Relationships 38
28 – Generalizations 38
29 – Generalization Quiz 38
30 – Generalization Quiz Solution 38
31 – Aggregations 38
32 – Aggregation Quiz 38
33 – Aggregation Quiz Solution 38
34 – Associations 39
35 – Associations Quiz 39
36 – Associations Quiz Solution 39
37 – Relationship Issues 39
38 – Summary 39

P2L03 UML Class Models 41
01 – Introduction 41
02 – Classes 41
03 – Name Compartment 41
04 – Class Features 42
05 – Attributes Compartment 42
06 – Operations Compartment 42

07 – Abstract Class Quiz 43
08 – Abstract Class Quiz Solution 43
09 – More Example Classes 43
10 – Class Description Quiz 44
11 – Class Description Quiz Solution 44
12 – Advanced Features 44
13 – Relationships 44
14 – Associations 44
15 – Assocation Class 45
16 – Aggregation Composition 46
17 – Aggregation Composition Quiz 46
18 – Aggregation Composition Quiz Solution 46
19 – Qualifiers 46
20 – Links 47
21 – Generalizations 47
22 – Constraints Quiz 48
23 – Constraints Quiz Solution 48
24 – Superclass Subclass Quiz 48
25 – Superclass Subclass Quiz Solution 48
26 – Summary Quiz 48
27 – Summary Quiz Solution 48
28 – Summary 49

P2L04 Design Studies 51
01 – Design 51
02 – Design Studies 51
03 – Definition 51
04 – Design Spaces 51
05 – Design Factors Quiz 51
06 – Design Factors Quiz Solution 51
07 – Teaching and Learning 52
08 – Projects 52
09 – Experiments 52
10 – Report 52
11 – 1 Context 52
12 – 2 Research Questions 52
13 – 3 Subject 53
14 – 4 Experimental Conditions 53
15 – 5 Variables 53
16 – 6 Method 53
17 – 7 Results 54
18 – 8 Discussion 54
19 – 9 Conclusions 54
20 – Deliverables 54
21 – Wrap Up 54

P2L05 Library Exercise (UML) 55
01 – Introduction 55
02 – Analyzing Requirements 55

03 – Refining Classes and Attributes 57
04 – Adding Attributes 58
05 – Identifying Operations 59
06 – Adding Relationships 60
07 – Refining Relationships 61
08 – Refining the Class Diagram 62
09 – Final Considerations 63
10 – Debriefing 63

P2L06 Formal Specification Exercise 65
01 – Specification 65
02 – FOL 65
03 – OCL 65
04 – Sorting 65
05 – Exercise Introduction 65
06 – Input Type 66
07 – Output Type 66
08 – Ordering 66
09 – Sensitivity to Input 66
10 – Circularity 67
11 – SORT in English 67
12 – Process 67
13 – Signature 67
14 – Comments on Signatures 67
15 – SQRT Signature Quiz 68
16 – SQRT Signature Quiz Solution 68
17 – SQRT Preconditions 68
18 – SORT Preconditions 68
19 – Postconditions 69
20 – SQRT Postcondition Quiz 69
21 – SQRT Postcondition Quiz Solution 69
22 – Comments on Postconditions 69
23 – Postconditions for SORT 70
24 – Ordered 70
25 – Elements 70
26 – ORDERED Precondition Quiz 71
27 – ORDERED Precondition Quiz Solution 71
28 – ORDERED Postcondition 72
29 – RORDERED Spec and Pre Quiz 72
30 – RORDERED Spec and Pre Quiz Solution 72
31 – RORDERED Postcondition 72
32 – SAME–ELEMENTS–AS Signature 72
33 – SAME–ELEMENTS–AS in English 73
34 – Permutation 73
35 – PERMUTATION Signature Quiz 73
36 – PERMUTATION Signature Quiz Solution 73
37 – PERMUTATION Postcondition 73
38 – Non Empty Case 74
39 – Non Matching Case 74

40 – Recursion 74
41 – Pasting 75
42 – Third Case 75
43 – All Together 75
44 – Some Questions 76
45 – OCL 76
46 – Notes 76
47 – Summary 76

P2L07 OCL 77
01 – OCL 77
02 – Why Do We Need OCL 77
03 – OCL Overview 77
04 – Uses of OCL 77
05 – Syntax 78
06 – Invariants 78
07 – Role of Invariants 78
08 – Invariant Constraint Quiz 78
09 – Invariant Constraint Quiz Solution 78
10 – Pre and Post Conditions 79
11 – Pre and Post Conditions Example 79
12 – Changes to Attribute Values 79
13 – Post Condition Quiz 80
14 – Post Condition Quiz Solution 80
15 – OCL Built in Types 80
16 – OCL Keywords 80
17 – Let Clause 80
18 – Navigation 81
19 – Navigation Example 81
20 – Navigation Multiplicity 81
21 – Bank ID Quiz 81
22 – Bank ID Quiz Solution 81
23 – Collections 82
24 – Other OCL Features 82
25 – Summary 82

P2L08 Library Exercise (OCL) 83
01 – Library Exercise 83
02 – Library Problem Requirements 83
03 – Class Model Diagram 83
04 – Requirements Quiz 83
05 – Requirements Quiz Solution 83
06 – Limitations 84
07 – Requirement 6 84
08 – Requirement 6 Quiz 85
09 – Requirement 6 Quiz Solution 85
10 – Checked Out Quiz 85
11 – Checked Out Quiz Solution 85
12 – Requirement 6 OCL 85

13 – Explanation 86
14 – Requirement 7 OCL Quiz 86
15 – Requirement 7 OCL Quiz Solution 86
16 – Operations 86
17 – CheckedOut Operation 87
18 – Explanation 88
19 – Requirement 4 OCL Quiz 88
20 – Requirement 4 OCL Quiz Solution 89
21 – Requirement 4 Explanation 90
22 – Side Effects 90
23 – Requirement 5 Signature 90
24 – Checkout Preconditions Quiz 91
25 – Checkout Preconditions Quiz Solution 91
26 – Requirement 5 91
27 – Checkout Postconditions 92
28 – Further Checkout Explanation 92
29 – Postconditions 93
30 – Derived Data Quiz 94
31 – Derived Data Quiz Solution 95
32 – Missing Pieces 95
33 – Observations 96

P2L09 Behavior Modeling 97
01 – Behavior Modeling 97
02 – States 97
03 – Tic Tac Toe Quiz 97
04 – Tic Tac Toe Quiz Solution 97
05 – Events 97
06 – UML Event Taxonomy 98
07 – State vs Event Quiz 98
08 – State vs Event Quiz Solution 98
09 – Modeling Techniques 98
10 – Combinatorial Modeling 99
11 – Decision Tables 99
12 – Decision Trees 99
13 – Sequential Systems 100
14 – State Transition Table STT 100
15 – Garage Door Quiz 1 100
16 – Garage Door Quiz 1 Solution 100
17 – Garage Door Quiz 2 101
18 – Garage Door Quiz 2 Solution 101
19 – STT for Garage Door System 101
20 – State Transition Diagrams 101
21 – Example Garage Door 101
22 – Example Telephone 102
23 – Problems with State Transition Diagrams 102
24 – State Charts 102
25 – State Chart Icons 103
26 – State Chart Extensions to FSMs 103

27 – State Chart Nesting 103
28 – State Chart Nesting UML Example 104
29 – UML Example Harels Notation 104
30 – Concurrency 104
31 – Synchronization 105
32 – Broadcast Cascade Events 105
33 – Data Conditions 105
34 – Special Transitions 106
35 – History States 106
36 – Complete UML State Description 106
37 – Complete UML Transition Description 107
38 – Relationship to Class Diagram 107
39 – Harels Digital Watch 107
40 – Harels Digital Watch Quiz 108
41 – Harels Digital Watch Quiz Solution 108
42 – Summary 108

P2L10 Clock Radio Exercise 111
01 – Modeling with Statecharts 111
02 – Description 111
03 – Exercise Introduction 111
04 – Percepts Quiz 112
05 – Percepts Quiz Solution 112
06 – Percept States Quiz 113
07 – Percept States Quiz Solution 114
08 – Display FSM 114
09 – Mode Switch Quiz 114
10 – Mode Switch Quiz Solution 114
11 – Station Indicator 115
12 – Station Indicator FSM 115
13 – Speaker 115
14 – So Far 115
15 – External Controls and Stimuli 115
16 – From Actions to Events 116
17 – Outermost Layer StateChart Quiz 117
18 – Outermost Layer StateChart Quiz Solution 117
19 – Adding Events 117
20 – Event Allocation 117
21 – New Sub–Machines 118
22 – Setting the Time 118
23 – Responses to Events 118
24 – Stimulus Response Table Quiz 119
25 – Stimulus Response Table Quiz Solution 119
26 – Stimulus Response Table 119
27 – Timer Events Quiz 119
28 – Timer Events Quiz Solution 119
29 – Internal States 120
30 – Other Internal Events 121
31 – Guarded Transitions 121

32 – Cascaded Events 121
33 – Example 121
34 – Still To Do 121
35 – Validation 121
36 – Statechart Modeling Method 122
37 – Conclusion 122

P3L01 KWIC Exercise 123
01 – Software Architecture 123
02 – Key Word in Context 123
03 – Example of Circular Shifts 123
04 – Example with Multiple Titles 123
05 – KWIC Exercise 123
06 – Diagramming KWIC Quiz 124
07 – Diagramming KWIC Quiz Solution 124
08 – Components 125
09 – Shared Data 125
10 – Pipe and Filter 125
11 – Pipe and Filter Diagram 126
12 – Abstract Data Types 126
13 – Abstract Data Type Diagram 127
14 – Implicit Invocation 127
15 – Implicit Invocation Diagram 127
16 – Shared Data Approach Quiz 127
17 – Shared Data Approach Quiz Solution 127
18 – Evaluation 128
19 – Enhancements Quiz 128
20 – Enhancements Quiz Solution 128
21 – Reusability Quiz 129
22 – Reusability Quiz Solution 129
23 – Data Change Resilience Quiz 129
24 – Data Change Resilience Quiz Solution 129
25 – Deletion Quiz 129
26 – Deletion Quiz Solution 130
27 – Lessons 130

P3L02 Overview of Architectural Styles 131
01 – Introduction 131
02 – Informal Definition 131
03 – Analysis to Components Quiz 131
04 – Analysis to Components Quiz Solution 131
05 – USP Definition 132
06 – Other Definitions 132
07 – Components 132
08 – Selecting Components 133
09 – APIs 133
10 – Connectors 133
11 – Example Connector 134
12 – Configuration 134

13 – Terminology 134
14 – Architectural Views 135
15 – UML Diagram Quiz 135
16 – UML Diagram Quiz 135
17 – Architectural Styles 135
18 – Arch Style Quiz 135
19 – Arch Style Solution 136
20 – Architectural Styles cont 136
21 – Catalog of Styles 136
22 – More Styles 137
23 – Style Issues 138
24 – Architecture Description Language 139
25 – Architectural Evaluation 139
26 – SAAM 139
27 – Summary 140

P3L03 Architectural Views 141
01 – Architectural Views 141
02 – Logical View 141
03 – Developmental View 141
04 – Diagram Types Quiz 142
05 – Diagram Types Quiz 142
06 – Process View 142
07 – Physical View 142
08 – Use Case View 142
09 – Context View 143
10 – Individual Use Cases 143
11 – Feature View 143
12 – Feature Diagram Quiz 1 144
13 – Feature Diagram Quiz 1 144
14 – Feature Diagram Quiz 2 144
15 – Feature Diagram Quiz 2 144
16 – Non Functional View 144
17 – Non Functional Requirements Quiz 145
18 – Non Functional Requirements Quiz 145
19 – Bug Reporting View 145
20 – Utility Views 145
21 – Conclusion 146

P3L04 Text Browser Exercise (Arch) 147
01 – Text Browser Case Study 147
02 – Text Browser 147
03 – Exercise 147
04 – Phase 0 Preparation 148
05 – Phase 0 Summary 148
06 – Phase 1 148
07 – Phase 1 Steps 148
08 – Decomposition 149
09 – Phase 1 Diagram 149

10 – OCL Postcondition Constraint 149
11 – Another Postcondition 150
12 – Third OCL Constraint 150
13 – Phase 1 Summary 150
14 – Phase 2 150
15 – Phase 2 Steps 150
16 – Text Browser Arch Quiz 150
17 – Text Browser Arch Quiz Solution 151
18 – Layered Implicit Invocation 151
19 – Benefits and Costs 151
20 – Assigning Components to Layers 152
21 – Phase 2 Diagram 152
22 – OCL Updates 152
23 – Resize Window Quiz 152
24 – Resize Window Quiz Solution 153
25 – Constraint Placement 153
26 – Invariant Maintenance Quiz 153
27 – Invariant Maintenance Quiz 153
28 – Invariant Maintenance Strategies 154
29 – Centralized Strategy Quiz 154
30 – Centralized Strategy Quiz Solution 154
31 – Decentralized Strategy Quiz 154
32 – Decentralized Strategy Quiz Solution 154
33 – Tradeoff Between Locality and Complexity 154
34 – Example Continued 155
35 – Aggregation 155
36 – Aggregated Responsibility Quiz 155
37 – Aggregated Responsibility Quiz Solution 155
38 – Distributed Responsibility 155
39 – Distributed Responsibility Quiz 156
40 – Distributed Responsibility Quiz Solution 156
41 – Mediators 156
42 – Mediated Responsibility Quiz 156
43 – Mediated Responsibility Quiz Solution 157
44 – Summary of Process 157
45 – Conclusion 157

P3L05 Non-Functional Reqs & Arch Styles 159
01 – Non–Functional Reqs Arch Styles 159
02 – Qualities 159
03 – Non Functional Qualities Quiz 159
04 – Non Functional Qualities Solution 159
05 – Functional and Non Functional Requirements Quiz 159
06 – Functional and Non Functional Requirements Solution 160
07 – Quality Catalog 160
08 – Applications Quiz 160
09 – Applications Quiz 161
10 – Architectural Styles 161
11 – Review of Architectural Styles 161

12 – Pipe and Filter Performance 161
13 – Pipe and Filter Maintainability 162
14 – Pipe and Filter Other Qualities 162
15 – Layering Qualities 162
16 – Blackboard Reliability and Security 162
17 – Other Blackboard Qualities 163
18 – Object Orientation Maintainability 163
19 – Object Orientation Security 163
20 – Other Object Orientation Qualities 163
21 – Implicit Invocation Qualities 163
22 – Side Effects Quiz 1 164
23 – Side Effects Quiz 1 Solution 164
24 – Side Effects Quiz 2 164
25 – Side Effects Quiz 2 Solution 164
26 – Summary 165

P3L06 Connectors 167
01 – Connectors 167
02 – Atomic Elements 167
03 – Pipe and Filter Quiz 167
04 – Pipe and Filter Quiz 167
05 – Service Categories 167
06 – Services Quiz 168
07 – Services Quiz 168
08 – Variety of Connectors 168
09 – Procedure Call Connectors 168
10 – Event Connectors 169
11 – Data Access Connectors 169
12 – Linkage Connectors 170
13 – Stream Connectors 170
14 – Arbitrator Connectors 170
15 – Adaptor Connectors 171
16 – Distributor Connectors 171
17 – Summary of Connector Types 171
18 – Connector Type Quiz 171
19 – Connector Type Quiz 171
20 – Composite Connector Examples 172
21 – Connector Design 172
22 – Validation Rules 173
23 – Linux Case Study 173
24 – Summary 173

P3L07 Acme 175
01 – ADLs 175
02 – ACME 175
03 – ACME Features 175
04 – Architecture Vocabulary 175
05 – Simple Architecture Example 176
06 – ACME Quiz 176

07 – ACME Quiz 176
08 – ACME Graphical View 176
09 – Decomposition 177
10 – Representations 177
11 – Example Representation 177
12 – Extending ACME 177
13 – Properties 178
14 – Properties Example 178
15 – Families 178
16 – Example Family 178
17 – Open Semantic Framework 179
18 – Acme Features Quiz 1 179
19 – Acme Features Quiz 1 Solution 179
20 – Acme Features Quiz 2 179
21 – Acme Features Quiz 2 179
22 – ACME Limitations 180

P3L08 Refinement 181
01 – Complexity Abstraction 181
02 – Levels of Abstraction 181
03 – Divide Conquer 181
04 – Horizontal Decomposition 181
05 – Vertical Decomposition 181
06 – Proper Refinements 182
07 – Property 1 182
08 – Bank Account 182
09 – Bank Account Class 182
10 – Bank Account Quiz 1 182
11 – Bank Account Quiz 1 Solution 183
12 – Property 2 183
13 – Notation 183
14 – Valid Operations 184
15 – Implications of Property 2 184
16 – Property 3 184
17 – Bank Account Refinement 184
18 – Bank Account Quiz 2 185
19 – Bank Account Quiz 2 Solution 185
20 – Property 3 Details 185
21 – More Notation 185
22 – Adequate Representation 186
23 – Note on the Exercise 186
24 – Adequacy Quiz 186
25 – Adequacy Quiz 186
26 – Total Representation 186
27 – Totality Quiz 187
28 – Totality Quiz 187
29 – Models 187
30 – Operation Inputs 187
31 – Interpretation 187

32 – Inputs Quiz 187
33 – Inputs Quiz 188
34 – Outputs 188
35 – Outputs Quiz 188
36 – Outputs Quiz 188
37 – Satisfy Property 2 188
38 – Summary 189

P3L09 Middleware 191
01 – Architecture of Distributed Systems 191
02 – Middleware 191
03 – Context 191
04 – Needs 192
05 – Exercise Application 192
06 – Characteristic Issues 192
07 – Network Communication 192
08 – Data Transportability 193
09 – Voting Application Quiz 1 193
10 – Voting Application Quiz 1 Solution 193
11 – Transactions ACID 193
12 – Voting Application Quiz 2 194
13 – Voting Application Quiz 2 Solution 194
14 – Coordination 194
15 – Voting Application Quiz 3 195
16 – Voting Application Quiz 3 Solution 195
17 – Reliability 195
18 – Voting Application Quiz 4 195
19 – Voting Application Quiz 4 Solution 195
20 – Voting Application Quiz 5 196
21 – Voting Application Quiz 5 Solution 196
22 – Scalablity 196
23 – Kinds of Transparancy 196
24 – Heterogeneity 197
25 – Implications of Heterogeneity 197
26 – LAMP Quiz 197
27 – LAMP Quiz Solution 197
28 – Other Non Functional Issues 198
29 – Challenges 198
30 – Kinds of Middleware 198
31 – Transactional Middleware 198
32 – Message Oriented Middleware 199
33 – Procedural Middleware 199
34 – Object and Component Middleware 199
35 – Middleware Quiz 199
36 – Middleware Quiz Solution 200
37 – Software Engineering Issues 200
38 – Research Questions 200
39 – Examples 201
40 – Service Oriented Quiz 201

41 – Service Oriented Quiz Solution 201
42 – Web Services 201
43 – Web Services Protocols 202
44 – J2EE System 202
45 – SOA 202
46 – SOA Services 202
47 – Characteristics of Services 203
48 – SOA Rearchitecting 203
49 – Summary 203

P3L10 Guest Interview: LayerBlox 205
01 – Introduction 205
02 – LogicBlox 205
03 – Role 206
04 – Typical Application 207
05 – Motivation for LayerBlox 209
06 – LayerBlox 210
07 – Assembly Spec 211
08 – Components 212
09 – Interfaces 212
10 – Refinements 212
11 – Variants 213
12 – Product Lines 214
13 – Possible Limitations 215
14 – More on LayerBlox 216
15 – Implications Advice 216

P4L1 Components 219
01 – Bottom Up Design 219
02 – Components 219
03 – Buy vs Build 219
04 – Buy Quiz 219
05 – Buy Quiz Solution 219
06 – Build 220
07 – The Third Way 220
08 – Third Party Quiz 220
09 – Third Party Quiz Solution 220
10 – Characterizations of Components 220
11 – Component Life Cycle 221
12 – Component Models 221
13 – Component Models Quiz 221
14 – Component Models Quiz Solution 221
15 – Examples of Component Models 221
16 – Issues 222
17 – Issue 1 Configuration 222
18 – Issue 2 Versioning 222
19 – Versioning Strategy 222
20 – Automobile Components Quiz 223
21 – Automobile Components Quiz Solution 223

22 – Issue 3 Extensions 223
23 – Issue 4 Callbacks 223
24 – Invariants 223
25 – Callback Example 224
26 – Callbacks Quiz 224
27 – Callbacks Quiz Solution 224
28 – Callback Summary 224
29 – Issue 5 Contracts and Guarantees 224
30 – Level 1 Signature Contracts 224
31 – Level 2 Correctness Contract 225
32 – Level 3 Collaboration Contracts 225
33 – Level 4 Quality of Service Contracts 225
34 – Guarantees Quiz 225
35 – Guarantees Quiz Solution 225
36 – Summary of Contracts 225
37 – Issue 6 Objects as Components 226
38 – Object as Component Problems 226
39 – Inheritance Dangers 226
40 – Fragile Base Class Problem 226
41 – Issue 7 Industry Scaling 226
42 – Issue 8 Domain Standards 226
43 – Proprietary or Domain Quiz 227
44 – Proprietary or Domain Quiz Solution 227
45 – Component Framework 227
46 – Shared Attributes 227
47 – Comparison of Differences 227
48 – Comparison of Supported Variability 228
49 – Future Directions 228
50 – Summary 228

P4L2 Coffee Maker Exercise 229
01 – Status 229
02 – Robert Martins Coffee Maker 229
03 – The Mark IV Special Coffee Maker 229
04 – Hardware Quiz 229
05 – Hardware Quiz Solution 229
06 – Hardware 230
07 – Hardware Design 230
08 – Two Approaches 230
09 – Traditional Approach Quiz 230
10 – Traditional Approach Quiz Solution 230
11 – Class Model Diagram 231
12 – Limitations Quiz 231
13 – Limitations Quiz Solution 231
14 – Use Cases 232
15 – Brew Button Quiz 233
16 – Brew Button Quiz Solution 233
17 – Brew Button 233
18 – Collaboration Diagram 1 233

19 – Containment Vessel 234
20 – Collaboration Diagram 2 234
21 – Use Case Addition 234
22 – Brewing 234
23 – Collaboration Diagram 3 235
24 – Collaboration Quiz 235
25 – Collaboration Diagram 4 235
26 – Alternative to OOA 235
27 – Role Based Design 235
28 – Hardware API Quiz 235
29 – Hardware API Quiz Solution 236
30 – Dependency Inversion Principle 236
31 – Example 236
32 – Realization 236
33 – Abstract Classes 236
34 – Refinement 236
35 – Solution 237
36 – Summary 237

P4L3 Object Design 239
01 – From OOA to OOD 239
02 – OOD 239
03 – 1 Intermodel Consistency 239
04 – OOA to OOD Quiz 239
05 – OOA to OOD Quiz Solution 240
06 – 2 From Analysis to Design 240
07 – 3 System Design 240
08 – 4 Abstraction Mechanisms 241
09 – 5 Collaboration Based Design 241
10 – Object Design 241
11 – 1 Sources for Methods 242
12 – 2 New Classes 242
13 – 3 Generalization 242
14 – Generalization Example 243
15 – Generalization Advice 243
16 – Generalization Quiz 243
17 – Generalization Quiz Solution 243
18 – Implementing Generalization 244
19 – 4 Implementing Assocations 244
20 – One Way Assocations 244
21 – Pointers 245
22 – Two Way Associations 245
23 – Associations as Objects 245
24 – Tables for Assocations 245
25 – Associations Quiz 246
26 – Associations Quiz Solution 246
27 – 5 Implementing Dependencies 246
28 – 6 Implementing Control 246
29 – 7 Abstract Classes 247

30 – Modeling to Implementation Quiz 247
31 – Modeling to Implementation Quiz Solution 247
32 – Summary 247

P4L4 Design Patterns 249
01 – Design Experience 249
02 – Architectural Patterns 249
03 – The Gang of Four 249
04 – Definition 249
05 – The Composite Pattern 250
06 – Composite Classes 250
07 – Composite Client Class 250
08 – Composite Component Class 250
09 – Leafs and Composites 250
10 – Aggregation 250
11 – Textual Content 251
12 – Intent and Motivation 251
13 – Applicability and Structure 251
14 – Participants 251
15 – Collaborations 251
16 – Consequences 251
17 – Implementation Alternatives 252
18 – Patterns and UML Quiz 252
19 – Patterns and UML Quiz Solution 252
20 – Code, Uses, and Related Patterns 252
21 – Composite Pattern Quiz 253
22 – Composite Pattern Quiz Solution 253
23 – Categories 253
24 – Creational Patterns 253
25 – Example Creational Pattern Singleton 254
26 – Applicability and Structure 254
27 – Participants and Collaborations 254
28 – Consequences 254
29 – Implementation 254
30 – Implementation Issues 255
31 – Singleton Quiz 255
32 – Singleton Quiz Solution 255
33 – Structural Patterns 255
34 – Behavioral Patterns 256
35 – Catalog of Behavioral Patterns 256
36 – Visitor Pattern 256
37 – Vistor Pattern Description 256
38 – Visitor Applicability 257
39 – Structure 257
40 – Comments on Structure 257
41 – Visitor Participants 257
42 – Visitor Behavior 258
43 – Visitor Collaborations 258
44 – Visitor Consequences 258

45 – Visitor Implementation 258
46 – Pattern Quiz 1 259
47 – Pattern Quiz 1 Solution 259
48 – Pattern Quiz 2 259
49 – Pattern Quiz 2 Solution 259
50 – Pattern Quiz 3 259
51 – Pattern Quiz 3 Solution 259
52 – Problems with Patterns 259
53 – Problem Area Object Schizophrenia 260
54 – Problem Area Preplanning Problem 260
55 – Problem Area Traceability Problem 260
56 – Summary 260

P4L5 Design Principles 261
01 – Design Guidelines 261
02 – Design Quality 261
03 – Design Guidelines 261
04 – Coupling 261
05 – Cohesion 261
06 – Orthogonality 262
07 – Information Hiding Principle 262
08 – Foundational Concepts Quiz 262
09 – Foundational Concepts Quiz Solution 262
10 – Design Principles Catalog 262
11 – Liskov Substitution Principle 262
12 – Law of Demeter 263
13 – Hollywood Principle 263
14 – Dependency Inversion Principle 263
15 – Open Closed Principle 263
16 – Design Principle Quiz 263
17 – Design Principle Quiz Solution 264
18 – Interface Segregation Principle 264
19 – Reuse Equivalency Principles 264
20 – Common Closure Principle 264
21 – Dependency Structure Matrix 264
22 – Lattix Image 265
23 – Acyclic Dependency Principle 265
24 – Stability 265
25 – Bad Smells 266
26 – Design Heuristics Riel 266
27 – Single Choice Principle 267
28 – Transparency and Intentionality 267
29 – Transparency 267
30 – Intentionality 267
31 – Principles and Heuristics Quiz 267
32 – Principles and Heuristics Quiz Solution 267
33 – Summary 267

P4L6 Design Reviews 269

01 – Introduction 269
02 – Exercise Intro 269
03 – Defects Quiz 1 269
04 – Defects Quiz 2 269
05 – Defects Quiz 2 Solution 269
06 – Observations 270
07 – Reviews 270
08 – Step 1 Planning 270
09 – Step 2 Preperation 271
10 – Step 3 Review 271
11 – Step 4 Rework 271
12 – Step 5 Follow Up 271
13 – Roles 271
14 – Moderator Responsibilities 271
15 – Recorder 272
16 – Recording Form 272
17 – Severity Classification 272
18 – Reader 272
19 – Reviewers 273
20 – Review Meeting 273
21 – Thoroughness 273
22 – Metrics 274
23 – Process Data 274
24 – Alternative Review Styles 274
25 – Guidelines Participants 275
26 – Guidelines Content 275
27 – Guidelines Process 275
28 – Effectiveness 275
29 – Other Costs and Benefits 276
30 – Summary 276

P5L1 Geeks in Black: The Code Review 277
01 – Introduction 277
02 – Part 1 277
03 – Part 2 278
04 – Part 3 279
05 – Part 4 280
06 – Part 5 280
07 – Part 6 281
08 – Part 7 281
09 – Part 8 282
10 – Part 9 283
11 – Summary 284
12 – Credits 285

P1L1 Introduction

01 – Welcome

Hello and welcome to CS 6310, Software Architecture and Design. My name is Spencer Rugaber, and I will be your guide through this material. This lesson tells you about the course. What its subjective are? How it is structured? And what its policies are? A written description of the course can be found at the link in the instructor's notes for this lesson. But let's begin with some introductions, both of myself and of Jared Parks.

02 – Introductions

As I said, my name is Spencer Rugaber and I have been with Georgia Tech for 25 years. Before that I was an industrial software developer, both for big companies like Bell Laboratories in Spery, and for a startup called Interactive Systems. Since coming to Georgia Tech I have also worked closely with industry Both for the large telecommunications vendor, and with several startup companies. Throughout I have tried to combine academic software engineering theory with everyday industrial practice. I am the developer of CS 6310, and have taught it many times, both on site in Atlanta, and remotely in Korea, and France. I also participated in the planning and development of other software insuring courses here at Georgia Tech including the course on software process that is part of the ahms program. And other courses on software generation and software requirements. One thing that characterizes these courses, is their emphasis on learning by doing. Particularly for design, no amount of academic knowledge can make up for the hard won experience of actually building things. You will have that opportunity in the project portion of this course. You can have a look at my teaching philosophy by following the link in the instructor notes for this lesson. I now want you to meet Jarrod Parkes. You will see him in some of the lesson videos acting as a typical student, trying to solve the exercises and asking questions that arise. Jarrod, can you tell us a little about yourself. Absolutely, as Spencer said my name is Jarrod Parkes and I'll be working with you and Spencer throughout this course. In terms of software experience I've worked in a number of different places in my career. I've worked in a telecommunications company and there I tested and programmed new features for dsl routing equipment. I've also programmed a number of small video games and prototypes for the biotech community. And I own a web and mobile development company with my twin brother, James. More importantly, though, is I want to help Spencer bring a new, exciting feel to this course.

03 – Course

This course is about software design focusing on high level architectural design rather than detail design. It is not a programing course. In particular, you should already know how to program in at least one programing language in order to succeed in this course. It is also not an introductory software engineering course. A prerequisite of the course is that you have already been introduced to this subject, even by another course or because you have worked

as a developer in a software development organization. This course focuses on object-oriented design, and the projects make use of the Java programming language. Experience with these topics is not strictly required, but familiarity will help. If you want a refresher, the class resources page linked to from the instructor's notes points to video lectures I prepared on these topics.

04 – Objectives

This course, has specific learning objectives. It is intended by the end of the course, you will be able to accomplish the following tasks. Fist off, express the analysis and design of an application using the Unified Modeling Language, also called UML. Specify functional semantics of an application using the object constraint language, OCL. Specify and evaluate software architectures, in particular, select and use appropriate architectural styles. Understand and apply object-oriented design techniques. Select and use appropriate software design pattern. Finally, understand and participate in a design review.

05 – Course Structure

This course, software architecture and design, is divided into parts, each interleaving content lessons within class exercises. After a few introductory lessons we will begin by looking at the process of analyzing a design problem and specifying it using UML and OCL. Both the static and dynamic aspects of analysis will be presented. The next part of the course has to do with software architecture. This includes different ways of thinking about and representing architectures, the role of non functional requirements in determining architecture, and how architectural designs can be refined into modules that can then be implemented. The final part of the course gets into the specifics of software design, including object design, design patterns, design principles, and design reviews.

06 – Readings

This class does not have a required textbook, although several books that cover the material are listed on the class resources page. Instead, there are a set of reading assignments listed on the class schedule page. And the material itself is available via links on the class resources page. You are expected to have read this material before viewing the corresponding lesson.

07 – Assignments

In addition to the lessons, this course features a collection of assignments meant to give you an opportunity to explore material not directly covered in the lessons. These topics include architectural styles, design patterns, and middleware. There are also two exercises, one on OCL and one on state charts, intended to allow you to try these technologies for yourself. Finally, there's an opportunity to see how design is done in fields other than software in order to better appreciate how universal design is.

08 – Projects Design Studies

The heart and soul of this course are its projects. I can't really teach you to design. You have to learn it by doing it yourself. The best I can do is to introduce you to some specific techniques, such as modeling, and technology, such as UML, that you can use in thinking about how to solve design problems. The learning experience comes from applying them to a problem and getting a feel for how they work. A course on software design faces an important challenge. Interesting design typically occurs when confronting large complex problems but

those can't be dealt with within the constraints of a single course. I have taken a different approach to structuring projects called design studies. The idea here comes from a world of building architecture in which the architect often constructs a variety of small scale models in order to explore a design space. For each of the projects in this course, I ask you to solve it several different ways and then to systematically study the differences. The projects themselves are relatively small, and although some coding is involved, you will be working on teams with enough members that you should have enough resources to get the job done, and then study the variations. I've also tried to make the projects similar to those in the real world. The first similarity is that you will be working with team members that you are not colocated with. You will have to learn to quickly organize yourself to work in a parallel distributive fashion. The second similarity, is that the projects are some what open ended. That is, though they are basically requirements, there is room for you to be creative in your solutions, and to go beyond those requirements, if you believe you can significantly add value to the product you are producing. Of course, the danger with an open ended project is you might get engaged in polishing an extension and not have a working solution to the basic problem. I strongly suggest that you tackle these projects in an incremental manner, that is, begin by producing a trivial working program providing only minimal functionality then add one new feature at a time. Ensuring that the older functionality still works. The third similarity to the real world is that the projects are phased. What this means is that the three projects all deal with the same problem domain and each subsequent project builds on the solutions developed in the previous ones. In fact, you're specifically encouraged to make use of code. From your own or other teams that was developed earlier. This situation is typical of the real world, where successful projects often have long lifetimes in which the additional features are added and released to customers. The fourth similarity is that the members of your project team change for each phase. In the real world, turnover of teams is common and you have to learn to work with a variety of people. One specific benefit is that by having team members with different teams, you will have familiarity with and access to a variety of working solutions from which you can choose components from previous phases. More over, managing teams in this fashion provides a motivation for designing your solutions in a way that they can be reused on future projects. Even if you don't know exactly what those projects are.

09 – Policies

As you can imagine, anything as institutionalized as an online course has a strict set of policies to which the students and faculty must adhere. Of course, Georgia Tech has an honor code linked to in the instructor's notes that you should read and follow. Unless explicitly stated in the description of an assignment or project, all work you submit must be your own. The course makes use of the Piasa forum, which allows you to ask questions and otherwise communicate with your fellow students. You are encouraged to ask for clarifications and about the details of a particular tool or technology. But you should not ask for nor provide answers to questions that are an explicit part of the assignment or project. This course is officially administered using T-Square. All material will be turned in using T-Square according to the specific instructions of the assignment or project. Any changes of policy or due date will be announced on T-Square, and you are responsible for monitoring T-Square to obtain the latest information. The projects and some of the assignments require you to turn in written reports. The format of these documents should adhere to Georgia Tech standards as described on the page linked to from the instructor's notes.

10 – Grading

There are several means by which your performance will be evaluated for this course. First are quizzes that you will take while watching lessons on Udacity. These quizzes are not graded. But are included to evaluate your understanding of material as you follow along. In terms of graded evaluation, this course includes projects, assignments, class participation and an exam. The projects and some of the assignments are team based. And all members of the team will receive the same score on that submittal. That said, you will have an opportunity to comment on the contributions, or lack there, of your teammates. Those comments will become part of their class participation scores. The staff of this course, will make a strong effort to express the grading criteria for each deliverable, as explicitly as possible. Moreover, it is our intent to provide you accurate, prompt and informative feedback on your work. Should you have any questions about a grade you receive, please send an email to the head TA. Do not post a question about grades on Piazza. Any general announcements about grading, will be made by the staff using T-Square.

11 – Class Participation

As I mentioned earlier, I can't really teach you software design. You have to actively learn it. To encourage this, part of your grade for the course will be a measure of your participation in it. This is necessarily a subjective element, but I can let you know of some of the considerations that go into it. First is your teammates' evaluations of you on the projects and assignments. Also your engagement in the quizzes that are part of the lessons. Third is your non-gratuitous participation on the class form. I consider this course, itself, as a design project. Although it has been offered many times before, it has necessarily been adapted to deal with the constraints of web-based delivery and a large number of students. Therefore, I expect some elements will work better than others. I specifically would like to solicit your contributions, such as the following. First, any bugs in the material. These can be as trivial as typos, or as vague as questions about unclear lesson content. If you have any suggestions for additional content, as long as the suggestions fit within the general intent of the course. Also, specific examples, either generated by you or ones that you have seen that were particularly edifying. Alternative assignments or assignment questions. Even suggestions for style elements that might be added to the course. In general, the first contributor on a particular item will get primary credit for that contribution.

12 – Resources

As part of this course, you'll be given access to a virtual machine that contains an environment and tools that should aid your work for the course. For example, the virtual machine has ARGO UML, a UML-aware drawing tool. You are not required to use ARGO UML, but it may enable you to get started more quickly with the UML related work. There are other tools such as Eclipse installed as well. In addition to the virtual machine you should also be aware of the class resources page. It not only contains links to all the courses required readings, but other items referenced in the lessons, and some pointers. That should allow you to dig deeper into topics that interest you. I encourage you to explore. Also, if you are aware of other interesting resources, let us know on the class forum and we will add them to this page.

13 – Conclusion

Thanks for sticking with me through this introduction. I look forward to seeing your contributions and hope you find the course engaging and enlightening.

P1L2 Text Browser Exercise (Analysis)

01 – Introduction

Design is better learned than taught, this means that you actively try to solve design problems. And this course is structured around a set of exercises that you should actively work on as you are watching them. Let's jump right in and try to design a text browser application.

02 – Text Browser Exercise

Imagine the problem of browsing the text in a computer file. Imagine that no graphical user interface, GUI toolkit supplies a single widget to do this. Imagine that you would like to devise a cleanly structured solution.

03 – GUI Elements Quiz

Assume that you do have a GUI library, such as Swing or SWT, it just doesn't have a text browser widget in it. From what atomic GUI components would you build your TextBrowser?

04 – GUI Elements Solution

Okay so for the GUI components of this text browser, we'll need some kind of window to display the text, and I can foresee if the text is too large to fit in that window, we'll have a scroll bar that allows to, to move around in the text document.

05 – FileManager

It turns out that we also need a component that's going to supply us the text. Now, this isn't strictly in the GUI toolkit. But in order to make this thing work, we have to access the text somehow. So we'll call that the file manager component. We're going to make some assumptions. We're going to assume that you cannot hold the entire files contents in memory. You're going to have to go to the disk to get it. And assuming you have at the operating level line oriented access to the file. So in your, in your system libraries you have a way of, of reading the lines at anytime. So you're going to need to have a module, that when requested can retrieve a limited length, consecutive sequence of the file's lines. And we're also going to assume we don't have to worry about opening the file or closing the file. Just the reading of the file, supplying the lines.

06 – ViewPort

For your window component, we're going to call that a ViewPort, and you need to be able to use it, to display the textual content graphically. And we're going to make some assumptions here, we're going to assume that the ViewPort displays an inter, integer number of lines, and we're going to be, assume that it can be resized to be any length between one and a 100 lines. And we're going to assume that all the text in the same font, is in the same font and has

the same point size. So these are simplifying assumptions to make the, make this particular exercise, you know, small enough to fit in a lesson and also allow us to focus on just what the important issues are.

07 – ScrollBar

As far as the scroll bar is concerned, scroll bars are one graphical way of supplying numbers to other parts of an application. Were going to use a traditional scroll bar in which there's it's going to be a vertical scroll bar and it's going to have a movable part of it. That is the user can move a part called the handle which sits in a tray. So you can move up and down, and we when we use the terms handle and tray to indicate that we can set, the user can set the position in the file by by moving this handle up and down on the tray. The handle position denotes that part of the file that should be displayed in the view port. So when you move it all the way up, you get the start of the file, and when you move it all the way down, you get the end of the file. Also, the size of the handle in proportion to the size of the tray denotes the portion of the file that is visible. So if all of the file contents fit into the viewport, you'd expect the tray to be, filled up. And if we have a gigantic file, that in a very small window, we'd expect just a thin handle to appear in the scroll bar.

08 – Use Cases

So we've, we've come up with three candidate structural elements. Now, let's look at the behavioral side of this TextBrowser. One way to get a handle on behavior is to imagine how the user will use the intended application. We call these descriptions use cases.

09 – Use Cases Quiz

So, what use cases can you imagine for this particular text browser application? Can you list a few?

10 – Use Cases Quiz Solution

So, the first, as we just talked about, the scroll bar and the handle. The user could click on that handle and drag it up and down. So, even more basic than that. Why does this application exist? What is the user's main purpose for the application? Just to view the text that we're- So, our basic use case is read text, view text. And then, you've got one for moving a handle to see other parts of the file. What else? And then, the user could drag the view ports, can adjust the size of the frame. So, we can resize the view port. So, those are the three primary use cases for this particular application that we're going to consider.

11 – Analysis Model

Once you have a handle on the major elements and behaviors you can begin to construct an analysis model. We'll use the UML class-model diagram to express this analysis. A class-model diagram has rectangles for classes which we're going to use the term components to indicate the structural elements. We're going to use classes to, denote the components. Each rectangle is divided vertically into three compartments. One for the compartment name, at the top. One for its attributes, in the middle. And one for its operations, at the bottom. And then lines between the rectangles are going to denote relationships among the components.

12 – Classes Quiz

So to start the drawing of the model, why don't you come up with some classes to represent these three components and, and put them into a drawing?

13 – Classes Quiz Solution

All right. So, just start out with a rectangle for each component that we talked about. So we have the view port rectangle, the scroll bar rectangle, and the file manager. Okay, and in the topmost compartment of each, you've put the name that we've chosen for those particular components.

14 – Operations

So, the next thing we'll look at as far as the constructing the diagram is the operations. In the analysis model, operations comprise those actions that the users can undertake to interact with the text browser. The call that we're concerned here with an analysis model and not a design, okay? So, we are not trying to fill in the implementation methods that are involved. But rather, we're using the class model diagram to describe at a high level what the user can do. You can use the identified use cases to answer the questions, what internally visible operations does the text browser respond to? And, you recall what those were? So the first was just viewing the text. That doesn't seem to really elicit any kind of behavior though. So you're right, there's no event that the application has to respond to, to do that. Assuming that we started it up okay. Okay, and then moving the handle. Okay, so that's certainly going to be an operation the system has to respond to. And lastly, re-sizing the view port. And, we could also specify parameters for those particular operations like, what's the size of the view port that the user would like to see? Or, what's the position of the handle in the tray, when the user is using the scroll bar?

15 – Operations Quiz

So can you add those those two operations into your diagram?

16 – Operations Quiz Solution

Okay, so what do you have for the viewport? So for the viewport, we've added our resize operation and we've left some space for, you know, this new position that we'll be giving it. Okay and what remember we said that the viewport is an, can hold an integer number of lines of text, so it makes sense that we express the size in terms of the integer data type. Okay. What's the return type of this operation? I guess we could return the new verified size or like maybe a Boolean that it successfully executed. Or nothing at all. You know, in this case, we're performing the operation for its effect rather than its return value. Okay, so we'll use the UML void type to indicate that there's no return value of interest to us here. And I guess that I had a question in regards to, if we're going to list some types for the arguments and then also the return values, but without getting too implementation specific. We're going to use types that are UML-based and not language-based. You know integers for Java, for instance, might be expressed a certain way. It might be different than another language, right? So we're not, we're not, we're not concerned with implementation data types, however, the particular tool that you may use to actually draw this, might express the UML types in terms of programming language types. So for example Argo UML does make exactly that choice and map the UML types to, to Java types. But for purposes of doing an analysis model we're concerned with kind of the concepts and we're willing to abstract away those implementation details. Okay.

So how about the other use case in terms of moving the scroll bar handle? So that's added to our rectangle for the scroll bar and I indicated that there would be an argument for the new position of the handle. And it- And a, and a return type? So void two. Okay. Based on our last one. Now, conventionally the operations go in the lower, the, the lowest of the three compartments rather than, than the middle. But UML is actually flexible and you can have anything between one and number of, of boxes there and you can use them however you want. Your particular tool may, may differ. There are some subtleties here which ultimately we're going to have to, to deal with. The requirements didn't say only that the size of the viewport was an int. It said that that int must be between 1 and 100. So the UML diagramming notation doesn't allow us to express that, and we would have to use some other mechanism to to get at that particular detail. And also our GUI tool kit, when it's dealing with the scroll bar is, is probably going to return some kind of pixel position. But we're at, we're at the analysis stage and not the design stage, and so we're just going to, once again, assume that we could deal with, with numbers between between 1 and, 1 and 100 as the particular position of the scroll bar handle.

17 – Visible Attributes Quiz

This kind of, of approximation or obstruction is typical, particularly at the analysis stage, and you should feel comfortable with, with making these decisions so you can focus on the, the important details and avoid getting bogged down in some of the nuances of things. The third part of the class model is the attributes and what we'll do here is, as far as an analysis model is concerned is we are going to try to capture in attributes, those parts of the application which the user can actually see. And those are called percepts, and so we're going to try to understand what all the percepts are, and model each one of those as an attribute. And so in this case, can you think of, for the viewport, what is, what it's percepts are?

18 – Visible Attributes Quiz Solution

So we can see, at any given time, the handle in the tray, and where its position is. Okay. And we can also see the size of the view port as well. So let's go back. What's the user's purpose in using this application? Is via the text. So that's got to be a percept, right? Okay. That has to be something that's there. And there's one other one. It's a little bit more subtle. That was part of the statement of what the application does. And has to do with the scrollbar handle. Oh, okay. So I think I might know what you're talking about here. The size of the handle will have to relate to how much of the document we're currently seeing. So if we were ultimately going to implement this thing, we have to make sure that that particular percept was updated when we change to a file of a different size or we change the view port Window's size. So we have these four percepts that are going to correspond to attributes. And we can assign them to the particular components that we're modeling.

19 – FileManager

So when we do this, we have, with the ViewPort, we have its height as a percept and we have its contents a, as a percept. For the scroll bar, we have the position of the handle is a percept and also the size is a percept. But we don't currently have any percepts for the FileManager, and in fact the user doesn't directly see the file manager. However if we took our 40,000 foot view of the system, and, and we said what is external to the system and what is internal to the system. The user is external to the system. Users, is, is the one that's going to be taken advantage of the, of the system. But also, the file system itself, the operating system is external to the system and the operating system is the one with which the FileManager

component has to deal. So we're going to treat the operating system as an external acto,r and the FileManager is going to interact with that external actor. And as far as the FileManager is concerned, it has an attribute which is the document. Providing that as a, as a resource to the rest of the system, and it's, it's, it's It also has an interface to this external actor, that is, the actor has to provide that, that document. So we have an attribute there which is, the document which is a sequence of, a sequence of lines.

20 – Relationships

So we have so far, developed a diagram that has some classes or components, some operations, and, and attributes which correspond to the percepts. That's the easy part really in doing the analysis. The hard part is dealing with the relationships. These are the relationships among the components. In a UML analysis model, you should be concerned with three types of relationships. Associations, aggregations, and generalizations.

21 – Relationships Quiz

One way of getting at these relationships is to determine which components have responsibilities for handling the two user actions. The use cases and corresponding operations can provide answers to these questions. However, each of these events is just the first step in the text browser's response. For each of these two actions, determine what subsequent events you would expect to see. So if, so for example, if the user is resizing the window, or moving the scroll bar, not only do we expect the window size to be different, or the scoll bar position to vary, but we want this, the rest of the application to respond somehow. So, can you lay out what other things you would expect to happen?

22 – Relationships Quiz Solution

So for the Move ScrollBar action, we will see, or the other things that'll be involved is that the handle itself will move, but at the same time, we're going to need to get a new sequence of lines from our file manager. So let's imagine we're scrolling up, and we would expect to see some subsequent lines in the file appear at the bottom of the screen. How about the width of the scroll bar handle, or do we expect that to change? The width of the scroll- The height. No, I don't think so unless we're resizing the window. Okay, then how about going on to the resize. Right, and I guess that would be one of the first things that would have to happen, is when you resize the window, the size of your handle can change. It may also move the position of the handle. So like if you had the handle at the bottom of Viewport and you start bringing the Viewport up, then it's going to kind of gradually, I guess that's part of moving the view, or changing the Viewport size as well so these kinds of things are all working together. Right, okay. And we might need to see some additional lines of the file. Right, right. Ultimately, each of those responses represents a relationship between the corresponding components.

23 – Number of Lines Quiz

Let's start with the relationship between the viewport and the file manager. So at any particular moment of time, we have a, a number of lines in the file, and we have a number of lines displayed. What is the number of lines that are actually displayed in the Viewport as a function of the window size and the number of lines in the file?

24 – Number of Lines Quiz Solution

Wouldn't this also be dependent on the position of the handle as well? I just want to know the number of lines displayed. Okay, so the view port's height would give you the number of lines that you need. Well, not exactly, go on. So. In some cases. Assuming that the file is big enough, then whatever the size of your viewport is, say it's 100 at the max. Then you can get all 100 lines because there's going to be maybe some extra [CROSSTALK] Okay, so the maximum number of lines that you can display is the size of the viewport, if there's enough lines in the file. Right. Okay? And I guess conversely, you could have a situation where you have a plenty big enough view port, but your, you might have a one line file. So you'd only need the one line. Okay, so what is the number of lines displayed as a function of those two factors? Okay, so, given the size of the number of lines and the file, it would be, okay. I guess start with view port size, view port height. So view port height minus the- So as you can kind of guess here, it's a little tricky to come up with an answer to this on the fly and it's clearly not going to be the case that our UML diagram can't express it. And we're going to need some other mechanism to do that. Clearly the number of lines is limited by the size of the view port. You can't display more lines than will fit. It is also limited by the number of lines in the file. It can't display lines that aren't there. The actual number of lines that's displayed is the minimum of these two. Okay. Okay? And we can't express this in the diagram. We'll have to use some other mechanism and that mechanism is called OCL which is the Object Constraint Language. This is a part of UML that's a textual part that allows us to more precisely express various requirements that we have to deal with. And we're going to have to bring that into our model in order to deal with this particular situation. The fact that the number of lines shown depends on both the file size and the viewport size indicates that there's a relationship between these two components. We call this the line's visible association. We can show its existence graphically with a labeled line between the two classes.

25 – LinesVisible Association

It's in UML, it's an association. We can't, as I said, we can't express this entirely within the graphical notation. We'll use UCL to do this, OCL to do this. And later in the course we'll look more carefully at OCL. For now, here's what the relationship looks like. It says that as far as this particular association, the LinesVisible association, is concerned, there's, a fact or there's an invariant that must hold that the size of the viewports must be equal to the minimum of the size of the file manager and the size of the, of the viewport.

26 – Another Association Quiz

The lines visible association indicates that the contents of the viewport must come from the file manager, but it doesn't really say what lines. These, those lines are determined by the position of the scroll bar handle. See if you can state in English what this relationship must be. And, and here's a couple of hints for you. We already know how many lines, we just, we just got that. Right. And so if we can come up with the first line that's displayed, okay? Mm-hm. Then we can determine the rest of the lines that are displayed by just adding in the number of lines. Okay? Right. So how would you, how would you say this?

27 – Another Association Quiz Solution

Okay, so, I guess to preface this, I feel like in English I know what it means. Mathematically, I'm not- Let's just start with English. So we're going to start with the top of the handle to represent the first line in our sequence. Okay, you mean the position of the top of the handle?

Yes. Okay, with respect to the The tray. Huh. Okay. So if it's halfway down, you'd expect to be halfway down the file. Correct. Okay. And then, for the rest of the lines in the sequence how many more lines do we need is based on the handle size. We don't even have to be that complicated. Okay. We can just say the number of lines that fit in the view port. Okay. Right? So if the view port has 50 lines, we can expect to get the next 50 lines. Okay. Okay. Ultimately we would have to translate this into mathematics or express the mathematics on OCL but for now our expectation is that that percentage of the way down in the tray indicates the percentage of the way down in the file.

28 – Explanation

Determining the top line, top visible line and the number of lines together tells us which lines will be displayed. This determination is based on the attributes of the viewport, its size, the file manager, its document size, and the scrollbar, the handle positions. Hence there is an association, a three way association, amongst all three components. You can also call that a ternary, as opposed to binary, association. Let's call this association displays, as in the view port displays the contents provided by the file manager and determined by the scroll bar

29 – Displays Diagram

In UML you can use a a diamond to indicate associations that have more than two participants. Here there's three participants. It's the display's association, and we've added in the OCL that gets down to this particular mathematical details as to what, what actually gets shown here.

30 – Handle Association Quiz

The final property we must describe has to do with the size of the handle in the scroll bar. So what is this particular association as to the size of the handle and it's association with the, with the other components?

31 – Handle Association Quiz Solution

I tried to put a little mathematics into this to try to understand it. So, say we have a document that's 1,000 lines long. And our viewport is 100 lines high, high. Then our handle would need to be one-tenth of the tray as we scroll. So the size is dependent on that relationship between, how many lines can currently be displayed by the viewport, and how many lines in total does the document have? Kay, sounds, sounds good. The size of the scroll bar handle with respect to the size of the scroll bar tray indicates the portion of the document's lines provided by the file manager that are currently visible in the viewport. We're going to call this the handle proportion association, and it's also a ternary association.

32 – HandleProportion

Here's what it, what it looks like, and the OCL is provided as well.

33 – Subtleties

However, there are some gotcha's here. What happens, for example, if the length of the document is zero? All of sudden, you have divided by zero. Right. Okay. Right. So we would have to consider that as a special case. Also, what exactly do we mean by the size of the handle? As we said, we probably from the GUI toolkit would get some pixels, a number of pixels. But we'd need to translate that into something like a percentage, or an integer count

of things. And then there's an even more subtle situation. Let's say that we've scrolled to the end of the file. Okay. And we re-sized the window to make it bigger. Okay. What happens? What do we expect to see in the viewport? So one possibility is we would see exactly the same lines, but now some blank lines at the bottom. Okay. Another possibility is that the line that was at the bottom of the viewport before stays at the bottom of the viewport, and we see some more lines at the top. And, in fact, if you go play around with actual web browsers out there, and with word processing editors and so on, you can see both of these behaviors. However, if we're designing the application, we have to make the decision about which one of those two behaviors we actually intend that the text browser to have. Okay? So, this process of modeling has forced us into thinking about something which we might not have, otherwise, thought about, which is one of the benefits of doing the modeling. It forces you to think through subtleties of things.

34 – Summary

This exercise has illustrated the construction of an analysis model for the text browser problem. We haven't yet begun to solve the problem, which is what design is all about. To begin thinking about design ask yourself, what is the key design question with which any implementation of the text browser must deal? We're not going to answer that question right now, but we will come back to this a little while later, and maybe by that time you will have thought through what it would take to actually to do a design here.

P1L3 Design Concepts

01 – Design Concepts

Good day, class. Today's concept is design concepts. And design is everywhere. From super complex manufactured artifacts like the International Space Station, to the dinner party you were planning for next week. In the course, although we are going to be specifically concerned with software design, many of the concepts of design in general are going to play a role, so we'd like to get into that. Let's start with a few definitions.

02 – Terms Quiz

For this quiz, I've listed four different terms, along with their definitions. The terms are overlapping, and include design, engineering, craft, and art. See if you can connect, the definition to the term that's being defined.

03 – Terms Quiz Solution

What is design? Design is deliberative, purposive planning. As far as software is concerned, this translates into solving some problem. Given in a, as a set of requirements. What is engineering? Engineering adds in the element of science and mathematics. And we're going to see the role of formal methods play in software design. What is craft? Craft is some sort of skilled occupation. Those skills come from long experience, and we'll see that with software design, the more experience you have on a particular kind of problem the better. And what is art? Art is the conscious use of skills. Taste and creative imagination in the production of aesthetic objects. And later in this lesson, we'll see that, certain design principles and aesthetic principles can play a role in software design.

04 – Programming or Design Quiz

Another quiz for you. Think for a minute about the difference between software design and programming.

05 – Programming or Design Quiz Solution

Here are two important differences. One difference is scale. When you're doing software design, iit's probably because you are dealing with a big problem that's going to have a big solution. If you're doing programming, you're more likely concerned with programs that are going to end up being 100 or 1,000, or maybe even 10,000 lines. The International Space Station has 30 million lines of code. Another key difference between programming and design, software design, is the role that non-functional requirements play. For programs, you may at a, occasionally be concerned with performance but by and large, non-functional requirements don't play a big role. With software design, the whole story is how you are going to deal with trade-offs among non-functional requirements.

06 – Software Design

What is software design? Software design is the process of building a program while satisfying the program's functional requirements and not violating any of its non-functional constraints. This is going to turn out to be a question of trade offs. How do you trade off between performance and resource consumption, for example? Software design is normally broken into two different phases, architectural design and detail design. Architectural design is the process of carving up the programs into components and assigning responsibilities for aspects of behavior to each component and talking about how the components are going to interact with each other. We're going to spend a great deal of time in this course talking about architectural design. Detail design is the process of dealing with the individual components. Particularly, with respect to their data structures and their algorithms. Let's look for a minute at some of the aspects of detail design.

07 – Design Notation

Here's a quote from Tony Wasserman about detail design. The primary activity during detail design, is designing the data structures that are there and by implication the algorithms that are going to work on those data structures. As far as those algorithms are concerned, sometimes you may wish to represent them using some kind of design notation. Here are a few that have been used in the past in which you may have little familiarity. Their pseudo code, which is like writing a programming language algorithm without the programming language. There's structured programming, which is a set of control structures, which allow you to organize the algorithm into sequences, conditions, repetition, and chunking in the form of calling subprocedures. Flow charts and call graphs are graphic representations of programs, that may be useful in helping you understand how that program is going to, going to work. And in some cases decision tables, which are lists of rules and the conditions under which those rules are going to apply, can be useful in helping to understand complex situations.

08 – Weather Quiz

Here's a little quiz for you that deals with detail design. Imagine that you were writing a program to predict the weather. The way that these programs normally work is by taking some geographical area and carving it up into a rectangular grid or mesh. That is, there are numerous cells and each cell contains some data such as temperature, wind pressure, humidity, and so on. And then running an algorithm which diffuses the information from cells to their neighbors in order to come to some conclusion about what the future weather will be. If you had to develop a weather prediction program you might have the choice between using arrays or objects.

09 – Weather Quiz Solution

The main reason for choosing arrays is performance. Arrays have been part of programming languages since Fortran in the 1950s. And those programming languages have been tuned to take advantage of the hardware architecture available in order to do array computations very rapidly. Objects on the other hand are a little slower, but they're much more flexible. If, for example, your weather program changed from having a rectangular grid to one where there's different kinds of shapes adjacent to each other, having an object oriented representation may allow you to deal with that situation more flexibly.

10 – Approaches to Software Design

There are many approaches to software design. Some espouse a particular point of view as to how best to structure a system, such as object orientated design. Some of them are intended for a particular class of application. That is the design of real time systems. And some of them are structured to deal with only a part of an application, such as user interface design. All approaches to design however, include three aspects that may be compared, the design method, the design representation, and how that design is going to be validated. Let's first look at design method. A method is a systematic series of steps by which you undertake to do your design and solve your problem. Typically, a design method suggests a particular way of viewing the problem. With object oriented design, we view the problem in terms of a set of cooperating objects. Only later do we assign the services or functions that each of those objects are going to be able to provide to the system. Other methods that we may be mentioning during the course of the term include structure design, and role based design. The design method that is chosen acts as the discipline for the participants, the designers and ultimately the implementers, forcing them how to organize their thoughts and and activities in certain ways.

11 – Issues with Design

There are, however, some issues with design methods. You as a architect or designer have to make some choice. Are you going to go do things top down, bottom up, inside out? There are a variety of choices there. Are you going to begin by thinking of the procedures and functions? Or are you going to begin by thinking in terms of the nouns and objects like you would with objectory development. A topic which we'll come back to later in the lesson is the issue of conceptual integrity versus cooperative development. An important decision in many, software development shops is the trade off or the tension between doing a design that takes a little bit more time. In order to save yourself effort and money in the long term, by supporting maintainable and general structures. Or are you going to be dominated by short term delivery schedule. And finally, is the role of tools. What, what tools are you going to use in terms of your particular design.

12 – Design Review Quiz

So imagine in your shop that you have a design method, and you've, you've chosen a design representation, and you've done a design. The result is some artifact expressed in the design notation. Now typically, these days, that representation is reviewed by a team, that is, there's some validation the design in fact meets it, the system's requirements. The question for this particular quiz is, why bother with the validation now if you're going to build the program and have tests, many of which may be automated, to check it for you?

13 – Design Review Quiz Solution

The key reason of course is that the earlier you find problems, the less expensive it is to fix them. Particularly if you've got a design problem and you don't detect it until you're about to deliver to the customers, it can be quite expensive to fix.

14 – Design Validation

The third important aspect of approaches to design, is how they are validated. As I just said, typically that means some kind of review, walk through, inspection by a team. It could also be the case that the tools that you're using, to represent the design can do some checking

for you. Some issues arise with design validation. And a key one is the independence of the validators. The problem here is that if you have the design team, doing its own validation. They may be blind to particular issues. If they didn't think about them when they were doing the design, they may not think about them when they're inspecting the design. Bringing in independent val, validators can help with the effectiveness of the design review. Second issue that arises is the, dependence of the design validation on the design method. For structured design, there's a complete set of rules associating metrics with each of the design artifacts. On the class resource page, there's some guidelines that I've written up concerning the things that you can ask about during a object oriented design review. A third key issue with validation is when do you do it? One strategy is to do it as you go along. That is, on a daily or weekly basis, review what you have and make adjustments. An alternative is to wait until you get to ma, major milestones, have design reviews and make your changes at that point.

15 – Other Design Issues

In addition to design methods, representations and validations, there are some other issues that arise with software design. We already talked about the difference between architectural and detail design and exactly where that boundary is. Of key importance is the respective energies we put into designing the functional part of the system versus dealing with those non-functional constraints. At a more abstract level, there's the difference between what and how. The specification process deals with what the system will do. The design process begins to say how we're going to do it. Finding the right boundary between those two is a key issue. And finally, what is there about your particular application that is going to affect the design process? For example, you have some existing resources that you want to reuse and build in your solution. They can affect the design that you do.

16 – Design Documentation

The next key concept to consider is design documentation. If we're talking about the software design of large systems, the systems are likely to be complex and the scale and complexity beg for having good design documentation. If you've invested all that energy in developing the system, it's likely that that system is going to around for a while and is going to be under maintenance, maybe by people that were different than the original designers. And having some form of written communication can be a big help. Different kinds of methods, different kinds of applications, require different kinds of documentation. Those may range from formal, multi-volume documents, to scribbled notes or, or slides in, that are used for presentations

17 – Documentation Quiz

Here's a brief quiz for you. Think of organizations that are doing software development. Pick a typical organization that would require a lot of formal documentation.

18 – Documentation Quiz Solution

One example of organizations that require lots of, of detailed documentation are military contracting organizations. On the other hand, if you're in a small research lab, and you're doing exploratory development, you may not need a lot of documentation, because it would only get out of date very rapidly

19 – Traditional Design Documentation

Traditionally design documentation has included information about the components you've carved the system up into, what their responsibilities are, what their primary data flows are and so on. Other elements of traditional documentation include, how you going to deal with performance considerations. And resource consumption, by resource here we might mean memory, we might mean use of peripherals, bandwidth and so on. It that's not enough for you, if your organization needs more detail documentation, you might rely on some IEEE standards, such as standard 1016. Some of the key elements that the standard add to the list that is traditionally used are things like who is the designer? It might be nice to know if you have to go back for a question, who was responsible for a particular piece of the design. What are the dependencies among the elements? Are there hidden assumptions that one component is making about other components? What are the tradeoffs among the non-functional constraints? How did you decide to take a particular tradeoff? What assumptions are you making about your users, about the technology that will be available for the hardware, and about the changing customer base? And which particular. Views of the software system as your documentation providing.

20 – Leonardo Objects

An even more elaborate approach to design information was taken by the Leonardo Project at the MCC in the 1980s. They devoted, a whole project to determining what is a suitable set of design information, and some of the, elements that they came up with that go beyond these, we've talked about already are. Explicit lists of the stakeholders involved. Okay. Most important is what issues were raised during the course of the design. And for those issues, what were the possible resolutions and why. Was a particular choice made? That is, design decisions and the reasons for making them. Leonardo also stressed various relationships among the design artifacts. such as versions. In producing this system you actually maybe producing several versions. Like the professional version and the free version and so on. And what exactly is in each version, and what design compromises had to be made in order to accommodate multiple versions. There is also the questions of revisions. And the time, the historical progress of the design. What went into each of the, the revisions along the way. Specific. Descriptions of constraints, upon the solution and how they're being dealt with. important, and what groupings or aggregates of, of design, artifacts implementation, artifacts configuration files, packaging, components and so on, did you decide to use, as far as your solution is concerned.

21 – Design Rationale

Taken together, a lot of this design information, can be thought of as design rationale. Rationale here means, the reasons that you did what you did in coming up with your design solution. The more you can make explicit choices with reasons for those choices, the better off we'll be, the downstream people who are trying to maintain the system. The bottom line as far as design information is concerned, is that there's many options to you. And you need to decide upfront, what it is that's going to be important in your documentation, and then capture it as you go along. Now I'd like to introduce you to some key design concepts that are going to be used throughout the term, when we talk about software design.

22 – Coupling and Cohesion

First, let's look at conceptual integrity, which I mentioned earlier in today's lesson. Let me give you two historical quotes that, that give an idea of what conceptual integrity is all about.

The first is from the philosopher Descartes. He said of these thoughts on the very first that occurred to me was, that there is seldom so much perfection in works composed of many separate parts,. Upon which different hands had been employed, as in those completed by a single master. More recently, Fred Brooks, in, The Mythical Man-Month, has said much the same thing. I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit. Certain anomalous features and improvements, but to reflect one set of design ideas, then to have one that contains many independent, and uncoordinated ideas. A couple of related concepts are coupling and cohesion. These originally came out of structure design, but they also apply to object orientated design, and other design approaches. Assuming you've carved your system into separate components, those components may be coupled to each other. Coupling is the extent to which two components depend on each other for successful execution. If you think about it for a minute low coupling is good. After you've delivered your system, and you have to maintain it. If you have a highly coupled system, and you change one module, that means that you're likely to have to change other modules. Whereas if there's low coupling, that likelihood goes down. A related concept is cohesion. With cohesion we're talking about a single module, and cohesion is the extent to which that module, or component has a single purpose or function. High cohesion is good. For example highly cohesive modules are much more easy to reuse. They have a single purpose. You need to reuse them to accomplish that purpose.

23 – Java Quiz 1

Here's a two part quiz for you having to do with coupling and cohesion, and the Java language. Which of the two possibilities reduced coupling, or increased cohesion, is Java's package designed help with? Reduced coupling or increased cohesion?

24 – Java Quiz 1 Solution

Packages are for reducing coupling. A package encapsulates a set of names and requires the programmer to explicitly import those names in order to get access to them. So a module can't get access and depend upon the names in another module unless it's been explicitly imported.

25 – Java Quiz 2

Second part of the quiz. How about Java's class inheritance mechanism, does that decrease coupling of increase coupling between the parent and child classes?

26 – Java Quiz 2 Solution

It actually increases coupling. That is the child knows about and depends upon, the details in the parent. This can be a problem, if you then later change the parent.

27 – Information Hiding

Next concept is information hiding, developed by David Parnas. And it has to do with encapsulating the capabilities that a particular module has behind an abstract interface. After all, if the rest of the world that's going to make use of that module only knows the abstract interface. It gives you freedom later to change the implementation details without breaking all the client programs. One key example of information hiding is if you're dealing with a system that has access to many hardware devices, hiding that access to the devices behind an abstract interface. See if you can come up with some other good examples of places in the system where you might like to hide information behind abstract interfaces. Some typical examples include

access to a database or some server some place, the specifics of an algorithm or how you're implementing, a data structure.

28 – Abstraction and Refinement

Now another pair of concepts. Abstraction and refinement. All design methods support these ideas. After all, we're dealing with large systems and the only way that we can wrap our mind around those large systems is to think about them in terms of abstract concepts, and then how we're going to refine each of those abstract concepts into lower level implementations. Programming languages, design techniques typically provide some conceptual mechanisms for dealing with abstraction. Here are a few, for example, the whole process of specification where we're dealing with the what, abstracts away all of the details of how we're going to solve the problem. Programming languages is typically have various aggregation abstraction such as arrays and structs and records and objects, that allow you to, if you wish, avoid the details of what all the features of those aggregations are. Obviously in object oriented languages, the whole idea of the class hierarchy and generalization allows you to abstract away from all the special cases. Even a fundamental thing like the, the parameters to procedure calls are function calls. Allow you to abstract away from what all the various possible calls to those functions are by specifying names for the parameters rather than all of the specific arguments. And finally, non-determinism, at least at the specification level, where you can avoid giving details of exactly how you're going to implement something by specifying that you in certain circumstances, you don't care.

29 – Aesthetics

Now let's come back to aesthetics. I mentioned this at the start, and I wanted to see how some two, in particular, two classical authors have kind of captured the notion of aesthetics and what relationship that might have to software design. The first is Aquinas. I won't quote for you the, the Latin but what it, what it boils down to is that beauty, elegance, resolve to wholeness, harmony and radiance. And we can think of that as far as software is concerned as completeness, consistency and conceptual integrity. Other quote comes from Pascal, you may have heard of the Pascal programming language. Pascal was a French mathematician, and one of my favorite quotes is he said, he apologized in a letter saying, sorry I didn't have I would have written you a shorter letter, but I didn't have time. If you think about it for a minute, what this means is it takes a lot of time and energy to come up with an elegant solution that looks quite simple on the surface, but really satisfies the complex requirements.

30 – Design Philosophy

Finally, I'd like to finish this lesson with talking about philosophy a little bit. And these insights come from the Danish Design researcher Piete En, and he is relating the process of software design to the thinking of four important philosophers. So, first philosophers, Descartes. We think about Descartes with analytic geometry. And this may translate for us into thinking about the analysis phase of software design. On the other hand, Marx is very concerned with social processes and classes. And understanding the social context of design, maybe even involving the users in the design process. Martin Heidegger, was concerned, among other things, with tools. And of course, tools play a big role in the automation of our development process and tools like IDEs in our active development environments and case tools, computer aided software engineering tools of course play a big role. And finally, my favorite is Wittgenstein, the Austrian philosopher, who came up with the concept of language games. And, what this means, is the inventing of a vocabulary, that helps you think about a particular problem. For

example, think about the introduction of personal computers and the role that thinking, the role of the terminology of the desktop, folders, trash baskets, and so on, play in doing that.

31 – Metaphors Quiz

Can you think of some other metaphors that are important to us in, in dealing with computers place your answers into the text box

32 – Metaphors Quiz Solution

Here are a few that come to mind. Client server organizations for systems, icons, firewalls. There are many more.

33 – Summary

Tying up this whole lesson, the important thing is that design is the most creative part of the software development process. Consequently, overall system quality is highly dependent on the designs produced. A key determinate of design quality is the extent of experience of this, on similar projects in particular, of the members of the design team.

P2L01 Review of UML

01 – Diagrams

Doing design you need to be able to represent those designs and a very popular way of doing that is with diagrams. This course focused on UML, using UML and constructing those diagrams and the purpose of this lesson is to review for you some of the different kinds of diagrams that are there. There's lots of diagrams, we're not going to be using them all. But I'd like you to be at least familiar with what ones are there. So lets step back a little bit in history. Diagrams have been a popular part of software development from the beginning. I still have a flow chart template that you use for writing out the control logic of an algorithm. Over the years, various sophisticated diagramming techniques have developed. Okay. Among those are SADT, Jackson Design, Structure Design and so on. Each with their own particular approach to diagramming. In the 1980s, object oriented tech, techniques began to emerge, and with them. Various techniques for diagramming [UNKNOWN] solutions. The first one we want to look at is called OMT.

02 – OMT

OMT was developed at General Electric corporation by James Rumbaugh and others that were there. It resulted in a book, a cover of a book as a, a die on which there are three visible faces. And one face has a class model diagram, one face has a state chart diagram, and one face has a data flow diagram. The class model gave you the structural aspects, the state chart gave you the behavioral aspects, and the data flow diagram gave you the functional aspects. That is OMT was a way of putting together three views of a system into a cohesive view. As OMT was coming along other competitive, object-oriented methods and diagrams were also being developed.

03 – UML

This led to the natural development where there was a goal of trying to unify these various techniques together, which led to the Unified Modeling Language, UML. There's a, an industry group called the Object Management Group, called OMG. That is the home for this standardization effort. It's where you can find the specification documents, which I referred to you refer you to during the term. UML has also been championed by IBM. They acquired a company called Rational, which had developed a development environment called Rational Rose, which supported directly UML. There's lots of such case tools now. Aside from Rational Rose, one that we've used is Enterprise Architect. These are commercial tools. There are also all kinds of drawing tools out there, which you can get for free. UML, itself, had three main architects. One was Rumbaugh, one was Grady Booch, and the other was Ivar Jacobson. Okay. They're sometimes called the three amigos. They produced, in addition to a unified modeling language itself, three books, a reference manual, a user's guide and so on. That had its main author as each one of the three, and then together. Conveyed the overall approach to UML.

It's important to note at this state the UML diagrams can be used both for design but also for analysis. I'll try to distinguish as we go forward whether we're talking about an analysis diagram or a design diagram. The main distinction is, analysis is concerned with the problem being solved and design is concerned with the solution to that problem. I've taken some of these diagrams you're about to see from the UML reference manual and from the UML users guide.

04 – Diagram Types

We're now into version two of UML. Which has as a total 14 different kinds of diagrams. We're going to go over those quickly here but in the course we're not going to be using all of them. We're going to concentrate on just three or four of them. And we'll have special lessons devoted to those but I wanted you to at least be aware of the different kinds of diagrams that are there. That said, on a particular project, you're unlikely to use all the diagrams. You're going to use the ones appropriate to the particular project. In particular, determine what is the trickiest part of the project and use the diagram that helps you best understand that tricky part. The two main categories of diagrams are structural and behavioral. Structural diagrams give you the pieces of the system that are always there and the relationships among them. Behavioral diagrams on the other hand, are concerned with the executions of the system. And the particular diagram may only convey one execution. That is, you may have to have multiple sequence diagrams in order to get a good feel for all of the behaviors of the system

05 – Diagram Quiz

Imagine you're in a development shop. And this development shop is old school and don't know anything about diagrams. And you have to convince them of the value of having design diagrams.

06 – Diagram Quiz Solution

One major use of diagrams is that you can use them to convey information to others in the group. The others might be your teammates, but they also might be people on other parts of the project which you don't normally communicate with. They might even be persons down the line time-wise. That is, the maintainers who are going to deal with the system five years from now. Communication is the first answer. Second, we're talking about, in the case of UML object-oriented development, and object oriented development has a particular process often used in association with it. That is, the diagramming technique can support a process that you're using. For example, your process is likely to have some kind of validation step associated with it, maybe a design review, and using a particular design diagram can help you structure that review. Also, by using a particular kind of diagram, there, you may be able to find some tool support for that diagram. The tool support might inform the user when a violation of the visual syntax occurs, or inform the user that a piece is missing. As far as the disadvantages are concerned, just like source code, any time that you build a diagram, you have to worry about it getting it out of date, that is, keeping it up to date with respect to changes in the rest of the system.

07 – Class Models

Let's start with the most popular diagram. The Class Model Diagram. This is also sometimes called the static model or the class structure diagram and it is an example of a structured diagram. It is showing the structure of the system. In particular, it has classes and the relationships among those classes. And there are numerous embellishments. Class Model

Diagrams have many, many different affordances on them, icons on them and so on that you can use. And we'll be devoting a lesson later to looking at those, and, how, what, what the meaning of those particular affordances are

08 – UML Classes

As a quick reminder, UML classes are depicted as having up to three compartments, separated by horizontal lines. The top compartment, typically, has the name of the class in it. The middle compartment has the attributes of that class; the instance variables. And the bottom compartment has the methods or operations that class provides.

09 – UML Relationships

As far as relationships are concerned, there's three main categories of relationships in UML. There are dependencies, depicted by dashed lines with an arrowhead, indicating that the class at one end uses the class as the other end. The solid lines without arrowheads are associations. And that says that class at one end affects or has a instance of the class at the other end. The solid line can be adorned with a diamond. The diamond is used indicate this has a or aggregation embellishment to the association. Third main category relationship is the generalization relationship. The class at one end is a kind of a class at the other end. In this case there is a solid line but it has that triangle at the end of it.

10 – Example Class Diagram

Here's a typical class diagram. It has some examples and associations. The associations don't have labels on them. There's also couple of places where there are generalizations. As I said, the class model diagrams can have a lot more adornments to them and we'll see those later on.

11 – Object Diagram

Related visually to the class model diagram is the object diagram. In fact they're the same with one major exception. Instead of mentioning classes, they mention instances. The label compartment at the top of the boxes has an underlined text line. A text line has two parts. One is the class name, just like in the class model diagram, but it also has the name of a specific instance, and those two are separated by a colon. So for example here, we have the company class, and in particular we have the c instance of a company class. Optic diagrams are obviously used to convey the specific use of the classes involved in a class model diagram. As you can see, for this particular instances here, many of the attribute fields have attribute values filled in for them, as they would be for a particular instance.

12 – Composite Structure Diagram

A less popular, less frequently used structural diagram is the composite structure diagram. This one is used for showing the internal structure of a class. Of particular interest to us are its interfaces. So on the left side of the interior class here are two horizontal lines coming out. The top one with the circle on the edge of it is a Provides Interface. That's saying that this class provides some capabilities to the rest of the world. Under it is another line coming out and this one with a semicircle that's open. This is a Requires Interface, that is, what does this class require from the rest of the world? You can then imagine having a variety of classes that plug into each other. That is, a provides from one class plugs into a requires from another class. This is one way of putting together the pieces of a software architecture.

13 – Component Diagram

In fact, that's exactly what a component diagram does. It's a static implementation view of how the components of a system fit together. As far as UML is concerned, a, a component is a physical, replaceable part of a system that packages implementation and conforms to and provides a realization of the set of interfaces. It's usually used to model code entities such as binaries, okay, that might perhaps come from a library. And relationships in the diagram are meant intended to specify that one of the components uses the services of another component. This particular type of diagram can also be used to convey architecture.

14 – Example Component Diagram

Here's an example component diagram. The rectangles with the two sub-rectangles on their side indicate components. This is one of Bouche's contributions to UML. He had a diagramming type called Bouchegrams in which these particular icons were used. The stick figures represent in this case people or actors of the system, and the dashed line indicates where components plug into other components.

15 – Deployment Diagram

If we're talking about complex systems, these systems may run on different processing units. And we'd like to convey the configuration of the run-time processing units, and their component instances in a way, that sees how they can interact. And this is, included inside a UML deployment diagram. A node in the diagram will correspond to a computational device, and the arcs indicate some kind of communication.

16 – Example Deployment Diagram

In this example, there's two major processing units indicated by the shadowed rectangles. Inside rectangles are some components, and then there are lines indicating the communications between the physical components, but also have interfaces plugged into each other.

17 – Packages

UML also supports packages, in the sense of Java packages. These are general purpose organizing mechanisms. Before UML 2, you could use packages as parts of other diagrams. In UML 2 there was a separate package diagram. Essentially this is providing namespace scoping so that each package can have its own set of names without worrying about collisions. And that there's dependency arrows between two packages if some piece of one package has a dependency arrow with some piece of another package. That is, it's an abstraction of that particular dependency at, to the package level.

18 – Example Class Diagram with Packages

Here's a use of packages in UML 1.5. In general, the indication that something is a package is it had a little tab in its upper left-hand corner, with a label on it. You see in this particular example there are also interpackage dependencies. The dashed line ending in an arrowhead.

19 – Example Package Diagram

In UML 2.0 there's a separate package diagram, but it's conveying the same kinds of information.

20 – Profile Diagram

The final structural type of UML diagram that I'd like to mention is the Profile Diagram. But this requires taking a step back. UML itself is a language, that has various pieces to it, such as classes and associations and so on. Those pieces. Essentially provide a, describe a system and therefore you could have a UML description of UML. That UML description or UML is called the UML meta model and in fact you can have, a UML class model diagram of a UML meta model. Even, more abstract is the fact that you, as a user, a designer, can extend the UML middle model. You can add new kinds of icons. You can give, special labels. To particular elements in the model. You do that extension in what's called a UML profile. And there's a UML Profile Diagram in which you can convey it

21 – Example Profile Diagram

So here are three UML profile diagrams. Notice that above the class names are some stereotypes. Those are the things in the double angle brackets. And the particular stereotypes here are metaclass and stereotype. So we're talking at the meta level. There's also some inheritance going on here. The overall purpose of these particular examples has to do with extending the UML language to deal with describing certain kinds of systems.

22 – UML Structure Diagram Quiz

Okay. Well, here is a quiz for you to test your knowledge of these different diagram types. In column one I have the names of the diagrams, in column two I have some definitions, and I'd like you to match them together. As a review that particular diagram types of the class diagram, the composite structure, the component diagram, deployment diagram, object diagram, package diagram, and profile diagram. And then the definitions are in the right column. The static structure at a particular time. The organization of physical software components. Three is the logical groupings and dependencies. Four is the components and structural properties. Five extensions to the UML metal model. Six internal structure and possible interactions. And the seventh choice is the physical system resources and how they map to the hardware. Try to match those together.

23 – UML Structure Diagram Quiz Solution

Well the class model diagram is the components and their structural properties. The composite structure diagram is the internal structure and the possible interactions among them. The component diagram is the organization of the physical software components in the system. Deployment diagram, that's the physical system resources and how they map the hardware. The object diagram, that's the static structure at a particular point in time. The package diagram, well, that's the one with the logical groupings and dependencies. And finally, the profile diagram, that's extensions to the UML meta model. Notice that there's a lot of overlap among these terms, which say, that the different diagram types have a lot of overlapping themselves.

24 – UML Quiz

Another quick quiz for you on this material is, which of these particular UML structural diagram types could be used to convey system architecture?

25 – UML Quiz Solution

Well, the component diagram, deployment diagram, package diagram, and class diagram, more likely so, some of the other ones less likely so.

26 – Behavior Diagrams

The second main category of UML, Diagram Systems Behavioral Diagrams. In contrast with the structural diagrams, which describe the system as a whole, the behavioral diagrams are concerned with a particular instance of behavior of that system. That is, you may have to have multiple sequence diagrams, multiple collaboration diagrams, to convey, to give an idea of overall system behavior. We're going to now survey these so you get a feel for what's available to you. Once again, it's unlikely that for a given system you'll use all of these diagram types.

27 – Use Case Diagram

Let's start with use case diagrams. UML does not include OMT's data flow diagram. Instead, it includes Jacobson's use case diagrams. A use case is a sequence of user-visible actions along with system responses. It's a story of how the system deals with a particular user interaction. Use cases are particularly useful for eliciting requirements. You lay out different stories of how the system is going to be used, and then explore ramifications. What happens if something goes wrong, what are some intermediate steps that maybe you didn't make explicit.

28 – Use Case Diagrams

Use case diagrams will have two major icons. One are some stick figures and these denote external actors. Typically, these are system users, but they may also stand for other systems or external devices. In the use case diagram, ovals are use cases. That is, this is, a use case diagram is not a system story. It's a description of the set of system stories. Lines in the diagram without annotations indicate participation. That means that the actor at one end is involved in the use case oval at the other end. There're two annotations available in use case diagrams. One is the extends annotation and the other is the uses annotation. Extends mean that you have one story and you'd like to extend it by some other contingencies, essentially getting two for the price of one. The uses stereotype is like a subroutine or, or function call. That is, a common piece of behavior that might used by several other use cases.

29 – Example of Use Case Diagrams

Here's a Use Case diagram. It's got six ovals. It's got four actors. Three of which are human. And one of which is a separate system. There's annotations on through the lines. One situation involves reusing a particular use case and more than one other use case. And the other is an extension situation.

30 – Individual Use Cases

The use case diagram lays out the set of use cases. But, what's an individual use case? Well, it's a story, and the story can appear as unstructured text or in a tabular form. The unstructured text might tell the story about an individual user named Foster who wants to buy something at Amazon. Foster goes to the Amazon web site, Foster browses until he finds the particular item that he wants, he adds it to the shopping cart, he then goes to the check-out page, he provides information about his billing, and then submits a purchase request.

31 – Tabular Version of Example

As an alternative formulation of the same story, we could have a table. And typically the table will have three columns. One column is the agent or actor involved in that particular step of the story, and this is going to be Foster. And it's going to be the Amazon web server. The second column indicates the action that's taking place. This might be the user action it might be the system response and the third column, contains information about any object, that, is conveyed, in that particular step. So it might be for example credit card information. Same information in the unstructured version and the tabular version. They're both examples of use cases, and they would represent the content of one of the ovals used in this diagram.

32 – Context Diagrams

The top level dataflow diagram is called the context diagram. The context diagram has a single oval. Which is the system as a whole. There are rectangles there that indicate the system actors. And then the lines indicate the flow of data between them.

33 – Example Context Diagrams

Here's an example context diagram for a system that plays chess with the user. The rectangle is the human player, the oval is the chess-playing program, and there are three lines between them. One line indicates that the human player is supplying a move to the chess-playing program. Another line from the chess playing program back to the user is the computer's move. And the third line indicates that the computer can also put out a diagram describing the board.

34 – Sequence Diagram

Another one of the most popular UML behavior diagrams is the sequence diagram. This can be used to convey a single use case.

35 – Example Sequence Diagram

The sequence diagram has columns corresponding to individual participants, usually objects, in the system. Time marches down the sequence diagram, and horizontal lines between columns indicate the passing of a message from one object to another object. Historically, these sequence diagrams evolved from message sequence charts which had been used in the telephony industry for many years. These diagrams will be semantically equivalent to communication diagrams which we'll look at in a minute, but from a slightly different point of view.

36 – Communication Diagram

An alternative view of a use case to that provided by a sequence diagram, would be a communication diagram. In the communication diagram, it looks like a class model diagram. That is, there are rectangles corresponding to classes, and there's lines between them. However, in this case the lines correspond to instances of communication, likely operation calls.

37 – Example Communication Diagram

Here's the same information that we saw in the sequence diagram. There are three particular objects. One is a client, one is a transaction and one is a proxy. There are lines between these particular objects. Notice that the lines are annotated with numbered message indicators. The numbers indicate the orders in which those messages take place. And it's using a

kind of, Dewey Decimal notation. So, first step is number 1, then number 2. And then 2.1, 2.2 and then step 3.

38 – Activity Diagram

The sequence diagram and the communication diagram that we've seen aren't particularly designed to deal with synchronization. UML has a separate diagram, called an activity diagram, designed for this purpose. In this diagram it's a variant of a state machine in which. Multiple states may be simultaneously active. That is have their own threads of control. This activity diagrams are derived from petri nets. Petri net diagrams that have been around for many years. In the diagrams trans, transitions are typically triggered by activity completion. That is you finished with one state. Rather than by external events. You can use these diagrams to model workflows, process synchronization, and concurrency.

39 – Example Activity Diagram

Here's an activity diagram you can think of it executing as follows. Imagine that you had some token that you could lay on top of any of the states on the diagram. It would come in at the start at the top where there's the filled in circle, and it would move along the horizontal line to get to the first state. And then it would move downward to the diamond. At which point it would split. That is, we'd have two tokens. One going over to the right and one going downward. The one on the right can continue downward again and finally coming into the diamond near the bottom. The second token from the top goes straight downward and is thwarted by the horizontal, the heavy, black horizontal bar that's there. This is a synchronization point. In this case there's nothing to synchronize with, but there are two lines coming out of the bottom. Those two lines will themselves both have a copy of the token on them, one will go over to the left into the two activities that are there, the second will go straight downward, and eventually those two paths will merge into the second horizontal line which is a synchronization point. You can think of those two paths, each having their own tokens, as running independently, and the horizontal bar at the bottom being a kind of a gate which only opens with both tokens have arrived from the top, hence synchronizing those activities. At the point at which the gate opens the two tokens are merged together, the single token goes out of the bottom into the diamond, and the diamond is essentially a joint point, which, once again, combines the two tokens and proceeds on then to the last state, and the final state at the very bottom of the screen.

40 – Interaction Overview Diagram

Now I want to mention two less frequently used UML diagram types. One is the interaction overview diagram. It's a kind of activity diagram, where the nodes in the diagram correspond to lower level interaction diagrams which could be of any sort, sequence, communication, interaction overview, and timing diagrams. In the diagram example, the term ref denotes a specific interaction occurrence.

41 – Timing Diagram

Here's a example of the UML timing diagram. If your familiar with the design of digital chips, it should look very similar. In digital chips obviously you're worried about electrical signals arriving at a certain time in response of the silicon and germanium in that chip, how long it's going to take to happen. If you need, in fact, to diagram out specific timing of situations in your system you can use the UML timing diagram to do that. Time marches from left to right and arrows indicate the places where timing has to be synchronized.

42 – State Diagrams

Final behavioral diagram I'd like to mention is the state diagram. This is the most powerful, the most complex of the behavioral diagrams. They're also sometimes called state charts. These diagrams convey extended finite state machines extended with the ability to represent aggregation, concurrency, history, broadcasting events and so on. We're going to devote a whole lesson to them, but let me give you one example diagram here.

43 – Example State Machine Diagrams

In this diagram there are two external states. There's an idle state and a maintenance state. And there's transitions between the two indicating the lines at the top of the screen. The maintenance state itself has sub-state machines separated by the horizontal dash line. These two machines, one called testing and one called commanding, are running concurrently. Each of them as a simple machine starting at, at its left in the initial state and moving towards right. And the commanding machine has a back loop from its command state to its rating state indicating that, that particular machine can execute several times before finally getting to its final state.

44 – Behavior Diagram Quiz

Here's a quiz for you on the behavioral diagrams. Once again, I'd like you to match between the diagram type and its definition. As far as diagram types are concerned there on the left, there's the Activity Diagram, the Sequence Diagram, Communication Diagram, Interaction Overview Diagram, Timing Diagram, Use Case Diagram, and State Diagram. Definitions, number 1, system functionality provided to external actors. Possibility 2, dynamic behavior in response to stimuli. 3 is flow of control from activity to activity. 4, synthesis of lower-level Activity Diagrams. 5, interaction of classes in terms of message exchanges. The next one is object interaction in terms of numbered messages. And the last one is a rotated sequence diagram.

45 – Behavior Diagram Quiz Solution

Activity diagram. Well that's the one that's, of course, there's a flow of control from activity to activity. As far as the sequence diagram, that's an interaction of classes in terms of message exchanges. Communication diagram. That's object interaction in terms of numbered messages. Interaction overview. Synthesis of lower level behavioral diagrams. Timing diagram. Well that's the rotated sequence diagram. That's also going to have specific time laid out on it. The use case diagram, well that system functionality provided to its external actors. And the state diagram, that's dynamic behavior in response to stimuli.

46 – Object Constraint Language

Well, those are the diagrams types. I just wanted to repeat that no particular system that you develop is going to use all of them. And, in fact, you're probably going to concentrate on the most popular ones. But they're there in case you need them. I also want to mention two other features of UML that don't involve diagrams. [COUGH] One is the Object Constraint Language and the other one, I hinted tthat a few minutes ago, the Metamodel. The Object Constraint Language, and we'll devote a whole lesson to this, is a textual extension to UML's vis, visual notation. Its purpose is to provide a more precise specification, to be able to specify things which you can't specify in the diagrams themselves. You can use this textural extension as annotations to class model diagrams, and statechart diagrams. Essentially, the

Object Constraint Language, this first-order predicate logic, plus the ability to navigate around the diagrams and some collection classes, like sets and, and bags and sequences. The overall purpose of the Constraint Language, Object Constraint Language, is to be very precise, if you need to, in the specifications of your system.

47 – Example OCL

Here's an example of OCL. In this case, we're talking about using the OCL as an extension to a class model diagram. In the class model diagram, there's an account class. The account class has a deposit operation. That operation takes in a real number called amount. The pre keyword indicates the precondition, in this case, that the amount being provided in this deposit is greater than zero. The postcondition, indicated by the post keyword is indicating what must be true after the execution of this particular operation. In particular, the balance afterwards must be equal to the balance before plus the amount that was deposited.

48 – UML MetaModel

The second non textual part of UML to be aware of is the Metamodel. I mentioned this before, it's UML defined in terms of UML. The UML Metamodel, is a UML description of the UML language. More over, you can extend the Metamodel. You as a designer can extend the Metamodel. Using UML profiles and we saw the profile diagram. Those extensions are more stereotypes, more tag values and you can add the constraints. These are constraints on the diagrams now, not on the models

49 – Class Model of UML MetaModel

Here's an example of a UML class model diagram describing UML. You notice there are classes for class, there's also class for attribute, there's a class for association, and there are associations among these classes. You should study this diagram, to make sure that you understand how the various pieces that comprise UML [UNKNOWN] together.

50 – Summary

Bottomline is you can't design an complex system with having, without having some idea of what it's supposed to do. That is what problems is it trying to solve. Diagrams can help you express that understanding and express your solution to that problem. UML provides a wealth of diagram types for you as well as OCL and the meta model. In general, the more precisely you understand problem the fewer subsequent problems you will have with that system's history.

P2L02 Object Oriented Analysis Exercise

01 – Analysis

As you know, this is a class about design. And the question is, how do you get started doing a design? Well, before you can solve a problem, you need to understand it. And the process for understanding a problem is called analysis. In this lesson, we will be concerned with a specific type of analysis called, Object-Oriented analysis, or OOA.

02 – Object Oriented Analysis OOA

OOA is a requirements analysis technique developed by Abbott and Booch in the 1980s. It concentrates on modeling real-world objects based on their descriptions in natural language to produce an object analysis model. We will express these models primarily using UML class model diagrams. Object oriented analysis pays primary attention to the objects with which a problem's concerned. Prior to OOA, on the other hand, the predominant method of analysis is analysis and design. With structure design, also called functional decomposition. In contrast with OOA, functional decomposition was concerned with the functions that the solution, that the solution was required to provide rather than the objects.

03 – Object Quiz

To get you started thinking about this, I'd like you to, for a moment, reflect on why objects might be a better starting point for analysis than functions. Choose among the following four possible reasons. First, functions are provided by all programming languages, whereas many languages don't include objects. Second choice, during maintenance functions change more frequently than objects. Third, functions are too mathematical. And fourth, objects are a more modern technique. Which of those do you think is the best answer?

04 – Object Quiz Solution

Well, you're right about that. Software systems have long lifetimes. During which they undergo evolutionary change. Unless, the responsible developers are careful. The original structure of the program may be lost as features are added, bugs fixed and libraries migrated. For example, consider a banking application. Over time, the particular functions such as interest computation, fees and tax rules may change. All of these are functions. However, the need to represent accounts, account holders and transactions. All of which are objects, will continue. That is objects are more stable than functions, over the lifetime of a system. Hence, designing a software around them, leads to more sustainable designs.

05 – Object Oriented Analysis and Design

Structured analysis and design techniques are functionally oriented. They concentrate on the computations that need to be made. The data upon which the functions operate are secondary to the functions themselves. Object Oriented Analysis and Design on the other hand

is primarily concerned with the data objects. These are defined first in terms of their attributes and data types. Later the functions are defined and associated with a specific objects.

06 – OOA

Well how does OOA work? First it takes a textual description of a system to be built, such as a requirements document. And looks for different kinds of words such as, nouns, verbs and adjectives. It's goal is to use the identified words to built up descriptions of classes and their relationships. Nouns will correspond to classes. Action verbs to operations. Adjectives to attributes. And stative verbs to relationships. The resulting class model can be reviewed with a customer for accuracy and completeness. Here is an overview of the steps involved

07 – Steps in OOA

First off, candidate object classes are indicated by the occurrence of nouns in the natural language description of the system to be built. The nouns can then be organized into related groups termed classes. The next step looks for adjectives, which often indicate properties that can be modeled as attributes. Subsequently, action verbs can be modeled as operations and assigned to the appropriate provider class. Other stative verbs are indi, indicative of relationships among the classes.

08 – Technique

More specifically, the actual OOA technique to apply is the following. First off, obtain or prepare a textual description of a problem. Underline in your textual description all the nouns. Organize the nouns into groups to become candidate classes, then underline all the adjectives. Assign the adjectives as attributes of the candidate classes. Then, underline the verbs, differentiating action verbs from stative verbs. Assign the action verbs as operations of classes, and assign the stative verbs as attributes of classes or relationships. Sounds simple, but we will also see that there are some stumbling blocks along the way, which we'll have to apply our own thinking to, to solve appropriately.

09 – Step 1 Locate Nouns Quiz

So let's see what happens when we try an example on this. I'm going to give you a paragraph describing a small program for counting the number of leaves on a tree. A tree is a data structure consisting of nodes connected by directed arcs, in which occurs a single root node from which all other nodes descend. Nodes without outgoing arcs are called leaves, and the intent of the program is to count the number of leaves in a given tree. Here's the paragraph description. So counting tree nodes, keep a pile of the parts of the tree that have not yet been counted, initially get a tree and put it on an, the empty pile. The count of leaves is initially set to zero. As long as the pile is not empty, repeatedly take a tree off the pile and examine it. If the tree consists of a single leaf, then increment the leaf counter and throw away that tree. If the tree is not a single leaf but instead consists of two subtrees, split the tree into its left and right subtrees and put them back on the pile. Once the pile is empty, display the count of the leaves. Okay, step one, go through the description and underline all the nouns, see what you get.

10 – Step 1 Locate Nouns Quiz Solution

Read through what you've got as far as the nouns are concerned. Sure. Pile, parts, tree, tree again, pile, count, leaves, zero. Most of these are repeats. Subtrees, leaf and leaves so. Yes, I think those are my main nouns. Didn't you underline them in there? I did. That one I was kind of wary of just because of it didn't seem to be something that would be descriptive enough to maybe give us an object later. It's just, what's the word? I think you're right there. Pronouns are going to refer to something else that's already in the text, so you don't worry about pronouns.

11 – Issues

So here are some of the issues that arise when we try to accomplish the first step in OOA. As Jared mentioned, some of the words are duplicated. You know, we'll try to condense those and just have one copy of each of the words. Some words share the same stem, for example, pile and piles. And some words are, are close to each other and really share the same underlying concept like leaf and leaves. And in these cases we're going to do what's called stemming. Stemming removes the prefixes and the post fixes, the suffixes to the words, and just uses the root word as the corresponding candidate and class. That leaves us with a question that you, you, you indicated both counter and count, okay? In this case, we're going to use our kind of discretion and say, let's for the, the moment treat those as separate classes, even though by getting rid of the, the suffix on counter, we get the same word as count.

12 – Step 2 Candidate Classes Quiz

So, now I'd like you to take this list of nouns that you have with the condensed into various groups, and try to propose out of that list some candidate classes. And then write down the candidates that you have. Okay.

13 – Step 2 Candidate Classes Quiz Solution

So what did you get? I wrote pile, tree, node, and counter. Okay, so clearly there's an important concept of pile. And tree is important, counter is important. What did you do about leaf? Leaf to me seems like it's just a special kind of tree. It's just a distinction for a tree that has no other children as a leaf. Okay, a little while later in OOA, we will look at the situation where one of the potential classes is a kind of another one of the classes. But for now let's include leaf as a separate, separate class. How about part? So part is connected to, I feel like I have a need to go back and see what concept that's connected to. Is that part of the pile, part of the tree? So the parts of a tree already I believe are nodes. So to me, I guess I'm substituting the word node for part. Well a part of a tree could be a subtree as well, right? It's a collection of nodes. So the suggestion is at this early stage in the process, To err on the side of including extra classes which we can then remove later. Okay So I would suggest having part as a candidate class as this stage. And then the other question I have is you threw the word node in there, which isn't in the original text. Now there's nothing wrong with using your judgement on adding classes in there, but I was wondering what you were thinking as you did that? Node I've seen use plenty of times for tree to constitute a part of a tree, a graph, a connected graph, and it seems in my mind that using node here might be a way to make it generic enough that it could grow later. It's a concept that's general enough to trees and grass. I think that's my premise. So we'll leave it in for now and we'll see what happens. In general, as analysts you're going to have to use that kind of judgement. To add in concepts that the original description may have either assumed or didn't think through at all. And one of the

values of doing an approach like this is that it will point out problems, and inconsistencies and incompleteness in the documents. So I would say go with your intuition at this stage. One other noun that you didn't say something about is zero, okay? So what was your thinking about not including zero as a potential classer? Zero to me sounds like a description of a tree or an attribute to a tree, it doesn't seem to represent its own, in a sense, an object. It's just a state of an object, for instance. Okay. Can you pick one of the classes here which zero might gravitate toward? Counter. Sure. So the counter's going to count something, and one of the possible things it's going to count, is it going to have values, and one of the values it might have is zero. So I think you're right at this stage of saying let's not worry about zero as being a class. Okay. But we can't forget about it either. Okay? It's going to be part of our model eventually.

14 – Initial Class Model Diagram

An object oriented analysis the above groupings that you've produced from candidate classes. In object oriented analysis a class is a description of a group of related objects, these objects are also sometimes called instances. We're going to use UML class model diagrams. And in those diagrams, classes are represented as rectangles, possible vertically partitioned into sub-compartments. Thus, taking what you have, we can come up with some initial class model diagram that has rectangles for counters and leafs and piles and parts and trees. And in your case in your case node.

15 – Caveats

Note that like in any analysis process, the conclusions that we reach are always tentative. As we engage in the process, we learn more about the problem, which may lead to revisions of our analysis. In fact, it was one of the early lessons of software engineering, is that requirements documents are always wrong in the sense that they're incomplete, or inconsistent, or they don't truly reflect what it is that the customer ultimately wants. And as an analyst it's your job to elicit that correct description. And OOA can help you do that. Questions that arise during OOA may require research on your part, or even going back to the customers. Often, the customers may not have even considered, what you, what you bring up as a question. And will thank you for realizing that there was more to the problem than what they originally had in my mind. Thus, the overall process of analysis is inherently incremental, hopefully leading to a joint understanding between the analyst and customer so that the design and implementation can proceed

16 – Step 3 Adjectives Quiz

So let's move on to the second step. We have some candidate classes, okay, and we're now going to look at the features of those classes. And in object-oriented analysis, there are two candidate, two kinds of features. One kind are attributes and these are going to correspond to adjectives, and the other kind is operations, which correspond to the action verbs in the text. So for the next step, I'd like you to go back to your textual list and this time, underline the adjectives, along with the corresponding noun, indicate the corresponding noun that the adjective modifies. And then by underlining these phrases and, and then combine them as you did with the nouns, we'll have candidates as candidate attributes for the various classes. Take a crack at that now.

17 – Step 3 Adjectives Quiz Solution

So tell me what you got. Oh, the adjectives that I found in the original paragraph statement were empty. Empty is used three times. So empty, single, leaf, left and right, and two. Okay. Those six. Okay. So presumably it's an empty pile, so empty goes with the pile concept. What is single a modifier of? Single is a modifier of leaf. Okay, and two? Also, no two is subtrees. Okay, and left and right are also subtrees. Now what is leaf an abject modifier of? Counter, so we have a leaf counter. Okay, so let's examine these possibilities as candidates for attributes of the classes.

18 – Adjective Issues

Long is, is normally an adjective of some sorts, but that doesn't mean that every adjective is going to necessarily contribute to this list of attributes. So, it's fine to have one and then discard it as not being relevant. Okay? You had single leaf, you have leaf counter. We have this one set modify sub trees. There is another, call it a trick, another way of getting modifiers that aren't directly, that don't correctly come from adjectives, and that is if you've got a prepositional phrase, such as parts of the tree, or count of the leaves, you could think of tree parts. Okay? And leave counts and so on. So once again, the, the, the simple rule of saying adjectives doesn't quite get you all the, all the way where you want to go. So, some of the issues that arise when you, when you try this technique. Okay? As we said, parts of the tree doesn't appear. In, in the natural language to be an adjective. However we still want to recognize that the word part signifies an attribute of tree. Same for count of leaves. As you indicated the use of the word long even though it's, nominally a, an adjective doesn't really contribute here in the sense of being a modifier for one of these classes. The phrase single leaf, if we think about it for a minute, has to do with the count of the leaves. This is a one, of a, of a potential different number of counts that leaves could have. That is, there's a count attribute that has a value of one. Okay, single being, single meaning one. Similar to the phrase two sub trees can be interpreted as a count of the number of a sub trees, in the tree class. Having a value of two

19 – Updated Class Model

Okay. Okay? And similarly, we can do the same thing with Part, thinking of part of the tree being an attribute, of the tree. The tree class now has attributes for the left sub tree, the right sub tree, for leaves and for parts. The Pile class, will have some kind of emptiness attribute associated with it. And we initially, we can say that's maybe a Boolean and has a value of true or false depending upon whether it's empty. And we can add all these into the diagram and we'll, we'll see, now that we've added a second compartment. Below the top compartment which had the label the class, the second compartment, has the list of these attributes.

20 – Step 4 Operations

Now let's look for candidate operations. An operation is a computational service provided by an object. In OOA, candidate operations are suggested by looking for verbs, particularly action verbs. In addition to action verbs there are other kinds of verbs including linking verbs. Which are typically associated with the word is, they're more likely to be descriptive, hence related to attributes rather than to actions. Another category contains the stative verbs, typically descriptions of situations rather than of objects. And they're going to be indicative of relationships among the objects

21 – Action Verbs Quiz

Well let's take one more crack at your, paragraph. And this time, go and look for action verbs and underline them.

22 – Action Verbs Quiz Solution

What did you come up with? So I'll just go in order, been or have not yet been, get, put, a couple of is', take, examine, consists, increment, throw, throw away, split, put them back and display. Okay, first off, what happened to keep, the very first word? Oh, I missed it. I guess I was assuming. Most sentences start with nouns. I probably just over looked it. Okay, so keep is clearly a verb. And then you had been, or has been. But really that's just part of the construction of has been counted. It's counting that is the verb there. You might want to add that to your list, replace been with count. So in the same sense that we dealt with nouns by stemming, we might stem here as well so the verb is count, and counting is just one form of that. And then you had get and put [COUGH]. Is, on the surface, is a linking verb but it was part of a phrase that says is set and set is a verb. And I don't see you having set in your list. Okay. So we want to add set in as a verb here. And you've got take, you've got examine, you've got consists. Do you have increment? Yes, okay, throw or throw away. You've got split and you've got display. Okay, you've got put twice and just like before, we're going to combine those together so that we only have a single occurrence of that. This is a list of candidate operations and of course we're going to want to associate them with the class that they belong to. And you have to use your knowledge of the English language in order to determine what that is. But if you start with, did you write down keep there? I did not write down keep. You might want to put that down. Okay.

23 – Operations in Class Quiz

24 – Operations in Class Quiz Solution

So what class would you think keep is associated with? Who's doing the keeping? The pile is keeping parts. The pile? Well, it's keeping a pile. Okay. One way of answering the question is who's doing the keeping here? Okay, and in fact, its the system. Its the solution that we're building that's doing the keeping. In general, every system that you design, you can think of the system itself as a class, or an object in a class. But usually you don't explicitly represent that system class in your class model. Okay, so in this case, what the system is keeping is a pile, and so we're not going to worry about keeping an operation of one of the constituent classes. It's an operation of the overall system, okay? And now count, what is it that is doing the counting? Counter is doing the counting of leaves. Okay. And in the text, what does it say about counted, in that first line at the end? Have not been counted. So- So, the parts have not yet been counted. So, the operation is counting parts. We can think of parts as being an argument to the count's operation, and the counter, as Jared said, is going to be doing the counting. How about get and put? We'll lump those together. This seems similar to the system is getting and putting- Well- We're getting trees. Once again, the trees are the arguments to the operation. Getting and putting or taking them off of the pile. So we're going to get them from the pile and put them on the pile. So in this case, we're going to associate those operations with the pile class. Okay, how about is set to? Is set to. Leaves, count of the leaves is set to 0. This seems to belong to the counter. You're right on there. Okay? Take off is similar to get and put, or taking it off of the pile. How about examine? Take a tree off the pile. The pile could be examine it. It seems like the pile and the system are this overarching thing, like they're the ones driving this operation. Well, the pile is going to be a part of this

system, but then again, so is the counter. And the counter is not the pile. It sounds like the counter's actions though are usually on behalf of an operation maybe that started with what the pile was doing, which is examining parts of trees. Okay. And at this stage, we're doing a static analysis. The interplay of the classes is more behavioral analysis, which would come later. Okay, so you're right that that's an important element of understanding the intended system. But as far as this initial analysis, we're going to do it and look for the structural elements that are involved. Okay, here should be an easy one. How about increment? Counter is increment. Okay, so we need an operation to increment the counter. How about throw away, or throw? Increment the leaf counter and throw away the tree. So the counter increments and then throws away the tree? So it could be the counter that's throwing it away. Well, a synonym for throw might be delete. Okay, and we're deleting it from the pile. Correct. So there might be an operation in the pile class for deleting or throwing away the element. Okay, how about splitting? Splitting? The pile splits trees into trees and sub-trees? I don't think that the pile does any splitting. The pile's looking at the split sub-trees. That's correct. Or storing them. Correct. Who is doing that? The system? The- So it's an operation, your splitting. Okay, who would you send instructions to actually to do the split? The tree. The tree. Okay, so we are going to have an operation on a tree. It says split thyself and getting out two results from that. How about display? Display, the counter's displaying its count. Sure. Okay, so you can see the back and forth in trying to understand these things that goes on. All the process we saw before of combining words together, stemming and now trying to associate the various operations with the classes.

25 – Operations in Classes

If we now try to summarize what we have so far, we have a pile class and its got operations. We can put a tree onto the pile. We can get a tree from the pile. We can take a tree off of the pile. For the counter we have increment and display. And for the tree class we have split and throw away. Now, a minute ago we talked about throwing away from the pile, and sometimes these things can go back and forth. For the moment we are going to associate with the tree class. In our class model diagram we added a new compartment at the bottom of the rectangle to hold the operations. In general, we could also at this time list of various arguments that go arguments to the operations, what their types are, what is the type of the return value, or we could hold off that process later. What we are trying to do is get a feel for what the elements of the problem are, and how, how, how they are represented in terms of attributes, and then what services the various the various classes can have in terms of operations.

26 – Operation Issues

Some of the issues that arise from doing this, we talked about keeping a pile, and it's really the system as a whole that keeps the pile and we're not going to explicitly represent the system ourselves. There was the phrase that we had, has been counted, and really has been counted is more a description of a state then an action verb. Okay, so we could say count is an action verb but has been counted as more a stative, or stative situations, so we're going to revisit that when we get to looking at relationships. And the phrases is set to, examine, and consists of. Well if you think about it those are really an expression of what is the case. So, when you examine something, you find out what is the case there. If you examine a counter, you get back the count. These are really ultimately going to be represented in our implementation with some kind of eq, equality check. And so, we're going to assume that all the classes that we have eventually come out with are going to have an equality operator associated with them. So we're not going to explicitly model those at this time.

27 – Step 5 Relationships

Step five in the object or in analysis process has to do with relationships. The main elements in the class model diagram are going to correspond classes and relationships. And we're going to depict the classes ash compartmentized, rectangles and relationships are going to be indicated by lines connecting the rectangles. And there are going to be three different kinds of relationships that we'll look at. One kind is generalization, and that's indicated by a line at, and at one end of the line will be a little open triangle. The second kind is called aggregation. This is going to be used for situations like what we mentioned parts of, and in this case, the line ends with an open diamond, and then if we don't have any adornment on the line at all, that's going to be a general association. So let's look at each of those three categories of relationships and see if we can find examples of them in our, our example problem.

28 – Generalizations

The first kind of relationship we look at is generalization. This relationship between two classes indicates that an instance of one of the two classes, the child class are, are the instances of one of the two classes are a kind of instance of the other class, called the parent class. This means that the instances of the child class are the subset of the instances of the parent class. In our text, words like kind of and type of, indicate a generalization relationship. Even if these words aren't explicit in the text, the class names, themselves, can serve as indicators. For example, cars are a sub class of vehicles.

29 – Generalization Quiz

So implicit in our text, it's not there explicitly is a generalization relationship. Can you, can you guess what it is? You kind of men, you kind of mentioned this earlier on

30 – Generalization Quiz Solution

So, let's add to our diagram a a rectangle for leaf and make it a specialization of the tree class. That is, tree is a generalization of leaf. And we indicate that by a line between the two. And there's a little open triangle at the, at the tree end of the relationship. What that meant was, that we took the leaf attribute outside, that was inside the tree class. And made it now, a separate, separate class

31 – Aggregations

The second kind of relationship to look for are aggregations. Aggregations are some kinds of collection, or set of things. An aggregation is heralded in text by words like consists of, part of, contains, has, incorporates or belongs to.

32 – Aggregation Quiz

Can you think in our example of instances in the description of leaf counting, where aggregation relationships might be indicated?

33 – Aggregation Quiz Solution

So during leaf counting we talk about a tree having or consisting of a left and right sub-tree. Sounds right, anything else? There's parts of a pile which is tree as parts so that's. Okay. kind of that left and right sub-tree kind of going back to that. So you've got it. In our diagram now, we're going to have an aggregation relationship between pile and tree. This means there's a line between the two and at the pile end of the line, is this open diamond indicating that

a pile is a collection of trees. Likewise if we have sub-trees being parts of trees, we have an aggregation relationship between a tree and itself. Okay, so we'll have a line, a looped line going from tree to itself, and at one end of that line, we'll have the diamond indicating that tree is a recursive data structure, as you might expect, okay? Consisting of parts which are trees.

34 – Associations

Generalizations and aggregations are two specific structural relationships, between classes. More general, is the idea of association. For example, went to school at, is an association between the university class and the student class. Stative verbs, denote a state of being. For example, the house sits on top of a hill. Stative verbs, often indicate associations. In class models, associations are indicated by lines connecting the associated classes. There are no special adornments on the ends, but the line is usually labeled with the name, of the association.

35 – Associations Quiz

In the tree counting example, there are no explicit associations indicated. However, there is an implied association. Can you determine what it might be, and what classes it associates?

36 – Associations Quiz Solution

Okay, so there's and association between the counter and the leaves, okay. And we indicate that by a line, and we can come up with a label on it that says. Count the, count the leaves in or just counts there. When we have, when we have done that, when we have added that line, when we now have our, our, our class model diagram relatively complete and it's gotten a little bit more complicated than we might have thought. If we have four classes, there's one generalization relationship. There's a couple of aggregations. There's general association. And then there are some, attributes and operations.

37 – Relationship Issues

Some of the issues that arise when we try to determine relationships. First of all, all of the indicated classes are really part of an overarching TreeCountingSystem class, as we mentioned before. Such system classes are not normally displayed in these diagrams. But you can think of each of the rectangles as being a part of, or an attribute of. The, this overarching class. The textual description from which we, began was not truly characteristic, obviously, of typical requirements documents, which can go on for hundreds of pages and have lots of specialized vocabulary. Also, we went into implementation details. It was actually describing an algorithm. Requirements documents don't necessarily describe solutions, they describe problems. In general it is important to distinguish the analysis and design phases of a software development effort in order to avoid prematurely biasing the approach taken towards solution.

38 – Summary

So to wrap this up, Object Oriented Analysis is a valuable first step to take during a software development effort. It can get you started in understanding the problem to be solved, and suggesting a breakdown of a solution system indicate, its component parts. However, as with all analysis techniques, it is important to validate the results with other stakeholders, and particularly with the customer.

P2L03 UML Class Models

01 – Introduction

In the previous lesson, we talked about object-oriented analysis, the process by which you can begin to come to understand the problem you're trying to solve. Today, we're going to talk about how would you express the results of that understanding using the Unified Modeling Language, UML. In particular, we're going to talk about UML's Class Model Diagram, which is the most popular form of UML. Besides being popular, it's also the most complex of the diagramming types in UML. However, there's no need for you to use all of its features, particularly at the start of the modeling process. There's nothing wrong with having an abstract version that you refine over time. Nevertheless, I'd like to introduce you to all of the features, so that when you need them you're aware that they're there. UML class diagrams are also sometimes called Static Structure Diagrams. They're one of UML's structure diagrams as opposed to the diagrams which you used to model behavior. Despite being called Class Model Diagrams, besides classes, they also have iconic representations for interfaces, objects, relationships, and so on. The official specification for UML can be found on the Class Resources page. And we're also going to be taking some examples from the UML reference manual, which is referred to on the resources page as well.

02 – Classes

A class in ULM or in an object-oriented language, is a description of a similar set of instances. Candidates for classes include domain objects, roles, events, and interactions. In the previous lesson we learned how to use nouns as a way of giving you ideas of what are good candidate classes. In UML a class is denoted by a rectangle that is horizontally partitioned in to three or more units. Actually, all but the first of those units is optional. The particular units which are most commonly used, are the name, the attributes, and the operations. And we'll be looking at all of those. In addition, if you wish, you could have units that describe responsibilities, exceptions, and so on. Here's and example of a UML class. There are three horizontally partitioned units. The top one features the name of the particular class, in this case it's window, it's the window class. In the middle is an area where the attributes of that class are described. In this case, they're attributes for the size of the window, the visibility, and other features. In the bottom most of the three units is a description of the operations which a window object can provide. In this case it can display, it can hide, and so on.

03 – Name Compartment

Okay, so let's drill down into the name compartment. Obviously, the most important piece is the name, and that should be a noun. After all, it describes a class of instances. For example, you saw that Window was in italics. That's one way in which you can express in UML that this particular class is an abstract class. Abstract classes, if you're familiar with object oriented programming, are contracts which describe the properties of sub-classes, and

can never have instances themselves. Besides using italics, you can also specify that a class is abstract using a tag with the word abstract in it. Why might you want to use an abstract class? Well, for one if you have some related sub-classes that have common features, you can factor those features up into the abstract class. Inside the name compartment, you can also have some other affordances. You can for example have a stereotype. Stereotypes are a way in UML of extending the base UML modeling language. You can also express some optional properties inside of curly braces. For example abstract is one of those key words you can use to give properties to the particular class.

04 – Class Features

Classes have features. By that we mean its attributes, and its, its operations. While classes, really describe the real world, that is, the problem that you're solving, features are something that are going to end up inside the computer. Obviously, your attributes are going to translate, into instance variables in your object oriented programing language. And your operations are going to be translated into methods. We're going to look a little bit at the attributes and the operations. But, in general. In addition to the names of the attributes and operations, we are going to have some type of information. And possibly also some, some names for the attributes so that they can be referred to in, in, in your models.

05 – Attributes Compartment

Here is our example class again. Now let's look at the middle compartment which is for purposes of describing attributes. You'll see that there are different attributes described. All of them have names. They have types. And they may have some other symbols that described how the names can be accessed from other classes. Those symbols, describe the visibility of that name. They're optional you don't have to provide them and there's no reason, at the start, to do that. They're a refinement that you add later on in the process. The four options in UML for visibility include. Publicly visible that's a plus sign. Private, which is a minus sign. Protect meaning that only sub-classes can access that attribute is the pound sign. And for those situations where you used UML packages that tilde indicates that the name is visible within the package. In addition to the visibility of course you must have the name of the attribute. You can indicate the multiplicity of ordering of the attribute. Now they're not shown in this example, but we'll see them later. You should give the type of the attribute. And UML has a set of built in types that you can use. You can optionally describe an initial value for that particular instance variable. You could indicate that the instance variable is derived. That is it's computed rather than being set directly. And you can give some additional properties to the instance variable using the the braces notation that we mentioned before. For example we can indicate that that particular instance variable is frozen. That means its value can't change.

06 – Operations Compartment

Returning once again to our, our window diagram. Now let's look at the third compartment, this is the one for operations. Once again, there's optional visibility, using the same symbols we had before. There's the name. There may be a return type, if the operation returns some value. It's not shown in, in these particular examples. Then there's a list of parameters. Just like you would have, if you were describing some method within an object oriented language. The parameter list includes a name of the parameter. It's type. You may express a default value. And you may also indicate, whether the particular. Parameter, is an input parameter, an output parameter or an in, out parameter. In, out parameters are those

in which the value can come in, and a different value can be returned. Those kinds of param adjectives in, out and in out are not shown in this particular example. In addition to the parameters in the operations section, you can give some properties. Some of those properties are expressed within the braces, like we've seen before. For example, you can indicate that a particular operation is a query operation. That is, it's only providing information about some existing attributes that in, within the class. You can, there's, there's, properties to describing concurrency. there's, there's properties describing whether or not this particular operation is abstract, as would be seen in an abstract method in obscuring language. And you can also show, that a particular operation has Class Scope. That's shown by an underline on the operation name, and what class scope means, is that it's not a operation of particular instance, but a operation for the class as a whole. For example, let's say you wanted to know how many instances of vehicles, you had already instantiated. You can't query any particular instance and ask it about other instances. Instead you query the class, using a Class Scoped operation, and the class if you've implemented things correctly can provide the answer back to you.

07 – Abstract Class Quiz

Okay, let's do a little quiz now. Start with something very simple, a class of vehicles. First, see if you can come up with some natural subclasses for vehicles. How about now giving some attributes that obtain for all vehicles? Third, how about some operations that vehicles can provide?

08 – Abstract Class Quiz Solution

Some natural subclasses of vehicles include cars, trucks and buses. I'm sure you maybe, came up with some more. Some of the attributes that vehicles have, are their number of axles. Their VIN, that is their vehicle identification numbers, with their current mileages, and so on. For operations, usually, vehicles are capable of moving forward, are capable of carrying passengers, and so on.

09 – More Example Classes

Here is two more examples of class rectangles describing classes in a in a system. On the left is the rectangle class. It has two instance variables describing points. These might be, for example, the upper left-hand corner or lower right-hand corner. The particular instance variables attributes have names p1 and p2. And they have types point which presumably you've also declared in your UML model. In the operation section, there are three groups of operations, each group is separated by a stereotype. You can see the stereotypes because they appear within the double angle brackets. For example, at the top, there's a stereotype for constructor, in the middle there's one for query operations, and at the bottom there's one for update operations. Within the, those three groups, there's obviously going to be a constructor operation. There's query operations for giving the area, and the aspect ratio of the rectangle, and they're going to return real numbers. Real is a primitive type within UML. And at the bottom we have update operations for moving and scaling the rectangle. On the right is the reservation class. You notice things are a little different here. We don't have any attributes. We begin initially with operations, and then we have two additional horizontal units. One for responsibilities and one for exceptions. Note specifically that the words operations, responsibilities and exceptions here are not part of UML. They're just here to show you what the boxes are being used for.

10 – Class Description Quiz

Okay, let's do another little quiz now. How about providing a class rectangle for bank accounts? At least give me a name, give me some attributes and give me some operations for a bank account.

11 – Class Description Quiz Solution

I've chosen the name Account. You may have something that's slightly different. But in, in UML there's a convention that class names begin with a capital letter. For attributes I have an account number, the owner of the account, the current balance, which happens to be a derived attribute. That is, it's computed rather than being something that is specified and changed. As far as operations are concerned there's obviously going to be deposits and withdrawals. Of course you could elaborate on these three compartments by filling in all of the possible notational possibilities that are, that are there. But for now this gives you an idea of what class rectangles are used for.

12 – Advanced Features

There are some additional advanced features of class models. Four that I'd like to just briefly mention are interfaces, parametrized classes, nested classes, and composite objects. If you're familiar with an object-oriented language like Java, you know that you can express in your program, a type by using the interface construct within Java. In UML you can also have interfaces. And in those interface descriptions, you typically describe what that interface provides to the rest of the system and what it requires from the rest of the system. Parameterized classes correspond to Java generics or C++ templates. That is, that is they provide a way of, for example, describing collection classes by giving a parameter that is a type of the class. You have a set of vehicles, you have a set of bank accounts. Thirdly, our nested classes. If you're familiar with Java, you know that within a Java class definition, you can have other classes. These are sometimes called nested classes or inner classes. And UML provides a feature for describing those situations. Finally, you can have composite objects. These are objects that contain other objects within them. Diagram allows you to express this by having class diagrams that have class rectangles that have other class rectangles in them. As I said, these are advanced features just so that you're aware that they're there.

13 – Relationships

In object orientated analysis we saw that nouns could give us a good lead into what the classes are going to be. Similarly verbs can be used for several purposes one of which is to describe what the relationships are between the classes. In UML there are three kinds of relationships. There are associations. For example, association between people and vehicles, people drive vehicles. There's generalization, that is a car is a kind of vehicle. And there's dependencies. There might be a dependency between cars and pollution laws. If a pollution law changes, cars might have to be adapted. For example, putting on some kind of pollution control device.

14 – Associations

Let's have a look at these relationships, beginning with associations. Associations are denoted by solid lines connecting two class rectangles. Here's an example of a UML class diagram containing two relationships and three classes. We have the Polygon class, the Point class, and a GraphicsBundle class. Between Polygon and Point, we have a association called

Contains. That is a polygon contains points. The little filled triangle to the right of the word Contains means that, when reading aloud that particular relationship, you would read from left to right. So polygon contains point. You wouldn't say point contains polygon. You would say something like point is contained by polygon. The second association at the bottom, between GraphicsBundle and Polygon, isn't named directly. This is fine. We'll see that we can describe it using roles, which are ways of saying, giving similar information about how the association is relating the two, the two classes. There's lots of possible notational affordances for associations. You can have a name, as in contains. You can have association classes. They weren't shown in this diagram, but we'll look at them a little bit later. And you can have aggregation and composition. In the example, we saw both of these. The open diamond indicated aggregation and the closed, that is the filled diamond, indicated composition. In both cases, we are saying that the two classes are related by some kind of containment relationship, that is a polygon is made up of points. We saw reading direction, that was the filled, filled triangle. We can also express Navigability, which is the appearance of an arrowhead on one end of, or both ends of the association line. This indicates that the primary access pattern for those classes is in the direction of the arrow. That is, we are going to be going normally from polygons to their points and not in the other direction. You can express multiplicity, in the, diagram we saw star. We saw 3 dot dot star. Star means any number of, instances. 3 dot dot star means between 3 and any number of instances. We also saw a property ordered which indicates that at least for the case of the polygon and its points, those points are in a particular order. They might be, for example, in clockwise order. Not shown in the example diagram is the ability in UML to, UML class model diagrams to express associations which involve more than two classes. In our text browser example, there were three classes involved and we used a rhombus into which the various lines, the various lines come in to indicate all the participants within that particular association. We saw also the fact that you could have role names. The word bundle, adjacent to the graphics bundle class, indicates the graphics bundle is playing the role of bundle in that particular association. You can have these role names on either or both ends of the association line, or you don't need to have them at all. Also not shown, are the fact that you can express qualification. You can think of qualification as this as indicating what are the keys into the set of instances. We'll see an example of that in a minute. And you can express also, certain, Constraints on the association. For example, that they're ordered, that they're frozen, that is, the association can't change, that you can only add things to it and so on.

15 – Assocation Class

I mentioned, a minute ago, the association class. You can think of an association class as ac-, as an association that has some class properties. For example attributes, or you can think of it as a class that has some association properties. Here's an example. If we have a company class. And a person class, and we have some kind of association between them that a person, has a job with a company. We might want to indicate what that person's salary is, from that company. So this is not really a property of the person. Because the person might have more than one job is not really a property of the company because the company certainly has more than one person. It's really a property of the association itself. Association classes are indicated by having a dashed line. That abuts into the association line. At the end of the dash line is another rectangle. In this case it is the association of class called job, and in the class rectangle for job there is an attribute of salary. Notice something else a little peculiar here. That job, has an association with itself. This is called a recursive association. For recursive associations in particular, you better use role names. This case, we're talking about the manages association. So one job might manage another job. The department head

might manage the staff and therefore we want to have roles for the boss or supervisor, and the worker.

16 – Aggregation Composition

I mentioned also, aggregation and composition. Using aggregation in particular is very common in UML class diagrams. You often want to say that one class is related to many instances of another class. And you use the aggregation association to do this. It's still an association, it's just adorned with an open diamond to indicate that it's a particular kind of association called an aggregation. I want to say a word, though, about the difference between aggregation and composition. It's somewhat subtle, and it gets to the point that aggregation doesn't really say much about the semantics of the relationship. In particular, it doesn't say much about the lifetimes of the participant objects. For example, let's say you had a house class and room classes. Clearly, a house has rooms so you'd expect there to be an aggregation there. But think further, if you destroy the house you're also destroying the rooms. Therefore, instead of using aggregation, we would use composition. That is, we'd fill in that diamond. In compositions, there is a responsibility for managing the lifetime of the constituent objects. That further says that a particular constituent can only belong to one composition. Compositions also have the transitive property. That is, a house can have room and a room can have closets. For aggregations there's no rules like this. Aggregations are general situations. We might say, for example, that a room has a table. Now this is an aggregation situation, because we could certainly destroy the room after taking the table out. They have separate lifetimes, and therefore we'd use aggregation instead of composition.

17 – Aggregation Composition Quiz

Okay, now let's have a quiz that tests your understanding of this distinction. I'm going to list four different pairs of classes and ask you to tell me whether they're associated by a composition, an aggregation, or a plain old association, which is neither a composition or aggregation. The four examples are courses and students, person and spouses, bank accounts and patrons, and fonts and glyphs.

18 – Aggregation Composition Quiz Solution

Courses and students are an example of an aggregation. That is, you can shut down a course without shutting down the students. Persons and spouses is a plain old association. They have independent lifetimes, and there's no containment relationship between them. However, bank accounts and patrons is an aggregation. That is, a patron can have a bank account. Finally fonts and glyphs. They're composition. If we get rid of a font, we get rid of all the glyphs that are in the font.

19 – Qualifiers

Another refinement of associa, of associations that I mentioned, is qualifiers. Qualifiers are indicated by small rectangles, that are on the sides or edges of class rectangles. The small rectangles contain the name of one of the attributes, of that particular class. The attribute within the small rectangle is the qualifier, that can provide access to, instances of that particular class. If you were doing a relational database model you would think of the qualifier as the key, into the set of the instances. In the leftmost example here, we have bank, and the account number is the qualifier. Note also that there's some multiplicity information, that a person can have any number of bank accounts, for example. On the right, we have the situation with a chessboard and its squares. How would we identify a particular square? In this case, we're

going to use a pair of attributes, giving the rank and the file of the particular squares within the chessboard. Notice also that in the chessboard situation, we have a, a composition. Notice the filled black diamond.

20 – Links

One further nuance to mention about associations, is the idea of links. Just like classes can have instances, associations can have links. For example, if we had the situation where a company hires people, we might have a situation where IBM hires Bob. IBM hires Alice. Hewlett-Packard hires Tom. Hewlett-Packard hires Alice. She has two, two jobs. In this situation, we would have four different links. One for each pair involved in the association. Notice in this particular diagram that in addition to indicating all of the particular instances of the classes, that the lines here are going to indicate links. That is lines between rectangles for which the rectangles are instances indicate links, that participate in association. Notice also that we have role names here and we have qualifiers.

21 – Generalizations

The second major kind of relationship that you use in UML class model diagrams is generalization. Generalization is also indicated by a solid line, but in this case the line ends with a triangle. The class rectangle that's adjacent to the triangle is the superclass or parent class. And the other class rectangle is the child class or subclass. The semantic import of generalization is that all instances of the subclass are also instances of the parent class. That is there's a subset relationship. Let me warn you though that generalization is not the same as inheritance in object oriented programming languages. Inheritance is an implementation technique, generalization is a modeling, approach. We'll see how that difference plays out later in the course. In UML, generalization supports both multiple parent classes for a given class and multiple child classes for a given parent class. Moreover, you can specify discriminators. That is names of groups of subclasses. So here's an example of the UML class model diagram in which generalization is illustrated. We have a superclass called Vehicle. Note that it's got four lines coming into it each with an open triangle so it's got four sub-classes. Those sub-classes are wind powered vehicle, motor-powered vehicle, land vehicle and water vehicle. Notice also that we have two grandchild classes. We have trucks and we have sailboats. So let's think for a minute about trucks. Trucks are motor powered vehicles are motor powered vehicles but they're also land vehicles, that is truck has two parent classes. Similarly sailboat is a wind powered vehicle and a water vehicle so its got two parent classes as well. With respect to the parent level. We have two categories of sub-classes. We have a category related to the power that moves the vehicle, it might be wind or it might be motor. And we have a category having this labeled here as venue, indicating where the vehicle does its moving. Is it on land or is it on water? Also visible in this diagram are some properties in curly braces. Those properties indicate properties of the sub-classes. If a parent class has two child classes, and instances can belong to both of the child classes. We want to use the overlapping property. If that can't be the case that is if a given instance can only belong to one child, we say that those particular sub-classes are disjoint, their members belong to one of the child classes not the other and. But by our definition of generalization, the instances do belong to the, to the parent class. A second kind of constrained or property we might want to express is whether or not the set, of child classes covers all of the instances or not. If that's the case we say, we would use the, property, complete, and otherwise we would say incomplete. Why might a modeling situation be incomplete? Well you might have some weird hybrid vehicle that doesn't belong to any of

the. The child classes, but nevertheless we want to have an instance that, that recognizes it, or, or models it, say the Segway, for example.

22 – Constraints Quiz

Let's do a little quiz that checks you on this, with respect to completeness, and overlapping. I'm going to give you two examples. The first example is athletes, and let's say we have, subclasses of athlete, for baseball players, football players, basketball players. Determine whether first of all, those subclasses are disjoint,and are they complete? Second example is books which can be Paperbacks, ComicBooks or hardboundBooks. Are they, complete subclasses? Are they disjoint, subclasses?

23 – Constraints Quiz Solution

For athletes, we have overlapping. Remember Dion Sanders? Played both baseball and football? And they're incomplete because they're are certainly a lot of athletes that don't play one of those three sports. With respect to books, I'm going to say that these three categories of, of subclass are disjoint. Moreover, they're incomplete because there's other categories, of, of books.

24 – Superclass Subclass Quiz

Here's another little quiz that checks your knowledge of, of generalization. Say we have two classes. One is omnivores, people who can eat anything. And we have vegetarians, people who only eat vegetables, or, or don't eat meat. Okay. Which of those would you have as the superclass and which is the subclass? Think about it for a minute.

25 – Superclass Subclass Quiz Solution

Your natural inclination might be to have Omnivores as the superclass, because it's probably going to be bigger, and doesn't work however. Omnivores have a property, that is they can eat meat, the vegetarians don't, vegetarians don't eat meat. Remember our rule that the definition of generalization is. That instances as the subclass have to have all of the properties the instances the parent class. What that says is that Vegetarians are the superclass and Omnivores are the subclass. After all, Omnivores can eat vegetables. Okay think about that for a minute in, before you do your next class model.

26 – Summary Quiz

Here's on final quiz for you. We saw in the previous lesson that there are 14 different kinds of UML diagrams, some of those are structure diagrams and some of those are behavior diagrams. What I'd like you to do is to draw a UML class diagram. That indicates the parent-child sub-classing relationship, amongst those kinds of diagrams. That is, you're going to have classes for each of the 14 diagrams. You're going to a class for class diagram and you're going to have a class for class structure diagram and behavior diagram. See if you can fill in the details.

27 – Summary Quiz Solution

Here's one rendering of the answer. You have the structure and behavior diagrams as child classes of the diagram class. And then we have seven different structure diagrams as children of the structure diagram class. And the other seven behavior diagram types as subclasses of behavior diagram.

28 – Summary

In summary, UML provides a rich vocabulary for modeling system structure. And the UML class model diagram exhibits many, many different features. However, there's no need for you to use all of its affordances. Particularly at the start of the modeling process. Never the less, each affordance implies a question to be answered. What is the multiplicity? Are these values ordered? What's the qualifier? Does the system that you are modeling, exhibit the property expressed by that affordance? One of the important benefits of modeling, is that it encourages you to face these questions early, in the development process. Because if you forget and they, they may later come back to haunt you.

P2L04 Design Studies

01 – Design

Design is all about making decisions. Generally trading off among nonfunctional criteria. There are various sources that can inform these decisions. Such as the customer, the end user, technology specifications, and competitor's products. Sometimes however, a more detailed analysis is required. Examples of such devices include simulations, prototypes, and the topic for today, design studies.

02 – Design Studies

When an architect designs a building, often one of the early steps is to undertake a design study. This takes the form of a series of scale models where different approaches are explored in order to get a better feel for the design space, which is the range of possibilities available as solutions. The same approach is used in other areas, other areas of design such as cars, planes and even clothing.

03 – Definition

A design study is a rigorous and systematic evaluation of the factors that influence a design. It should begin with a determination of the relevant criteria, how they are to be measured, and what measurement values are deemed satisfactory. The study itself consists of a comparison of the various possible approaches in which each approach is measured against the predetermined criteria

04 – Design Spaces

The process of doing a design study helps the designer explore a space of possibilities. Although aesthetics may play a role in the ultimate decision process, other more objective factors should be examined as well. For buildings factors such as the cost of construction, availability of building materials, conformance to building codes and zoning regulations and effect of traffic on traffic patterns may be taken into account. For this class, we want to use design studies to evaluate the design of computer programs.

05 – Design Factors Quiz

What are some of the factors that might be used to compare different versions of a program? Type your answers into the text box.

06 – Design Factors Quiz Solution

Did you say things like performance, memory footprint, time to construct? Note that we particularly don't include correctness in the factors to compare. That is, we assume that all the versions that are constructed work, but differ in other nonfunctional ways.

07 – Teaching and Learning

I hope this doesn't come as a surprise, but I can't really teach you design. The best that I can do is to teach some of the surrounding skills such as analysis, modeling and evaluation. Instead, design must be learned, learned from doing. Hence a significant portion of your effort in this course should be spent actually designing. The vehicle for doing this designing is a series of projects, each accompanied by a design study.

08 – Projects

Each of the projects involves solving a design problem in several ways. To determine which approach is best, you are asked to evaluate your solutions in a systematic way, that is, to conduct a design study. The result of each study is documented in a report that conforms to a prescribed format laid out in a template file linked to from the class resources page. We will now go through the contents of the template as a way of illustrating what a software design study is all about.

09 – Experiments

You can think of a design study as an empirical scientific experiment. As such, there are research questions, subjects of study, experimental conditions, methods, tools, metrics, independent and dependent variables, data collection, statistical analysis, and conclusions. As with a scientific experiment and overall goal of a design study is repeatability. That is, someone else should be able to take your study report, use it to recreate the study conditions, and reach the same conclusions that you did. It is one of the goals of this class that you'll learn the skills to produce and present an industrial quality design study.

10 – Report

The design study itself is presented in a report. It may include charts, tables, graphs and screenshots, as well as, as well as descriptive text. It is, however, not a narrative, but a dish, dispassionate description of a systematic exploration. I want your reports to be professional in quality. This means that you should treat it like you would if you were preparing to show it to customers or submit it for publications. Its spelling and grammar should be checked, and it should be carefully proofread by a team member other than its author. We will now go through each of the sections of the report, indicating what is expected in that section. This should also give you an idea of what you need to do during the study itself to gather the data that goes into the report.

11 – 1 Context

The first section is titled Context. It provides background and motivation for the study. So the reader who is not familiar with the class or the project can make sense of what you have written. It should also define any specialized vocabulary necessary for the reader to understand what you are saying in the report.

12 – 2 Research Questions

Section two is titled research questions. A designed study examines the tradeoffs between various non-functional requirements, for example, space and time. Each tradeoff can be expressed in the form of a question, such as, how are execution times and memory footprint effected as the amount of pre-processing computations vary? The second section of your report lists such research questions. Each question should be formulated in a neutral fashion

with regard to the due, dependent variables being measured and also indicate what factors are being varied. That is, the independent variables. And each question should be numbered for later reference in your report

13 – 3 Subject

In experiments, a subject is something that you are studying, usually a program. A design study compares multiple subjects. In the third section of your report, each subject should be briefly described, differentiating it from the other subjects.

14 – 4 Experimental Conditions

The fourth section is titled Experimental Conditions. A software design study normally means running several versions of a program, making measurements, and evaluating the results. These programs' executions take place on computers configured with resources. Such as their number of cores, the amount of RAM, their clock speed, and potentially net, the networking that networks them together. To support the goal of repeatability, this configuration information should be explicitly documented in your report. The fourth section of the design study describes the experimental conditions under which the study is conducted. In particular, it describes the environment of which the study will take place. This includes elements such as the machines, [COUGH] their models, operating systems, programming languages any virtual machines and their versions. Where relevent, the network, the build and execution parameters, input files, and confounding factors. Such as other users on the machines at the same time or other, processes going on

15 – 5 Variables

The fifth section is titled variables. Design studies themselves have to be designed. In particular the independent and dependent variables must be identified and appropriate metrics specified. Design studies, like experiments, allow designers, like scientists, to alter conditions and note results. The altered conditions comprise the independent variables. And the corresponding results comprise the dependent variables each variable has a unique name a description and a unit of measurement such as seconds. Sometimes the units are easy for example time other variable such as maintainability require you to think carefully and invent an appropriate way to measure it. But this section of the design study describes the variables, both independent and dependent, the units and measures, and how the research questions address them. The section should also include a summary table, with three columns, which for each research question, lists the independent and dependent variables you used, in answering the question.

16 – 6 Method

The sixth section describes the method that you use to conduct the study. This includes the number of trials or measurement devices and tools, any randomization techniques were appropriate, and number of significant digits you used in your measurements and so on. This should also include an explicit statement of which subjects will be run, and the arguments used for each of your trials. For example if you were studying the relationship of performance to grid size, you would want to specify what different grid sizes you will be using. The section should also briefly describe any statistical techniques you will use, for example linear regression

17 – 7 Results

The point of conducting a design study is to produce data and the seventh section is where you describe these. It's titled the Results section. It presents the data collected and their statistical analysis. Any speculations and generalizations are reserved for the next section.

18 – 8 Discussion

The eighth section is where you get the opportunity to interpret the data you collected and provide a discussion of its implications. This often means offering an explanation for of any unexpected values you see. This section also allows you to reflect on the experimentation itself including any suggestions, any suggested further work or for improving the study process itself.

19 – 9 Conclusions

The final section allows you to summarize your results, and draw any conclusions. In particular, in this section you should provide explicit answers, to each of the research questions you raise in the second section.

20 – Deliverables

Each of the projects in this course has three deliverables. The source code, involving, solving a specific problem in several ways, a project report containing project specific content, and a design study report. The design study represents an, the explicit knowledge about the design, that you learned during the project. In summary, here are the expected sections to be included in the design that are important. Section one includes the context. That is the background, motivation and vocabulary. Next section is the research questions. Then descriptions of the subjects, experimental conditions and the variables, both independent and dependent. Section six has the method. Then come the results, discussion and conclusions.

21 – Wrap Up

I want to repeat that I can't teach you design, you have to learn it. And I want you to learn it using the projects that have been defined for the course. I encourage you to invest energy in those projects and to think systematically about the design issues that each one of them raises. Express that systematic thinking in the form of some experiments that you run, then write up those experiments in the form of a report. I think by doing this, it will force you to reflect upon the design process, and thereby, make it much more real to you.

P2L05 Library Exercise (UML)

01 – Introduction

Hello and welcome to a tale of analysis and design, featuring Spencer Rugaber, as the librarian, and Alex Orso, as the software engineer. Hi! I'm here waiting for Spencer, my librarian friend. He needs some help developing an information system for a library. So I asked him to write down the requirements for the libra... Oh, that must be him. Hello Alex. Hey Spencer. How's it going? Good. Did you get those requirements I emailed you? Oh, you emailed them. Now let me check. And, by the way, get some coffee for you here. Thank you very much. Oh yeah. They're right here. Let me see. Oh, good. Oh, yeah, good. We have, what we need. So the, the way I like to do this is. I like to start by looking at the requirements and identifying the nouns in the requirements, because those tell us the kind of the relevant elements in the, in the requirements. So if you don't mind we can start looking at those and you can tell me you know, whether the ones that I am identifying make sense or not. Sounds good. All right.

02 – Analyzing Requirements

Okay so let me start underlining these nouns, and I'll start identifying the ones that are relevant, and I'll ask you some questions or you can ask me questions if you see something that doesn't make sense to you. Good enough. okay, let's see, patron. It seems to me that patron is definitely an important entity. That's, that's what its all about. Okay, all right, so actually, the way I'm going to do this, I'm going to take all these relevant entities and I'm going to start putting them into what I call a class diagram. So you don't really need to know what that is exactly, but imagine this being a, a diagram in which I'm drawing, I represent in all development items as rectangles with a given name and, and then later on some attributes. Okay. Okay, and I'm, I'm just going to put them there. So I'm going to start with patron. I'm going to create one class for the patron. I'm going to give it the name patron. And by the way, assuming that you'd probably figure out, it's important that we represent, we use the right names so that it's clear when we're looking at the class diagram what we're referring to, so I'll just use the, the nouns themselves as names. Okay, library card seems to be also a relevant element. Every patron has a library card. All right, perfect, so we'll just create a library card here. And let's see. As, as long as they're in the system. And I saw that there's a system here, this concept of system, this concept of library. And based on my experience, normally, those are kind of in an overarching themes. So this is really what we are modeling. So the only thing that will make a difference is if there were more than one library or more than one system. Is that the case? We just want one system for our one library Okay so, in this case I won't even represent those because basically what I'm representing is the system and the library. I understand, I understand. 38 00:01:42,740 – 00:01:44,420 Okay and then, oh name, address and phone number are interesting because these are important entities, but this seems like, you know, they're not entities in themselves, so they're more attributes of something else.

I would imagine that this is the way you identify, or these are elements that are important for the patron? That's what we take down when we issue the cards. Okay. Perfect. So, I'm going to take those and make those attributes of the patron, which means that I'm going to take the class that I created before, and I'm just going to write them down here so that they're represented and, and we know that these are kind of what characterizes the patron. Gotcha. Okay? And then, I guess similar consideration for the library card number. So this is to be associated with the library card? It's printed right on it. All right, so we'll put this as an attribute of the library card, then. And then, in addition, at any particular point in time. Okay, so time seems to be a relevant entity right, because time seems to occur several times in this description. For example, I think you guys keep track of how long a book has been loaned, right? Right. And there's some time associated also here. And a children's age. Oh yeah. The children's age here that I didn't see before. Yeah. So, what I'm going to do, I'm going to represent this in a sort of generic way, as a date. Okay. These are kind of, kind of classes, utility classes we call them, that are normally in every system. Okay. So I'm just going to put it down here as a utility class that will be used by different elements in the diagram. Okay, so I want to calculate the items. So the items also I mean I for what I know about libraries they seem to be pretty relevant elements, right? So these are all This is what we check out, this is what we're for. Okay, so then items definitely will become a class, and then we have a due. Oh there's also this concept of fines. I guess that seems to be important. Right? You guys give fines to people who are late. Right, right. Right, collect fines and so on. So we create a fine class down here and the children. So children are special customers, right? It's their age makes a difference? Is that the way it works? Right. They, they can only check out a few books. Okay. So I'll create them a special kind of case, a special kind of customer so I just create here a class for children. And I can see that they're categorized by their age. Right. So I'll just put the age here as an attribute of the child. And, okay, so the next one is restriction. And restriction is kind of tricky because just to be sort of a general concept. I mean, in a sense, all of those are restrictions, right? Right, this is just another one of these requirements. Oh, okay, so, so we don't need to represent it explicitly, right? Right, right. It's just telling us how the children, yeah, okay, right; this is just another requirement, so I just won't consider that for now. And oh, I see that these books and audio video materials, I guess these are things that the patrons can check out, right? Those are some of the items, right. There are two more down here, right? Reference books and magazines? But, they can't be checked out, but they're definitely in the library. Okay, so then I'm going to represent all of those actually, now. So, I'm going to have books, I'm going to have audio video materials, reference books, and magazines. And I'm just going to have those as classes. Then, okay here we have week, and we already represented this general concept of time, so week will be represented by the date class as well. And oh, I see best sellers. So best sellers are also, I guess, items that can be checked out, right? Right. Okay, so I'll just represent those as a class as well and an additional item that is relevant for the library. And the limit, this is also a time limit, right? Right. So it can also be represented with a, with a class. Oh, here we have cents, and for cents, same consideration that made for time. This is kind of the money, is a general concept that in all currency, many, in many IT systems. So, I'm, I'm going to just have a money class here, which is another utility class. Okay Okay, and, oh, here I have value, so value is a property. Let me look again at the requirement. Oh, it's the value of the item. So value I'm going to put in the item as an attribute. Okay? Okay. That's how much it cost us. Okay. Perfect. Seems like we got them all. Right? Anything I forgot? That looks like it. Okay, so this one, what I'd like to do. We have a kind of a first take, first cut at the class diagram. I'd like to kind of move to that and go through the different classes with you. And I'll ask you some questions again.

And you can tell me whether there is something that jumps at you that's not right. And then we're going to try to refine that. Okay Okay. Sounds good.

03 – Refining Classes and Attributes

Okay, so this is our first, class diagram. So, let me ask you something about. Okay. What we've done so far. I also sent, in what I sent you, I also had some stories about how the actual Library is used. You asked me to do that and are we going to take, use that here? Glad you asked actually. yeah. Those are, you know, what we call use cases, or what we will use as scenarios kind of things that we will use to derive use cases. And they're also a very good way of extracting requirements. We're not going to look at them right now because now, because we're more working on kind of the static structure of the system. But after we're done with the class diagram, you know, we will do it at a different time. But we're going to use those to see how the libraries actually use them, and see whether we can get more information that we can use to refine our requirements based on that. Okay. Okay, So, for now, we'll just focus in on the, structure, but, just so you know, I'm, I'm glad you sent them, because they were going very useful as well. Okay. So let's see. Well, first of all, let me, seems like that this is already pretty crowded, right? We have a number of, classes. So let's see if there's, some class that may be superfluous and we can model in a different way. So, for example, you, while, while thinking of this I was thinking, the library card, it doesn't really contain much information, right? So is it basically just the number? 32 00:01:22,736 – 00:01:23,948 The card has a number on it. We have a separate vendor that does that for us so. Oh. We don't need, it doesn't need to be part of this system, we just have to make sure that every patron has a library card. Okay, so basically for you, in a sense, the library card is just an ID that gets associated with a patron. That's right. So I think that the best way to represent this, I mean, unless you need an entity because you are creating it yourself, but it seems like you are not. I would just remove this one and I would like to put this, basically to take the library card number and add it to the pattern. Okay, makes sense. Okay, so I'll add it here. And as an additional attribute. Okay, and it will eliminate this class. Okay. Okay. Oh, and, wait a second, so I guess also the child needs a library card number, right? Child needs a library card number, but let me ask you about that. Is, is child a separate class, or is it just another kind of patron? Oh, I see, I see. Because, yeah, it is sort of a special patron, right? And, so maybe we should, maybe we should represent it as a kind of a refinement of the patron. Hm, but then that made me think. So what is the only thing that characterizes children? Is it just the age? Well, if they're, that they can't check out more than five books. Okay. And the, and the only difference is the fact that they are less than, you know, twelve years old. Twelve or less, right. Twelve or less. So, I guess, you know, I would probably like to represent this by making the age explicit in the patron rather than to represent it as a class. And I'll tell you why, because one, one of the issues, and you know, that might happen again, is that, basically, there are patrons that are children. And they're no longer children, when they come you know 13 or older right. Right. And if we represent them with a separate class in a sense, then we cannot really change the type of an instance of these classes. So we're left to kind of destroy the patron, create a new one, so that means we also have to transfer any history we want to keep history and so on. So I, I think I kind of like better the idea that I represent the age exclusively in the patron, and then I'll behave differently, based on whether the patron is 12 years old, or younger, or 13 or, 13 or older. This, do you see any problem with that? It makes things a little simpler. Okay, and we actually, it allows us also to eliminate one class here. So I'm going to proceed this way. I'm going to eliminate the children class, and I'm going to put the age in the patron. Okay, and let me see. But in this spirit, actually, something else that jumps at me is this idea

of the bestseller, because I kind of feel like, we might have the same problem. So, what is the story? What is a bestseller. Well it's an item that we want to restrict how long people can keep, because there is such demand for it. I see, and so basically a book that's a bestseller, like the New York Times bestseller, is a bestseller forever? No, no, no it's hot for awhile, and then it becomes just a regular item. I see. Hm. Then I guess it's a similar situation to the one I was mentioning before, right? Okay. That if we have a book, it will kind of have to change its type if it becomes a best seller. Then we have to change its type again, if it's no longer a best seller. Right. So it seems to me that a better way to represent this, is just to eliminate this BestSeller class and instead, I'm going to put the best seller attribute, which would just be a Boolean in the book. Okay, what do you mean by Boolean? Right. We don't know what Boolean is, right? The Boolean is basically just a number. It can have two values, right? True or false. Okay. So we usually, normally use it in this in this case. Imagine one, zero, right? Then it's just kind of the basic. Okay. You know, the bits, right? Okay. So, this is just telling us, it's like a flag that is telling this book is a best seller, or not. Okay. It's very easy to change this value and make a book a best seller or not a best seller, than just creating and destroying instances of these classes. Okay, makes sense. Okay, so at this point, this already looks better, right? Because we have, less classes, and I think we did, yeah, we did some serious cleanup. That's good. Okay, so now that we eliminated some of this, what I would like to do, as I said, we are going to both clean up, but also refine. I would like to go back to our, requirements and see whether we can identify additional attributes for this, class that maybe are not as obvious as the one that we saw so far, okay?

04 – Adding Attributes

Okay, so let me look at the requirements and it's something that I can see here that we didn't point out before is that there seems to be clearly some concept of a due date. And I'm telling you why I'm saying that because here, for example, I notice that it says when items are due. We mention overdue several times, so is this something we need to keep track of? Yeah remember when we used to stamp them on the books? In the stamp pad? Oh yeah yeah yeah! Oh course! Right? Yeah we definitely keep track of, the system has to keep track of when books are due. Okay. So it seems to me that one good way of doing that is by basically adding an attribute to the, item. Okay. And I'll just call it due date. Okay. So basically for each item in case it's loaned there will be this attribute that will contain the value of Okay. Of when, when the item is due. And then, something else that I noticed here is that down here, it seems like the requirements are saying that an item can be renewed only once. So, I guess, that's something we need to keep track of, right? Yeah. The system needs to know. We have to know whether they've renewed it or not. Okay so, I'll do a similar thing here. I think I want to go and add a an attribute that we'd call number of times renewed, and add it to the item class. Okay. And this is kind of more generic than what you need, because here it says only once, but let's say that in the future you want to allow it to, kind of renew twice, you'll be able to use these attributes again because, we can just count how many times it was renewed. Okay? Makes sense. Alright. And one last thing I want to point out. And this seems obvious but I'm going to check with you anyways. And seems like there is a basically the need to keep track of whether an item is checked out or not. If you look at the text here, the requirements here, I can see that check out and checked out are mentioned five times. So, I'm assuming that that's something also that we want to know about an item, whether it's checked out or not. We have to keep track of whether they're checked out. Okay, so I'll add an additional attribute there. So I'm going to again go back to the diagram and I'm just going to write here also the checked out attribute. And, I think that's it as far as I'm concerned. Is there anything that you think

is missing? Well, I do have a question. Would checked out, better not be the case that someone can check out a reference book. Oh, I see, I see. Okay. I mean, it's only the books and the audio visual material that can be checked out. Right, right, right. Okay, so I, I guess, well the way I will fix that is, I'll probably put yet another attribute in the item class, and I'll call it loanable. And basically, this attribute is just telling us whether an item is loanable or not. So, when it's not true and loanable is not on. Basically, that item can be checked out. Okay. And, the system would know this. The system will know that. And prevent it from happening. And prevent it from happening. Okay? Alright. Perfect. So, we're going to do that and, any other objections, any other? No, that was my question. Okay, perfect, so what I'm going to do next, I mean, I haven't mentioned that yet, but you know classes right now we just looked at the attributes right that give you sort of the state of the class. And there's something else, there's a second part of the class that is kind of an orthogonal aspect, which is what the class can do. And we call those operations. So normally these kinds also have operations, I guess you know it would make sense to you as well. And one way, one very natural way to identify operations is to look at the requirements and look for verbs. Because verbs associated with an item will tell you basically what the item can do. Okay. So I, I'd like to go back to the requirements and start, the same way in which we underlined, nouns, we're going to underline verbs and we're going to see which ones of those verbs actually represent actions that we want to represent explicitly, we want to model explicitly in our class diagram. Okay. Okay.

05 – Identifying Operations

And before we get started actually, I'd like to mention that there's just, you know, FYI, there's different kinds of verbs because what I'm looking for is really action verbs. So verb, verbs that clearly express an action that can tell me that, you know, what, for example, an item could do, 'kay? Okay? Not the verbs that represent, for example, relationships, 'kay? Okay. So, and the, there, and the ones that I've identified und, underlined here actually, I, I underlined complete sentences so that you kind of we can look at the verbs in in context. And the first one is this sentence that says that the library may need to know or to calculate the items a patron has checked out, when they are due, and any outstanding overdue fines. So I, I will imagine that this is representing a situation in which you bring up a patron's record and you start looking up this information. Is that [CROSSTALK] The, the patron often wants to know what they have currently checked out. Oh, alright. Or when are their due or how much they're owed or. Oh, in fact, and then now that you mentioned it, I think you sent me. One of the scenarios you sent me had to do with that, right, with the patron coming in and asking for this information. So yeah, and it makes a lot of sense. So what I'm going to do, I'm going to model this by adding this three operations to the patron method. The first one, I'm going to call, itemsCheckedOut and, basically, it's an operation, but you don't need to, you know, understand the implementation details, but when you call this operation, it will give you back exactly this information, so the items that are checked out by that patron. The second one, I'm going to call it whenDue. That will tell you basically when a, when an item is due. And the third one is going to be called the outstandingOverdueFines and, you know, as the name says, it's going to tell you what are the outstanding overdue fines for that patron. Okay. And as you might notice I mean, I, I'm going to separate the, the, the attributes from the operations by having a separate kind of subrectangle so, in this way, it's clear what is attribute and what is, what is an attribute and what's an, what's an operation. Gotcha. And let me see then. Okay, for the second one you can see that that patron can check out books and audio visual materials. So I guess, similarly you, you build kind of the record for a patron. The patron will give you an item and you will record the fact that the patron is kind of checking

it out. Right. And is that operation related to this, the checked out attribute that we did a minute ago? It is actually because what will happen then again, you know, if we jump ahead a little bit would be that every time you invoke this operation. So I'm going to represent this as a checkOut operation for the patron. Every time you invoke this, you will also have to say something about the item and so we will also flip kind of that that, that build information in the, in the, in the item. Okay. Mm, 'kay? And, and finally, here, I can see that a patron can request a book or an audio video item Is not currently in. So I guess this is referring to items that are already checked out but for which there is interest. Is that? Right. So, particularly, the popular items the patrons want to get on the list so that they get notified when it comes back in and. Oh. And check it out. I see. I see. Okay. Then I'm going to do the same thing here. I'm, I'm going to add this method, which I'm going to call request and I'm going to put it here in the list of the methods in the list. Okay. Of operations for the, for the patron, okay?

06 – Adding Relationships

OK I like the way this class diagram is coming along. So at this point I think we have all the classes that we need. For each class we specified the attributes or the characteristics of the class. And we also specified the operations so we know what the class can do And, I like to kind of move forward on this, but I first want to see that you're fine with the class structure. So that's the way the class structure is going to be in terms of attributes and operations. So anything that bothers you? Well, one thing I didn't understand is how come you put check out over where the patron when it's really the item being checked out? Right. Okay. So that actually is you know, is a perfect segway for the next thing that really wanted to model. Because what you're talking about is basically a relationship between two classes which is something we haven't touched on yet. So we haven't, haven't looked at individual classes. But now, it is typical, now we are looking more at requirements, we're starting to find more things about our system, and what you're pointed out right now is the fact that patron and item are somehow related. So this checkout operation is not really something that belongs only on in the patron, because it needs to know about the item. And it doesn't belong only on the item because it needs to know about the patron. So, it's something that associates the patron and the item. Okay. And that's exactly the way we call in the UML which is the notation that we're using here this kind of relationship. So, we're going to represent that by drawing a line between these two classes that tells us there is an association. And we're also going to give a name to this. Since this refers to the fact of checking out items. We're just going to call it, checkout. Gotcha. And notice that this basically you know,eventually will end up kind of replacing this attribute. Because the existence of this association will tell us that this is checked out. We're, we're not going to, you know, do it right now, but in the final cleanup or the diagram, this name will disappear. Okay. Okay. And so since we started talking about relationships and associations, is there any other kind of relationship that you see here? Well, what you just did with checked out is, it seems similar to the whole issue of requests. It is, it is. So a request is something else that happens in both, you know, in the patron and in the item, it involves both. And in fact in a request, I would definitely represent this as an additional association. So I will just draw an another line between these two that represent that specific kind of relationship and I will call it request. So that indicates that this association refers to a request that also connects the patron with an item. Okay. And, let's see. Any, anything else that jumps at you? Yeah, well, how about all these ones down at the bottom? I mean book and item's got to be related, right? A book is a kind of item, And audiovisual... are there associations between them? Can you repeat that, you said that the book, yeah? Is a kind of item. Perfect, that's exactly what we're modeling next, which is, this, what we call the is-a

relationship. So you said, a book is an item? A book is an item. And, we can model that in the diagram. So, we do that using another kind of relationship between the classes. So we're going to represent that as a specialization we call it. And, a specialization is indicated in this way. Okay? With this arrow at the end, so a solid with this kind of arrow at the end. And we can do the same for book, magazine, reference book and audiovisual material. So we're all going to connect, we're going to connect all of them, to the item, using the same kind of connection. And now that we have connected all these four, with item and indicated them in subclasses. That's something else that we can do. So we can make this kind of cleaner. And I'll tell you what I mean by that. So now we have this loanable attribute that refers to item, but it seems to me from what you were saying before, that loanable is not really an attribute of an item. Right? It's more of a characteristic of the type of item. Right. Is that right? Right. Books, and audio/visual are loanable but the others aren't. Okay, and so representing it here, it's okay to, it will work. But it's not really right so from the style standpoint it doesn't really you know, it's not the best way of modeling this. What we're going to do instead, we're going to use this specialization relationship to make that more explicit. To make it cleaner. Okay, so what I'm doing here is I'm going to take this hierarchy of classes, this is just on two levels now, and I'm going to kind of make it a little richer. So I'm going to add an intermediate set of classes. And in particular I'm going to have these two classes that I'm going to call non loanable item and loanable item. So, they're both items but they tell me clearly that some items are loanable and some items are not. Okay. Okay. And then I'm simply going to put book and audio video material as subclasses of loanable item and reference book and magazine as subclasses of non-loanable item. So, if we look at this diagram now it's pretty clear what is loanable and what is not. And it's actually is a very clean, much cleaner design. And, and I see you've, gotten rid of the loanable attribute, too. I did. Because at this point this is already represented by the fact of having these two classes. And actually, something else that I did is that I moved all these attributes, value, due date, renewed and checked out, that makes sense only for loanable item. From item to loanable item. So at this point, this really is telling me that, you know, these characteristics are just meaningful for the loanable item, and not for the other ones. Well, speaking of that, the way that you got the lines going in the diagram here is you still have request and checked out going to item, even though you can't request non loanable Items. You can't check out non loanable Items. Oh, you were right actually. You got me on that one. You're exactly right. So this associations are between the two wrong classes. So, I guess, at this point, you can probably go and fix the diagram yourself. Well, can we just make the lines go from patron to loanable item instead of to item? That's exactly the way in which we are going to fix it. So, we're going to move these two associations down here. And at this point, this will represent the right relationships in the, in the diagram, and in the system. Makes sense to me.

07 – Refining Relationships

Spencer, I gotta tell you, I'm impressed. You're getting very good at this. So, why don't you go wild and continue, there anything else you think we can improve here? Well something was bothering me, that what happens if there's more than one book with the same title and somebody puts in a request? Oh, I see. That's a good point. So basically what you are telling me is there's kind of a difference between an item and the title, so the title is kind of a more general concept, in a sense. So if you can have multiple copies of a given title, is that right? Yeah, we have five copies of Tom Sawyer, and the persons, the patrons, really putting in a request for any Tom Sawyer. They don't want like copy number three of Tom Sawyer, right? They want, they want to read Tom Sawyer. Okay and I can represent that. So, in which

I suggest we do that, and you can tell me whether it makes sense to you is by introducing an additional class, which I call Title. And that represents exactly the concept that you're mentioning. So this is a title which represents some specific content. That is not related to a specific physical element. Like it can be rated to multiple, physical elements. So basically I'm going to create this title. And then I'm going to create a relationship between the title and the item. And what the relationship is telling me, the the association between these two in this case. Is an association, that we call aggregation. So it's a special kind of association, that basically indicates that an item of this type, so a title can consist of a multiple elements of this type of multiple items. So it's telling me that one title can consist of multiple items, and I'm going to indicate it with this annotation, which is a this diamond at the top of the association. And so we can move our request line, up from loanable item to title, because that's what they're really requesting. Definitely, definitely, and in fact, you know, that represents exactly the situation that you are mentioning, at this point when the patron makes a request. It makes a request to a title and not to a loanable item. And then, and when the actual loan will take place, then that will be connected to a specific item. Right. Okay that makes sense. Makes sense? Yeah. Okay, good.

08 – Refining the Class Diagram

Okay, so let me see if anything changed after we did this last modification. Acutally, there is someting that I would like to do here. Because looking at this a little bit more, I noticed that there are two attributes, renewed and due date. That we have in loanable Item, right? But they don't seem to be really, attributes or characteristics of loanable Item. They're more of the characteristics of the association between the Loanable Item and the patron. Wouldn't you agree? Well, yeah, it's not like you could only renew a book once in it's entire history. Right. Exactly, exactly. So, that's why what l like to do is I would like to move those out of loanable item. And actually there is a construct that we can use to express this. It's called, we haven't seen it yet, but it's a special kind of class. It's called an association class. So, it's a class that is connected to a specific association. So what we can do here, we can create this class, which I'm going to call checked out. I'm going to, associate it with this, association. I'm going to connect it with this association. And then I'm going to move the due date and the renewed attributes From the LoanableItem here in this checked out class. So in this way, seems to me that it makes it very explicit for somebody looking at this class diagram, that these characteristics are characteristics of the loan, and not of the elements involved in the loan. Can you do the same thing with Fine, isn't Fine a property of the loan? Yeah, actually is right because a fine is a fine for a specific loan, right? That's correct. Okay, so yeah. Then we can do that. We don't need to represent fine as a class, we can just transform that into an attribute that we can put into the checked out association class. Gotcha. Anything else? Yeah. It occurred to me that there's another thing that happens in one of my scenarios, I put down about the patron actually returning an item. Right. Okay, so we would probably need an additional operation, I guess, for the patron. Right. So, okay, so what I'm going to do, that's pretty easy to do, I'm just going to add the return operation here in the patron, and when that happens, that will mean that I'll get rid of this association class because the item is returned. Is that right? Well, what happens if somebody drops the book in the book drop, but doesn't pay the, if it's overdue and doesn't pay the fine? Will that get rid of the information about what they owe? Oh, I see. So you can have the item being available, but you still want to know whether there is any pending fines on the book. Uh-huh, and how much those fines are. And how do you compute how much it is? It's how many days it was, from the time it was due, to when they returned it. I see. OK. So you know what we can do? I think we can

put an additional attribute in the checked out class and I'm going to call it when returned and that item will have either a special value or it will contain the date in which the book was returned. So in this way you should be able to keep this in the system until it's paid, and also to compute how much the fine is. Is that working? So the special value is for a normal situation when they haven't, they don't owe anything and haven't returned it yet. Exactly so that will tell us that, that the loan is still active basically. Great. Does that work for you? Yes. And you know, I like this. I mean, I feel pretty good about it. I think we have a nice class diagram. So what I'd like to do is just go off and clean it up a little bit, and put it in an IDE so I can pretty print it and rearrange things a little bit. And then I'd like to sit down again and just go through it for a last time. And for some final considerations. So if you don't mind we will take a ten minute break and reconvene here. That's fine. Alright.

09 – Final Considerations

Okay. So this is what I've done as you see, it looks a little nicer than it was before. And I didn't really change that much. I just made a few changes, so I just wanted to point them out to you, so that you know what they are. And the main thing, one of the main things I did is really to introduce these derived attributes. So these are attributes that are basically computed. Based on some other attributes. Okay, they don't have a value themselves, but their value is computed. And I used two. The first one is age. So basically we know the age of the patron based on the birthday, of the patron. So you guys, I don't know if you have that information currently in the system. No, we'll have to add that to the form patrons fill out, when they get their card. Is that, that an issue? Can you do it? No yes, we, we can easily do that. Okay, so then, perfect. So we'll do it that way. I think it's a, in a little cleaner. And similarly, since you told me that the fine was computed based on the amount of days that an item was late. The patron was late returning the item, then I also added this as a derived attribute that is computed based on the due date and when the item is actually returned. Makes sense. Makes sense? Okay. And the rest is kind of really minor things. So the, the only one I want to point out is I didn't, you know, discuss that with you before, but I added this, which is called cardinality for some of these relationships. And what they say is basically is how many elements are involved in the relationship. So, you mean the stars? Yeah, like the stars and the one... Okay. Here for example, this is telling you that for each item there is only one title. And that for each title, there are multiple items. So, star means many. Stars mean many, yeah. Okay, go you. Sorry that's kind of computer science lingo - we use the star for that kind of stuff. And, similarly, for the patron, it's telling me that, you know, each patron can have multiple, can request multiple titles, and that the same title can be requested by multiple patrons, which I think is the way the system works. Right. So except for these minor changes, we already had a pretty good model in our hands, so I think is a, we can finalize this and then just move to the low level design and then implementation, and be done with the system. Sounds good.

10 – Debriefing

So Spencer, now that we went through this process and, I'd just like to hear whether you enjoyed it, whether you think it was useful. What are your thoughts? Well, ti was very interesting. I not only learned something about computers and about how you design information systems in UML, but I, it was interesting. I also learned something interesting about the library. And things that, that I knew but I never really, explicitly written down. Uh-huh. Came up during the course of doing this. And I think I now much better understand what this information system that you're going to build for us, is really all about. Okay, well, I

mean, I'm very happy that you say that, because I really believe that, you know, doing this kind of analysis and design exercises really helps you figuring out whether there's any issues with the requirements. So for example, you can find out whether there's any missing information, or maybe conflicting information. And I think that's exactly what happened today. So I'm very glad to hear that it worked for you. That you enjoyed it. I hope you enjoyed it as well. And I strongly encourage you to do this kind of exercises for different kinds of systems. So as you can become more familiar with analysis and design techniques. So, any final thoughts? I look forward to receiving your delivered software. All right. Will do.

P2L06 Formal Specification Exercise

01 – Specification

If you want to design a program, you might want to know first what the program is supposed to do. These instructions are called specifications and they might come from the customer, the end user or the analyst team. They might pr, provide it to you in informal text, in a structured document, or using a formal mathematical notation. This lesson focuses on precise specification using mathematical notations

02 – FOL

In this part of the course, we will use two notations, mathematical logic and OCL. The brand of mathematical logic we will use is called by various names, including first order logic, FOL, and predicate calculus. You should be familiar with FOL from your undergraduate computer science courses. If you need a brush up, there's a paper on the class resources page you can look at. Also for purposes of this lesson, we will use a textual syntax for FOL that is described on another paper on the class resources page. As a reminder, FOL enables you to precisely express propositions, combine them using logical connectives like and, or, or not. And quantify them using the operators for all and there exists.

03 – OCL

The other notation we will use, is OCL. OCL stands for the Object Constraint Language, which is a part of the Unified Modeling Language, UML. OCL provides a syntax for FOL, that can be used to annotate UML diagrams.

04 – Sorting

To build up our understanding of precise specifications, we will use a simple example. Sorting. Everybody intuitively knows what it means to sort a list. Everyone that is except the computer. You need to provide a sorting program too. Imagine that we want to specify a program that could sort vectors of integers, in ascending order. Let's name the program sort, and say that it takes an input argument named X, which is a vector of integers. It returns another vector of integers Y, that was a version of X sorted in ascending order. We will begin by using First-order logic.

05 – Exercise Introduction

We'll start our specification exercise in using English. I'd like you to provide an English language description of the expected behavior of y equals the sort of x. Where x is an input sequence of integers and y is an output sequence of integers. And assuming that you're sorting in ascending order. So think how you would say this in precise English. Okay. So, given a input vector x of integers, we will return to y and output vector of integers in which for each element that is an x, the element that comes after it will be greater than that element. You

mean and why, the outcome? And why, yes, yes. So, once it's sorted, each element and why, of the element that comes after it will be greater than it, except for the last element. Well, first of all, say the input had some duplicates, is it greater than, or greater than or equal? Okay so it could be greater than or equal to if we have duplicates. Okay, and what do we do about the last one? The last one will not have a element to come after it. Right. So it will be the greatest element and the, the output vector. Or possibly equal to the one before it? Yes. [LAUGH] Ok. So there's a lot going on in, in trying to express that in a precise fashion. Let's see what we can, what we can do about it.

06 – Input Type

So, the, the first thing to consider here, is whether or not your specification stated what the input looks like. Okay? And, and Jared, in fact, said it was a vector of integers. If not, then your program would allow the vector, apples, oranges, rambutan, physalis and pepino to be sorted in a meaningful way. Those, by the way, are some weird fruits that I came across on Wikipedia. So you might say, and which Jared did given is a vector of integers named X.

07 – Output Type

And Jared also mentioned that the output is a vector of integers. And in this case in named Y.

08 – Ordering

And of course a key with sorting is that the output is in order and in particular here we're concerned with ascending order, and Jerred tried to indicate what that means by talking about the values at particular positions with respect to the position that comes after them. In the output vector. And we had to make a special case concerning the, the last element. because it doesn't have anything that comes after it.

09 – Sensitivity to Input

So now, a key thing here is that we have to also make sure of another important property. And I don't recall you saying exactly that the contents of the output has to be the same as the contents of the input. So, if we had input which was three, two, one, and we had output which was four, five, six, that is in order, okay? And it is integers. But I don't think it's what we would mean by saying that the output was a sorted version of input. Okay. So you want to try again here? Yeah, let me make sure I have what you just said, correct in my head. So, given a input vector of three, two, one. Right. We're going to map that to four, five, six. So, is that a legitimate output of the sort, what we're trying to do in the sort routine? No, it's not. So we need to say something along the lines of that for every element in the output vector y, that element exists in the input vector x. So it better be the case. We're not allowed to have that 456 there. So once again if we started with three, two, one, how about an output that is one, one, one, two, three? Okay, so we have three two one, and that's going to map to one one, did you say two three? Something like that. Yeah sure. Okay. All right so, we also need to say something then about, I guess, the cardinality, or the, how many times the elements appear in the input vector x, they need to appear that same number of times in the output vector y. Okay, so for everything in the input it's got to, for every individual item in the input, it has to show up as a separate individual item than in the output. Well how about if we then take three, two, one, and map it to one, two, three, four? So it satisfies your specification. Right, so we can't have any new elements appear either in the output vector y. It sounds like we're getting a little verbose with our original definition here. So you're not comfortable that we've

captured what seems to be a simple concept of sorting, okay, in a similarly simple expression of what the specification is. Okay, and If you think about it for a minute, sorting is pretty simple. If you imagine trying to specify the software on the International Space Station or something you'd better get right or if doesn't work there are safety issues, it makes sense to spend some time in trying to state precisely what's going on here.

10 – Circularity

And just to point out that Jared didn't run into this problem. But, sometimes in people trying to specify things they might use the concept being defined in the definition. So, saying that sort is defined in terms of sort. That would be a circular definition and you have to look out for that because. It doesn't really get you any place as far as better understanding of what the problem is.

11 – SORT in English

So here's, here's one way of kind of combining the things that we eventually got around to with, with Jared. Given is a vector of integers named X. Produced is a vector of integers named Y. The output vector Y must be ordered, in order, okay? And the contents of Y must be somehow the same as. The contents of X. That is everything in X must be in Y, everything in Y must come from X, and the number of occurrences of each item in Y must be the same as the number of occurrences of each item in X. Even that seems a little a little long and so we'd like to now take this same problem and see what it looks like when we express it in a precise mathematical notation.

12 – Process

So how do we go about doing this? Typically, when we want to do mathematical specification, we break it into three stages. The first stage is called the signature. The second is the precondition. And the third is the postcondition. Let's look at each of thee three pieces

13 – Signature

The signature gives the name of the program, the names and types of the input arguments, and the name and type of the results. For SORT, the signature looks like the following. Vector of type integer Y equals the SORT of the Vector of type integer X. That is, SORT takes a single argument named X, that has a datatype, which is a Vector of integers. And produces, produces, as a result, Vector Y, which also holds ints.

14 – Comments on Signatures

In, in specifying the signature we have given explicit names to the variables Y and X. And we do this because we'd like to be able to refer to them in the pre and post conditions by, by some, some name. Clearly this is simple, similar to what you would do in, in writing a stub in a programming language like Java. Where you can give whatever names you'd like to the arguments as long as you consistently use them. The other thing to note about this is that, we could apply our SORT program to things other than integers if they ha, satisfy one particular condition, which is that the basic elements have to be suitable as arguments to some kind of ordering operation. So greater than, less than, and so on, are ordering operations. So we can sort ints, we can sort reals, we can sort, strings and lexical graphic order if that's what we needed to do.

15 – SQRT Signature Quiz

Okay, here's something. Here's, here's, here's one for you to try out. Let's, let's give a signature for a function called SQRT, short for square root, that takes as input a real number, and returns another real number whose value is the square root of, of the argument. See what you can come up with.

16 – SQRT Signature Quiz Solution

Straight forward. Hope, hopefully you all got, got that one. So, second step is preconditions. And we're, we are talking about a function here, or functions. And the precondition will take the form of an assertion about the function's input arguments. In particular, think. Along the following lines. If you were writing some code to compute a function that took some arguments, one of the first statements you might have in your code, is something that checks whether the input arguments are what you expected. What we like to do, is in our specification, state what those conditions are. And the set of those conditions is the precondition for the function's execution.

17 – SQRT Preconditions

Okay, fair enough, now of course if instead of reels, we had complex result, okay, then we wouldn't have the same precondition, but deciding what the form of the. Output is whether it's real or complex is an important part of the specification process. For this particular exercise, we're going to go with, go with the real numbers and the precondition says that X which is our input argument is greater than or equal to zero. Notice that were not saying what happens if. X is less than zero. We're specifying the behavior of the function in terms of what does this function mean when it gets expected arguments? Now, if you wanted to have a variant. Which worked on any argument. But race and exception or produced a return code if x was less than 0. We could specify that as well. But we're going to keep it simple for now and really say we're defining square root over the non-negative reels.

18 – SORT Preconditions

Okay, square root was intentionally simple here. Let's try now giving a precondition for sort, okay? What do you think are the required conditions in order for sort to execute? I guess depending on how you might want to write your function, we should say something about having an empty, we need a non-empty set of a vector of integers. I guess you could have an empty set of integers, or an empty vector and still have integers, but you wouldn't get anything meaningful back. So it seems to kind of fall in the same category with the negative number in our square root function, maybe. Does it matter? Okay, so Jared mentioned a couple things here. One is he did say that it was important in our particular case to deal with integers. And what we're going to do there is we're going to say as far as our preconditions and post-conditions are concerned, we're not going to be concerned with the data types of the input arguments and the output results. We're going to assume that the signature took care of that. And ultimately when the program is turned into code the type checker on the compiler is going to do that kind of checking, okay? So we're not going to include in the precondition any statements about the types of things, okay? And then the second thing that Jared mentioned was, there was an uncertainty about what to do in the case of an empty vector, okay? And he said if you give in an empty vector, you wouldn't get any meaningful results back. But I would come back and say, well, isn't the sort of an empty vector an empty vector? It is. Okay, so it's a meaningful result there. Now we could have a precondition that said the length is greater

than or equal to one. And we have a perfectly good sort routine there. But we can stretch it, make it a little bit more general, by saying that the sort of an empty vector is an empty vector. And of course, all of us are computer scientists, and as such we like to have general solutions that work in as many cases as we can deal with. So in this case we're going to go with the fact that we're going to specify something that will work on an empty vector and produce an empty result. So when you factor all those things in, what's the precondition of sort? Okay, so x must be a vector, and- But that the compiler's going to take care of. We're not worried about that. Right, okay. Okay? What are the conditions on x that must hold. I don't know if there are any. That's right. And if we are not concerned about the type checking parts of things, then any input vector of integers sort should be able to deal with. So in this case there is no precondition, or we could also say the precondition is true. That is, the precondition always holds.

19 – Postconditions

So that was signature and that was precondition. The third part which is usually the trickiest one is post conditions. A post condition is also an insertion. And it says what must be true about the output produced by a function. Typically this means expressing how the output relates to the input

20 – SQRT Postcondition Quiz

So going back to our, our square root example, okay? See if you can come up with a post condition for SQRT.

21 – SQRT Postcondition Quiz Solution

I feel like this is the best I can do. Our output Y should equal the square root of X. But it looks a whole lot like our signature so- Well, not only does it look like the signature, but it's circular definition, right? Right. Okay, we haven't gotten any place. So what is the relationship, the numeric relationship between X and Y? State it in words. X times itself equals Y. Y times itself equals X? Oh, yes, sorry. Backwards. [LAUGH] So why don't you write that down? Okay. So, Y*Y=X. That, in fact, is something that must be true after square root executes. And so it's a post-condition. And in fact it is the only thing we need to worry about, okay? So it completely specifies any routine which we would believe to be a suitable square root routine. As long as the output that you get when multiplied by itself gets the input, you say it's the square root.

22 – Comments on Postconditions

So some Comments on Postconditions. First of all, we've only been concerned so far with pure functions. And a pure function is one in which the output is completely determined by the input. However, in real programming languages, computational units like functions, procedures and methods may be impure. For impure functions, in addition to describing how the output relates to the input you should also indicate any side effect. These include changes to global variables and any operations like input and output that aren't reflected in the results of the function. And if we're talking about and OO. That is, an object oriented programming language, then changes to any instance variables of the of, of the object that we're dealing with, are also things that we would have to express inside the post conditions. But so far, with square root we don't have to worry about that. And it's also the cates that, that sort is going to be a pure function, so we don't have to worry about it there.

23 – Postconditions for SORT

Okay, square root's easy. Sort's going to be harder, as far as the por, postcondition is concerned. So I'd like you to, let's go back for a minute to the natural language specification and revisit that and then consider what the postcondition of sort's going to be. So we said in the natural language specification. That the output vector Y must be ordered, and somehow the contents of Y must be the same as the contents of X. That is, everything in X must be in Y, everything in Y must come from X, and the number of occurrences must, must match up.

24 – Ordered

For every element in Y, if there exists an element that is after it, then that element must be greater than or equal to the current element we're looking at. Almost there, okay. you, what, Jarrod did is he broke the specification into two parts. One part is all the elements except the last one. And he, he stated exactly what the post-conditions is for that. But he didn't say anything about the last one. Okay, so if we wanted to have a precise specification, we'd have to deal with, with that one as well. Okay? Now, it turn out in this particular case that we can do a little proof in our head. To say that if the post condition that he specified for all the other element is true, that implies that the last one must be the greatest one. So, we could get we could get away with that, and in fact that's a pretty nice, clean way of expressing it. Notice, also, that Jared used the word" for each," and when you hear that phrase, it's suggestive of, in our first-order logic, of one of those quantifiers that I mentioned, the universal quantifier, for each or for all. And so we're going to, we're going to see when we specify this in first order logic that that, that quantifier is going to be there. First order logic when we introduce a quantifier at the same time that, or, when we use the quantifier at the same time we introduce a variable which is going to stand for the typical element of the vector. Okay, so for each i where i is going to be an index position into y, then we can say something about the value that's held in position i and the position i plus one Okay? So, the quantifier for each has a variable that comes with it we can call it i or j or whatever you would like.

25 – Elements

And so, if we're going to talk about all the elements, we'd better have a handle on how many elements there are. So, how would you deal with that? We could use some variable, I guess, to represent the number of elements in the vector, say N. That would work. Okay, and are there any limits on the value, then? No, I don't think so. Well, it better be non-negative, right? Right, right, yes. It wouldn't make sense to have a vector of negative length. So, we're going to, when we talk about all the elements, and the typical element is going to be in the ith position. The value of i is going to go up to this value of n. And, of course, with programming languages, you have to worry about whether they start counting from zero, or they start counting from one. And here, we'll say that we start counting from one. That is the first element of the vector using position one. And then, the subsequent elements go up to the last one then being in position n. So, if there are n elements in the vector, and we are talking about all but the last, that says we're talking about from position one to n minus 1. Right. Okay? And we want to have some property that's true about that. And you said that the property was that, if we look at that position and get the value in that vector, it's less than or equal to the value in the next position. Correct. Okay. So if we're going from one to n minus 1, and n is initially 0, isn't that asking us to look in the position minus 1? I guess n can't be negative, but couldn't we write it in such a way that we say, i, and then i plus 1? So, we could start at zero, and then go to one? Well, I think we have the same problem there. Okay. But,

what we're trying to do is make some statement for all i, i being from one to n minus 1. If n is zero, then we're saying from all i, from one to minus 1, and there aren't any i there. So vacuously, it's true. Okay? So, even though we run into this seemingly nonsensical situation, it doesn't actually effect the truth of the post condition. Okay? So, think about that a little bit. And given that, we're then allowed to make this quantified statement for all i, from one to n minus 1. And what must be true of each of those i possibilities? You're talking about the output. This doesn't get any easier. [LAUGH]. But we have . Okay, so y of i is less than or equal to, or no. We're going to do an i minus 1. So it's going to be greater than or equal to y of minus 1. Nope. I think you had it right the first time. Oh. If i is starting at one, okay, then we're going to be talking about and we're going to n minus 1. We're going to be going to y sub i plus 1. Okay. Okay, so you might want to change your slide there. Okay. So what it means to be ordered is, if we look at the first n minus 1 elements, any one of those, and compare it to the one next to it on the right, it's got to be less than or equal to it. And if we have that property, then we say that the output is ordered. Notice, that we didn't say anything in this part of the post condition about what those y's are and how they relate to the input. That's what the second part of the specification is. Before we move forward, is it okay to start trying to express things? This is kind of like predicate calculus kind of notation. Is that what we'll be leaning towards once we start expressing things in OCL? So, in this part of the lesson, we're going to use first order logic. It turns out that OCL is just another syntax on top of first order logic. Okay? We're going to stick with first order logic here. Eventually, I'll show you a little bit of OCL. And then, in later lessons, we'll get into the whole OCL as a language, which in addition to first order logic has some other things that help it deal with UML.

26 – ORDERED Precondition Quiz

That was the signature for ordered. Now think for a minute about what the, precondition for order is.

27 – ORDERED Precondition Quiz Solution

So, much like our ordered function that we talked about earlier, I don't think there is a precondition for this because we could take any kind of vector whether it's empty or non-empty. Okay. So assuming that we got the type checking handled for us, then as long as we have that vector of integers coming in we could always ask the question whether or not it's ordered, and expect to get a boolean results back. Does raise the question, though, of whether a empty vector is ordered or not. Have any thoughts on that? That seems like something, if it wasn't stated in the requirements for our system, we could come up with a distinction for, but- Well, our job here is to state what we intend by this particular function. We are essentially giving instructions to the developers, and we certainly wouldn't want the end user to try to use this and get some surprising results on an empty vector for ordered. So, what is your natural feeling about whether the empty vector is ordered or not? I feel like it's ordered. We don't have anything, so it's ordered. Sure, sure. And in fact, when we think about what Jared was saying earlier about all the elements are greater than, less than, or equal to the one that comes after it, well, that's true of an empty vector. Okay? So, we're going to hope that our post condition for ordered, when we write it down, will, if we plug in an empty vector, we'll get out a value of true for that. Third step is the post condition for ordered. And this is going to turn out to be the trickiest one, and, but it is a pure program. And for all pure programs what you're really saying is what the relationship of the output is to the input. And I'm going to give you a couple of hints on this one, okay? One is you're going to have to use a quantifier. And what you can say about the value of the ith element of the vector, that is a

typical element of the vector, in what can you say about that in relation to the value of the ith plus first element?

28 – ORDERED Postcondition

It written out in nice predicate logic we have what's shown here. It says for all i, okay, so i, sorry for all is the quantifier, i is the index variable. And we're going to qualify i by saying it's, it's greater than or equal to 1. And it's less than the length of Y when we put a vector inside of the vertical lines that's the, the length or cardinality of it. It must be case so the, the dot that we have here separating the two parts, you can read it must be the case. We can, use that i to pick out an element of Y and compare it to the element that Y plus 1 if in fact we have ordered output it had better be less than or equal to it. Notice that we said less than the length of Y and that gets us around the problem of trying to index into a value of the vector into a position of the vector that doesn't exist.

29 – RORDERED Spec and Pre Quiz

Now let's see if you've got it. I, I, want to, I want you to now specify a full specification in FOL for the function RORDERED, which is just like ordered except it's in, in descending in descending order. So go through the, the three steps. First do what the signature of RORDERED might be.

30 – RORDERED Spec and Pre Quiz Solution

So our signature, our name for this function is r ordered. It'll take an input, a vector, we'll say type integer call it x and it will return a Boolean value, representing whether it's in reverse order or not. And we'll call that Bool y. So it's exactly the same as what we had before, except the name of the function is different. It's only when we get to preconditions and post conditions that we're going to see a difference between them. Okay, give a crack at the precondition for this. Okay, so for the precondition, I fell like that one's the same. I think- You got it. Okay. This should be straightforward. A precondition is going to be true, or you can just leave out the precondition.

31 – RORDERED Postcondition

Now, see if you can take that quantified expression we had before. Mm-hm. Okay, as far as the post condition is concerned and play with it a little bit to get a post condition. And this one, I'll give you a hint, is going to be different. Okay, it better be different. Mm-hm. Or otherwise we're specifying ordered again. See what you can do with it. So, very much like our other post condition, we will start by saying for every element i. That of, is index of, of a integer y from one all the way to the element one less than the cardinality of the vector y. It better be the case that, if we're looking at Y sub i that it is greater than or equal to the element that- Succeeds it. Succeeds it, sorry. [LAUGH] Comes in, comes , okay? That's precision is, is what this is all about. Mm. So exactly right. In fact, the only change is that instead of a less than or equal, we have a greater than or equal, okay? And notice that we could've done this same thing with j. The exact letter that we use, doesn't make any difference.

32 – SAME–ELEMENTS–AS Signature

Mm-hm. So as I said before, the actual letter that we use doesn't make any difference as long as we're consistently use it. So we could say here bool y, we could say bool z, as long as is, it's different than those input names that we used.

33 – SAME–ELEMENTS–AS in English

Okay. And it points out a choice that you have as a specifier here. We could if we were given a a vector of length three and a vector of length 10, say no they, they don't have the same elements and remember by same elements as, we're talking about the same number of occurrence as. Or, we could treat it as a pre-condition, okay? And say that the lengths must be the same in order for us even to apply same elements as, as a function, and that's the choice that we're going to, we're going to make here. Okay, so remember that the vertical line is used for cardinality or length. And the cardinality of x must be equal to the cardinality of y. Now we're talking here about first order logic. We're not talking about a programming language. So equals means equal. It doesn't mean a sign to or anything else. We don't have to worry about anything like the C or Java where we'd have to use two equals, symbols in order to designate equality. So we're just going to say that the length of x equals the length of y, and that's the precondition for, same elements as. Now, going back to when we first asked the question to state in, in English, what it means for the output to be the assorted version of the input, we said things like each element in x must be found in y, each element of y must be found in x, and the number of occurrences the elements in x must be the same as the number of occurrences the elements in y. And we could take that and we could go through the stage of writing out each of those parts in first order logic.

34 – Permutation

Fortunately, there's a better way. We can make in, make use of an already defined mathematical construct called a permutation, that is, we can describe the same elements as in English by the following. X has the same elements as Y if the elements of X are a permutation of the elements of Y. Or, the other way around. The elements of Y are permutations of the elements of X. Because we know that permutation is well-defined, has well-defined mathematical properties, we don't have to write it out for ourselves. We could just rely on an already existing well-defined concept. Nevertheless, for the, for the purposes of this exercise, let's, let's build up a a specification of what it, what it means for one vector to be a permutation of another vector.

35 – PERMUTATION Signature Quiz

So, once again we start with signature. Why don't you try laying out, what a signature for permutation would be?

36 – PERMUTATION Signature Quiz Solution

And it makes sense. If we're trying to use permutation in place of the same elements as, that it has a very similar, or in this case, identical signature except for the name of the, name of the function that we're dealing with.

37 – PERMUTATION Postcondition

So defining the postcondition for permutation is a little tricky. Here's one answer that is worth studying to ensure yourself that you truly understand how a formal specification might look. We will break down the specification of permutation into several different special cases. Okay? And this is a typical approach for tricky specifications or even for tricky programs, is to break them down into special cases and handle each one of those special cases by itself. Okay, the first case to look at is when both vectors are empty. We already know that the lengths are the same, so if we say, we can check whether they're empty by just checking whether the

length of the vector x equals zero. And, in that case, we already talked about this, we will say that they have the same, well no we didn't talk about it. We talked about it with order. So, if we have an empty vector and we know that the lengths are the same, that's saying we're going to have an empty vector as output, is that output vector a permutation on the input vector? It is. Okay. So, special case number one, the length of the input vector is zero. We will say, in fact, that the output is a permutation of the input.

38 – Non Empty Case

Okay that was a simple case. Okay, and now have to consider the, the case in which the vectors do have elements, and we're going to also break this down into two more cases, depending on whether or not the first elements of x and y are identical. Okay? So first case is, yes they are identical. So in this case, determining whether X and Y are permutation, boils down to whether or not the tails of X and the tail of Y are permutations of each other. And the tail of a vector is everything except the first element. So we could say that, for this case, in order for the output to be a permutation of the input, the syntax on the. That you see on the slide holds. In particular that says, if the length of X is greater than 0, that's our special case. And our second special case was that the value in the first position of X is equal to the value of the first position in Y. And the third condition is if the permutation if, if the tail of X is a permutation of the tail of Y than we can conclude that in fact X and Y are permutations of each other. So what were doing now we, we handled the case where they were empty vectors and said they were permutations and in this case if these three conditions hold, they are permutations, okay? The other thing to note, is that, what we've defined here. Is a recursive definition. Now I warned you before about recursive definitions, often leading to cases where the definition is not meaningful. Now here I'm going to give you a specific situation in which you are allowed to do that. And we're allowed to do that because it essentially is, is an inductive argument. Okay? The permutations that we're using in our definition, are permutations on the tails of the input and output, okay? And the tails are everything but the first element and so they're shorter. And we also already have handled the case where we got down to 0 length. So we handled 0 length, and we handled everything in terms of its tail. We have a well founded induction here. And so in fact, our definition of permutation is in fact meaningful. We couldn't say that, X is a permutation of Y if X is a permutation of Y. Okay? That wouldn't be a meaningful induction. But here because we're shrinking it at every step, it is in fact a meaningful induction.

39 – Non Matching Case

So two cases down, one third case to handle is if in fact the first elements of x and y are not the same. To deal with this case what we're going to do is we're going to carve up Y into three pieces, and we're going to determine what those three pieces are based on where and why the first element of X is. So, the first element of X is five. Someplace in Y there is a five otherwise it's not a permutation. So let's call the position of five in y the jth position and our three segments are going to be from one up to J minus one. Second segment is going to be the jth position all by itself and the third segment is going to be the j plus first position all the way to the nth position. The last position. Three segments and we are going to define whether or not x and y are permutations in terms of those three segments.

40 – Recursion

So to state this in logic, we first have to define what j is. And we're going to use the other the, the there exists quantifier. And that's represented by a backwards facing E. So there's

going to exist some position which we're going to call j. And it's going to be greater than one. We've already taken care of the k squared's equal to one. So it's going to be greater than one and it's up to, the length of Y. And what must be the case is that the value in Y of the jth position. At the jth position must be equal to the value in X at the first position. In order to make use of this we're going to then use the three segments that we have and define. And we're going to use a recursive definition like we did before. And, but it's going to be in terms of these three segments.

41 – Pasting

The way we're going to deal with this is by pasting together the segments, leaving out the J position. In particular, we're going to say that in order for the output to be a permutation of the input, in the case where, we don't have a match in the first position, okay? And it better be the case that, the following two things are permutations of each other. First thing in our permu, in, in our check is the tail of X. That is we're going to leave off that first element. And then, we want to compare that permutation wise with the results of pasting together the first segment of Y with the last segment of Y. So, we have left out the first element in X. And now, we are going to leave out that same element in Y by pasting together the first, remember, which J minus one elements, okay? Then from J plus one to N, we're going to paste those together, leaving out the J position. And we are going to ask the question, is the tale of X a permutation of that? Now we know from our equality check, that in that J position, we matched the first one of X. We've left that out. We've left out the one in the J position. And we're now asking recursively the question about whether the remainder of X matches is the permutation of those two segments pasted together.

42 – Third Case

Okay, so it is another well-founded induction because we've shrunk the lengths of the things that we're comparing recursively. in, in logic this would look like x is the value at position 1 of x does not equal the value in position 1 in y, that's our condition here and there, what we had before about the position x so there existed j. j is greater than one, j is less than the length of y where x of 1 equals y of j and it must be the case that there's a permutation between the tail of x. Leaving out the first element and the results of pasting together y from 1 up to j minus 1 with the result, with the the third segment which was from j plus 1 through the end of, end of y. And the little funny symbol with, looks like a cap hair is pasting together and the real name for that is concatenation. We can concatenate two vectors together using this operator and if those conditions all hold then we'll, then what we're defining x to be a permutation of y.

43 – All Together

Here's the result of the entire post condition for the permutation function. Okay, we'll say that X is a permutation of Y, if and only if one of the three, any one of the three cases hold. The first case was that the vector was empty, then it's a permutation. Or the second case was if it's non empty and the first elements are the same then it's a permutation if the tails are the same, and the third case was it's non empty, the first elements don't match, and there is some place in the output vector Y, where the first element does occur, and we're going to call that position J, and it's then the case that there's a permutation between the tail of X and the results of pasting together the segment before the J position with the segment that comes after the J position. Please make sure you understand this example. It's the toughest bit of specification we'll see in these lessons. And it's important for you if you're going to be able to

express things precisely that you understand how we approach this particular problem. I think that second case was, you said the first elements in both X and Y hold, and then the remaining tails of both X and Y are permutations of each other. Not the same as each other like, is that okay? So, you got me there. And but, one of the points we're making with this is that same elements as is, the same thing as permutation. Okay. Okay. So, I did state it incorrectly.

44 – Some Questions

Okay, to check your understanding of this, let me ask a couple questions. First of all, what is the specifications say if x is the empty vector? So if x is the empty vector, then y is also the empty vector. They're equal. Okay, they have the same elements. Right? And in fact, there was a special case that checked exactly that thing. Okay? Here's a little bit trickier one. What happens if the first element of X appears in more than one place in the output. I feel like we're still covered because the way that we defined it. We know that for this element that does exist in x there's a place in y that is also that element also exists. There's at least one place. At least. At least one place and the exists quantifier doesn't care which one you've got. Mm-hm As long as you use that position consistently. If there is more than one occurrence of x sub 1 in the output, okay. Then when we do our recursive permutation check on smaller and smaller segments, we'll run into this situation again and eventually we'll get down to where we're comparing the empty vectors which we'd say are our permutation. Right.

45 – OCL

So far, we have expressed specifications in first order logic. UML includes a variant of first order logic, called the Object Constraint Language. We will be going into OCL in subsequent lessons, but for now, I would like to show you, what it looks like. Here's the complete specification of ordered. That was the first part of our sort routine, in OCL. That is the signature, the pre and post conditions.

46 – Notes

Several differences that we saw in the OCL from the First-Order Logic. It uses the vertical bar, not only, to get at the length of things, but I'm sorry. Not not as we saw in the First-Order Logic to indicate the length of things, but it serves the role as the dot we saw before to separate the part which is saying. What the, what the variable is from the remainder. The limitations on the value of i appear as part of the proposition itself. The limitations on the value of i are separated from the proposition itself, by the use of the implies OCL keyword. I've also cheated a bit, by using OCL's built in sequence class instead of vectors. OCL doesn't have vectors, it has sequences, but those are essentially the same.

47 – Summary

To summarize all this, sometimes you need a means to precisely express exactly what the functionality of a required system is. There are a variety of formal languages, and accompanying tools exist for writing such specification. Many of these are industrial strength. That means they're used in industry, and there's tool suites that come with them. Although using them requires you to think hard about exactly what you want to say. The effort can save you a lot of rework, resulting from misunderstood requirements

P2L07 OCL

01 – OCL

The topic today is the Object Constraint Language, a part of UML. So far when we've looked at UML we've been looking at diagrams. But diagrams don't tell the whole story. There are places in the specifications and the designs of your system where you need more details. And that is what OCL was designed to provide to designers. OCL is a language, it's not a programming language, it's a specification language. It's declarative, it's strongly typed, and allows you to specify the functional details of system properties. OCL consists of a means to express constraints plus some collection classes. And an ability to navigate around the various classes of relationships in your diagrams. OCL's a mature technology is part, an official part of UML and supported by various tools. Such as Rational Rose, ArgoUML, Eclipse, Poseidon, Enterprise Architect and so on

02 – Why Do We Need OCL

Why do we need, a language that goes beyond the diagrams? The diagrams are great, at describing structural relationships. We'll see with state charts, that they can also describe some behavior, but there's, there're times when you need to be more precise, particularly about the functional details. Of exactly what it means for this particular component, to do this particular task. OCL extends UML, with class in variance. With descriptions, precise descriptions of operations in terms of pre and post conditions. And they can also be used as guards on transitions in state chart diagrams, which we'll see, subsequently in the term.

03 – OCL Overview

As an overview, three aspects of, of OCL to be aware of, is first of all it's declarative. It's not a procedural language, it's not a programming language, it's a way of specifying properties. In programming language terms it's a pure expression language, that is, it describes values, it doesn't, describe activities. It doesn't have any assignment statements, instead it, it specifies assertions or constraints or properties, usually with equal signs. The language is strongly typed and it comes with some primitive types that you might expect in terms of reals and integers and so on. And the neat thing about OCL is it only has one key concept involved with it. And that's the concept of a constraint. A constraint is some formal assertion of system properties.

04 – Uses of OCL

You can use OCL for a variety of voices. You can specify invariants on classes in your class model diagrams. You can describe pre and post conditions on the operations, in your diagrams. You can specify derivation rules for derived attributes. Remember them? You can describe guards on transitions in statecharts. You can specify the targets for messages and actions, you can speci, specify type invariants for stereotypes, which we'll describe a little bit later. You can even use it as a query language. Class model diagram is describing a set of.

Possible instances, and you might wish to query those instances for certain things, OCL can be used as a query language in that sense.

05 – Syntax

Here's the Syntax. There's only one statement in the language. It's a statement which is a constraint. The constraint has a couple of keywords in it, and a couple of expressions. The first keyword is context. And it's followed by an identifier. That identifier gives a name to the context. The context is where you are in a diagram. Usually this means a class, so it's the name of a class. It might be the name of a particular method in a class. Then there comes another keyword which is the kind of constraint, and we'll see that there's three kinds of constraints. One is invariance, one is pre, preconditions, and one is post conditions. And then comes a Boolean expression, which is the actual constraint that the statement is expressing. From what I've just said you can infer correctly that OCL constraints are inherently connected with UML class model diagrams, and you probably will have already developed the class model diagram. And then gone into details by specifying the details with the OCL constraints

06 – Invariants

An invariant is a statement of a property that's always true. You can think of it as an expression of a key system requirement. Might be an essential relationship among the values of objects in, in your system. The keyword that's used to indicate that, you know, you're expressing an invariant constraint is inv. For example, you might say in large companies that the official definition of a large company is a company that has more than 50 employees. And you could express a constraint that says that in OCL by having in the context of the large company class. An invariant that says its number of employee's attribute must be greater than 50.

07 – Role of Invariants

If you're familiar with the relational databases, you may have come across the idea, of integrity constraints. And can, integrity constraints are just another name for invariants. For example, in a database, you wouldn't want to have a record for a, a person saying that they work for a company, and that company has gone bankrupt and is no longer in the database. In programming terms that's a dangling pointer. You want to make sure that your database or your system, never gets in situations where there's those. There are those kinds of integrity violations.

08 – Invariant Constraint Quiz

Here's a short quiz for you. See if you can come up in the large company example with another integrity constraint.

09 – Invariant Constraint Quiz Solution

As far as the stock market is concerned, large companies have to have a certain amount of capitalization, and here we've invented a constraint that says the market capitalization for large companies had better be at least $1 million.

10 – Pre and Post Conditions

The other two kinds of constraints are pre and post conditions. These are used for expressing precisely what it means to use an operation that belongs to some class. The key words here are P-R-E for pre, and post for post. In a given, constraint, you can have one of these or both of these. Or you could have two constraints, one with pre and one with post in it. Preconditions says the circumstances under which it's allowed that a particular operation to take place. Post conditions says what is the results of executing this particular operation. Typically that means, what's the relationship of the return value to the input parameters? However in an object-oriented language it might also mean what are the effects on any attributes of the classes that take place because of the operation has been invoked.

11 – Pre and Post Conditions Example

For example of pre and post conditions think about an operation for taking a square root. In English we might say something as far as a precondition is concerned, the argument had better be a non-negative number. As far as a post condition is confirmed something like the square of the computed results must equal the argument. That's a little bit backwards way of thinking about things but in fact it is a true expression of equality okay, that must, must be the case if square root has the meaning we expect it to have. If we were to express these particular constraints in OCL, we might do them in the context of the built-in class reel. And having, adding an operation called square root, that returns as a result, a real answer. The precondition is that the argument which is, in, in this case, is the number we're taking a square root of had better be better than or equal to zero. And the post condition is that the argument should be equal to the result when multiplied by itself.

12 – Changes to Attribute Values

The square root example has to do with specifying the properties of the results of a computation of a function. We might also consider situations where the effect of a particular operation is to change the attribute values for some class. How might we do that? Well, let's consider the example of a bank account. And has an attribute which is the current balance and has operations for deposits and withdraws. We might wish to guarantee that the balance, the current balance reflects any deposits that are made and any withdrawals that are taken out. How might we express such a constraint? Well here's an example, if we have a deposit operation in the account class, that takes a real argument. Which is the amount being deposited and as the sanity check we make sure that the amount is greater than 0, that's the precondition. We might try to express the post-condition with something like saying the balance equals balance plus the amount. However, remember that the OCL is a declarative language. An equal sign here means equality it doesn't mean assignment. So what we're saying with this as the way that it's written is the balance equals the balance plus the amount. Well, that can't be the case. All right, that doesn't make sense. Fortunately, OCL has a mechanism for allowing us to express these sorts of situations where we're changing, changing values. And that particular mechanism is, consists of an @ sign followed by the, the word pre. And what that denotes is the value before the operation executed. If we don't use @pre than what we're seeing when we express balance or deposit is the value afterwards. So we can express the post-condition this time correctly by saying that balance equals balance @pre plus amount. That is, we take the previous ba the previous balance, add in the amount being deposited and we get the new balance. Looks like an assignment segment, but it's really an equality.

13 – Post Condition Quiz

To check this out, try the following quiz. Imagine that you had a class with two attributes, a and b. And you wanted to write an Operation swap that swaps the value of the two attributes. Say you're going to do this in a post-condition. See if you can write down a post-condition that expresses that the effect of executing swap as if those two values had been interchanged.

14 – Post Condition Quiz Solution

It's even easier in OCL than it would be in a programming language. You don't have to use some temporary variable to hold one of the results. You could say simply the post-condition is that a's resultant value is b's previous value. And similarly, b's resultant value is a's previous value.

15 – OCL Built in Types

And that's pretty much all there is to the basics of OCL. We have some Built-in Types, Booleans, Integers, Reals, and Strings. We have the ability to express literals of those types. And we have some Built-in Operations on those types. So we can combine Booleans with your favorite, Boolean operators ands, and ors, and so on. We can add and subtract and multiply integers and reals. And we can we can deal with strings, we can convert them to upper case or we can concatenate them together.

16 – OCL Keywords

The entire OCL language has a small set of keywords. We've already seen invariant, pre, and post. There's an if-then-else if you need that to, describe conditional expressions. There are Boolean operators. There's a packaging mechanism that reflects UML's ability to partition things into packages. The context keywords you've seen. There's, several key words that allow you to do some definitions. Definitions can be useful to save you typing effort if define something to use the short version there's ability to indicate that your computing the value of derived attribute the derived key word. There's the ability to indicate that you're specifying an initial value, we, and we've already seen result and self.

17 – Let Clause

Let's just have a look for a minute at the let clause, which is way of doing a local abbreviation or a local definition. Say you have you a relatively complex computation that you're going to include in one of your constraints, and you're going to use it more than once. Now, you could type it out more than once, but that's extra effort and you might make a mistake. So instead, you could use a let clause to introduce a new identifier that has the value of that expression, and then use that identifier in a subsequent constraint. So for example, if our income is expressed in terms of the sum of our of our salaries for all of our jobs, okay, we could use a let clause which says exactly that. We could introduce a new variable, or identifier called income and then we could have expression, in this case is an if then else expression that says if someone is unemployed then their income is less than 100 else their income is greater then or equal to 100. It's just as if we've typed in the long expression in both the places where we used to.

18 – Navigation

I said at the start that OCL had constraints, it had collection classes, and have navigation. Let's say, let's talk for a minute about the navigation aspect. I also said that OCL typically is associated with a particular class model diagram. And when you remember that the, each of the OCL constraints has a context clause that says which class or operation you are starting with. Well, it's certainly a value to be able to give constraints on the instances of a particular class. But it's even more powerful to be able to say, that several classes are related in certain ways. That means, that in your constraints, you need to not only be able to specify the attributes of the, the context class. But the attributes of other classes, as well. Okay? Well, how do you do that? In OCL, there's the concept of navigation, which allows you to essentially walk your way through the diagram. And every time you take a step, you add the period and the name of the next class or relationship along the way.

19 – Navigation Example

Here's a diagram that involves a group of classes and we're going to assume, that our contacts class is the customer class in the upper left hand corner. Now let's say that you would like to write a constraint that involved, the date in which an order was made. Remember you're in the cu, you're in the customer class. To do that, you can use this series of steps each separated by a period. You can say self.order.date. Self is your class, order is the next class along the chain, and date is the attribute, of that particular class.

20 – Navigation Multiplicity

What do I mean by multiplicity? Well, we've seen with class models, that we can adorn the associations, with stars and numbers and so on. And this indicates, how the number of instances of one class, is related to the number of instances in another class. For example, we could have 1 to 1 associations, like spouses. Okay, we could have 1-m, 1 to multiple associations, like a parent and the parent's children. Or we could have m to n, multiple to multiple associations, like we might have between students and courses. That is, a student could take multiple courses, and a course could have multiple students in it. There's the ability in UML to express each of those possibilities by, adorning the ends of the association with numbers or stars and so on. In UML, when multiplicity is used, the result of navigating is some kind of collection. It might be a set. It might be a bag, it might be a sequence. And, UML, and OCL in particular, has a notation, that allows you to po, perform operations on those collections. The notation is a hyphen followed by a greater than. You can think of this as an arrow.

21 – Bank ID Quiz

See if you can express the navigation from Customer to BankID in order to determine the number of different banks that were used to make payments. See if you can come up with an expression that gives the number of banks. And as a hint, collections have an operation, a built in operation, called size, which for any collection will return the number of elements in that particular collection.

22 – Bank ID Quiz Solution

Here it is, self.order.check.bankID and then our pointer that says to the size operation.

23 – Collections

So we've, we've been over constraints, we've been over navigation, the third main element of OCL is collections. I've hinted at what that is. There's, there's four built in collection classes. We already talked about sets, and bags, and sequences a little bit. There's an abstract class that sits above them all called the collection class. These four classes are organized with the collection class being a parent class, and the collection class has various operators such as, size which we saw and count, and sums, and ways of interrating over the collection and so on. That are inherited by all of the three other collection classes. Moreover, those collection classes, those three other concrete collection classes may themselves have some specialized operations. The OCL reference manual has a complete list of all the operations that are available to you for dealing with collections

24 – Other OCL Features

In addition to the three main features, there's some other relatively lesser, less used features of OCL that I just wanted to mention. There's the concept of tuples. This is similar to what you would do in a programming language where you would have structs or records. There are frenzy enumerations that you see in, in Java and other languages. There's the ability to express messages. We haven't gone into this as much but in UML diagrams, you can express messages. There's access to the UML meta model. And there's the concept little word concept of automatic flattening. Say, we did our navigation, and along the way we came upon two situations, two associations which were have many participants in them. So we might end up with a set of sets, or a set of bags, or something like that. OCL has made the decision to do automatic flattening, that says if you have a set of sets, you just get one set. You don't in your syntax have to express two levels of access in order to get at the contents. This can make it a little bit easier to write your expressions.

25 – Summary

And that's it for OCL. It's a, it's a, it's a relatively simple language. There's some tool support for it. What it is, does is it gives you the ability to precisely specify the properties of your system. They're a complement to the diagrams, which can give you the structural and behavioral aspects of things. But OCL allows you to become as precise as you'd like. In order to get a true sense of what it is your system is supposed to do

P2L08 Library Exercise (OCL)

01 – Library Exercise

Remember back, several lessons ago, when we analyzed a set of requirements for a library information system, and we expressed the results in a UML class model diagram. Along the way, we were actually able to improve the quality of the requirements as we discovered several things, that weren't mentioned explicitly. But there were some aspects of the problem that the UML diagram could not express. In this lesson, we will use, the object constraint language, a part of UML, to fill in the gaps.

02 – Library Problem Requirements

As a reminder, here are the requirements. A simple set of 11 sentences that describe possible behaviors, and elements of the information system we're trying to design.

03 – Class Model Diagram

And here's the Class Model diagram we, derived from it. You, you recall, it had a Patron class, it had a class of Loanable Item, and we had to invent a Title class, in order to be able to correctly deal with the user's requests to reserve a a book. Along the way we also came up with several associations, one for making requests and the other that recorded the actual checking out of the loanable items.

04 – Requirements Quiz

As a first quiz on this, I'd like you to look through the set of 11 requirements and determine which of these the diagram is able to, by itself, adequately address and which of these requires something more to be said.

05 – Requirements Quiz Solution

So the second requirement talked about being able to express the address of a book. It's phone number, library card, how to reference it. And that information was apparent in the diagram. You could see that those attributes were supported. So two seems like it's expressed, at least already in the diagram and then I said about requirements. Okay, let's just stick with two for the time. Okay. So, typically, in situations where the requirements talk about the availability of information, such as with requirements two, the UML notation is completely adequate for describing that. You were saying three partially. Three partially. There's mentioning of, based on the functional stubs, that things could be overdue, or that you could, I believe, get the information about that. But there's a lot that needs to be done in order to fulfill that requirement that I don't think the diagram can actually do by itself. Okay, I think you're exactly right. And requirement three talked about essentially some queries being made in order to find out information about items a patron has checked out for example. And a query is a kind of operation and whenever the requirements indicate that there's an operation,

you can in your UML diagram give the signature of the operation. But the actual details of what the operation is supposed to compute, UML can't deal with that. So we're going to need OCL for that. Was there another one that you thought the diagram was sufficient for? I listed both five and nine, and requirement five talks about audio visual materials, which there is a class for that in the UML diagram. And then for reference books, I believe magazines that can't be checked out, they have classes in the UML diagram. In that sense I think they're kind of partially specify, or they're partially taking care of the requirement, but in terms of not being able to check them out, that type of functionality, it doesn't exist. Okay, so recall in the diagram, we invented some subclasses, one for things that were loanable and things that were not loanable. So references and magazines were not loanable. Right. So by the nature of the diagram we were able to express that. So that's requirement nine. Right. I guess I don't know if it was enforceable by the diagram. In terms of, functionally when you actually try to check something out, it seems like there's still something you have to do as a programmer, outside of the diagram. Okay, so another thing to look at in the diagram is the association line between the patron and it went only to loanable item. It didn't go to the non-loanable item. Okay? So there's no association expressed in the diagram that supports checking out references and and magazines. Okay, so nine I think is covered. The other one you mentioned was requirement five which talks about a patron checking out books and audiovisual materials. This is also an operation. This is going to require that the system, ultimately implemented, do some work at the time that that happens. And that corresponds to an operation, and once again, we need to say what that means, okay? Mm-hm. So for example, if we were to talk about checking things out, some record has to be kept of that. We have to be able to ensure the other requirement about the kids only being able to check out a limited number of books and so on. So there's going to be work to do there, and we need to be able to express that work. So five is something that is going to require us to say a little bit more about it. Okay.

06 – Limitations

So as far as the answer to the question's concerned, two and nine are completely expressed with the diagram. But the others require some varying amounts of work ih, to, to get precisely at what's intended by those requirements. And so to, begin our exploration of the use of OCL here, we're going to look at Requirement number 6.

07 – Requirement 6

Requirement 6 recall, says, "Books are checked out for three weeks unless they are currently bestsellers, in which case the limit is two weeks." Okay, what's happening a key that points to an issue here is that they're explicit numbers. And if they're explicit numbers then typically that means there has to be some kind of a check. And if there's a check, where you going to express the check okay. It turns out that UML provides a way to express requirements like this and associate them with elements of the diagram, and this part of UML is called the Object Constraint Language, or OCL. It's an official part of the language. It's available and supported by tools, including drawing tools that you can then annotate various parts of the diagram with, with the OCL. It provides, that is OCL provides sufficient expressive power to convey to any level of detail the functional requirements of a system because it's essentially equivalent. In power to the first order of logic, okay? So, let's see how we would express Requirement six using OCL

08 – Requirement 6 Quiz

So to think about writing something up in OCL, the first thing to think about is, which of the classes in the diagram, is the most appropriate place to annotate to make that expression. And, OCL has a key word called context. And the context is typically the name of a class or a method in a class, and in deciding that you're going to write an OCL expression, you need to first decide, which of the classes or methods you sh, you should use. Okay. So, with respect to these books are checked out, requirement, which of these classes do you think would be the appropriate one to associate with that requirement.

09 – Requirement 6 Quiz Solution

Well, that was easy. Okay, the, the, the keyword there is the first word of the requirement is books, are currently are checked out for three weeks. So our context is going to be, is going to be book.

10 – Checked Out Quiz

The second thing to decide about in coming up with a constraint is whether the constraint is associated with the system or a part of the system that is a class. Or whether it is associated with an operation. If it's associated with a system as a whole or with a class then we're going to use an invariant. Saying what must be true about the system or what must be true about the class. Whereas if it's associated with an operation, then we talk about the pre-conditions for the operation and the post-conditions for the operation and we will use the pre and post keywords to express those. Okay, in this, in this case, when we're talking about the check out period, does it sound to you more like it's an invariant or as an operation.

11 – Checked Out Quiz Solution

It holds true. It's invariant okay so yeah, we're going to say that there's an invariant over the book class, that we, we now want to express, okay? Now one. one, one thing to realize is, just like when, in drawing the diagram, there's never only one correct answer. And so we could if we wished have for the the check-out operation, we could have postcondition there which says that that, that this is true. And do it that way but it ma, makes more sense in situations where something is always true, so you have an invariant for it

12 – Requirement 6 OCL

So once we have determined the class to use and the type of constraint, we can actually express the constraint in OCL and here's what it slooks, looks like. What I've put in bold are key words that belong to the language and the parts in plain text, not bolded text correspond to the particular. Particulars of the UML diagram so we have the context is book. We have the keyword invariant and then we have a conditional expression. Recall from the statement of the requirement that there were two situations, one for books in general and the other is for bestsellers. So it makes sense that we have some kind of conditional going on there. This is a conditional expression. It's not a statement in the sense of a, a programming language. That is it produces a value rather than a change of state. Okay? And so if we read through it says, if bestseller and recall that bestseller was a bullion. Then it is the case that the check out period, which is an attribute of book, is two weeks. And otherwise, the check out period is three weeks. So, one of two possibilities exist, and it depends upon the value of that, that boolean.

13 – Explanation

What we've written expresses a single constraint, that is a single property of the system which must always ho, hold. Okay? It happens to be an invariant as indicated by the INV, keyword. What, one thing that I've glossed over is, the fact that, we use the numbers 2 and 3, explicitly. Without any indication that they're, they're dates. Now that can be inferred from the type of checkout. But if we were to get this completely right, we have to make sure that all the types matched up. And in fact, we were talking about two weeks here or three weeks here. We do have a date class but we have to make sure that we're using it appropriately. Each OCL constraint is interpreted in the context of a particular class. What that means is that any names that occur without qualification. That means without having only a single part without any period in there are interpreted in the context of a particular class. And what that means is that the name could be the name of the class, it could be the name of an attribute, or it could be the name of an operation. We're also allowed within the language to refer to elements of other classes. And in that case, we're going to have to explicitly name the class. And then put a period separator. And then the name of the attribute or the name of the operation in the other class. Those are qualified names. But if we're doing it in, if we're referring to names that belong with the class itself, we don't have to which is why we have the context keyword. This particular constraint is conditional and says that for each book object, if the bestseller attribute of that object is true, then the checkout period attribute for that object must have the value of 2. Denoting two weeks, otherwise the checkout period attribute must have the value if 3

14 – Requirement 7 OCL Quiz

Your turn. This time for requirement, 7 which is that A V material may be checked out for two weeks. Remember the series of steps that we went through. What's the, what's the context, and what kind of invariances is it?

15 – Requirement 7 OCL Quiz Solution

Okay, what do you got? So, for the context, this belongs to the audio video material class. And the attribute we're talking about is the checkout period, which would be two weeks. So, context, audio, video material. This is an invariant of that class. That the check out period should be two weeks. So very similar to the previous one. This one doesn't have to be a conditional, because there's only one possibility here.

16 – Operations

So in the, the previous two examples we were talking about the value of an attribute and those are usually pretty straightforward. Let's now talk about operations. OCL provides a way to specify operations using pre and post-condition constraints. These are different keywords in the language. In this case, we're going to look at requirement three, which describes some query operations. Now by query operation, I mean an operation that is asking about the value of an attribute, but not changing anything. So, in this case, it should be straightforward to have an operation that returns that value. So, the requirement itself says, in addition, at any particular point in time, the library may need to know or to calculate. The items a patron has checked out, when they are due and any outstanding overdue fines. So, let's concentrate on a part about the items a patron has currently checked out. So, previously when we did our analysis, we associated this text with an operation class patron called items currently checked out. What we need to do now is to say something about that particular operation. We need to

make it stated more precisely then just saying that it exists. And in fact, we have to say that the value computed by this operation corresponds to just those items that are checked out for that patron. Recall that we have a, an association between patron and loanable item. And that association is going to record what items are checked out. And now we're talking about the operation in patrons, so essentially that operation is querying the association, and we want to make sure that what the operation returns is in fact, what's appropriately expressed in the association.

17 – CheckedOut Operation

So in this case, first question to ask is if we are trying to come up with a constraint for this operation, what's its context? It is the patron class. So it's in the patron class with constraints that talk about operations. We can go a step further and say which operations. So, we're going to essentially have Patron, and in this case, there's going to be double colon, separating Patron from the name of the particular operation. And we're allowed to go one step further and actually have the signature there. Now the signature was expressed within just the UML diagram part of things. So to fill in this part of the constraint, we're going to say context. We're going to have a patron colon, colon. And then the items currently checked out. And in this case there's no arguments or at least there's no explicit argument. In object-oriented languages there's always an implicit argument of the object receiving the request. So the Patron itself. So the Patron itself is an argument and we're going to qualify with respect to it. And so there's no explicit argument listed in the signature, however there is a return value. Given the diagram, what do you think would be the type of the return value of this CheckedOut operation? It would be a LoanableItem. Okay, but the requirement says the LoanableItems, plural. Well, okay, yeah, so a list or some group of some LoanableItems. Okay. Where that association becomes, when you actually program it in. Okay, in this case because there's a plural in the expression of things, we know there's going to be more than one. And OCL provides us various, what it calls collection classes, ways of organizing things. And it's important in deciding exactly which collection class to use, to think about the properties that are required. So I use the word list and list tends to have an order. In this case do we care about the order? Not particularly, I don't think. Okay, so what other data structure is a collection class but doesn't care about order? A set or an array or? So OCL does have a set operation, so in this case the return value for this operation is a set of loanable items. Is it okay, because when you use the word set, it makes me think that you're implying we can only check out one of a certain type of book which may be a requirement, but I don't think that was explicit in the requirements statement. You can check out multiple copies of the same book, couldn't you? There's nothing about either the requirements or this OCL that prevents you from checking out multiple copies of the same book, because each of those is going to be a separate loanable item. Okay so- But what it is preventing you from doing is checking out the same book twice at the same time. Right, that's what I mean. Right. So we can't do that, even. Is that specified in the requirements, or is that just something we're now implying- Okay, so great point. So we call one of the subtleties of the original requirements analysis had to do with what happens if you check out a book, hold it overdue, you have money due on it, you return it so it's not accruing any extra as far as the fine is concerned, but the system has to remember that, right? So in a sense that CheckedOut record has to still be there to hold that information. Now what happens if you try to check it out again? If we use a set here we run the risk of clobbering the record and breaking things. So I think you've pointed out a place where we have to be very, very careful about doing this right. And in this case it looks like yes, we would clobber things in that particular situation. For operations that we're trying

to model in OCL, the next question typically is what are the preconditions. And that means the circumstances under which it is meaningful for the operation to execute. In the case of items currently checked out, as with most operations that provide a value without affecting any change in state there are no preconditions. In OCL we have two ways of dealing with that. We could have a precondition which has the Boolean value true which says it always is the case that it's okay to run this operation. Or we can leave out the precondition entirely which has the same implications. And so for readability purposes you may want to do that to make the overall constraint a little shorter. The third part in specifying operations is to specify which value is returned by the operation. To compute the items that are currently checked out, we merely navigate along the CheckedOut association to the corresponding loanable items. So here's what the overall constraint looks like. We have the context which had our signature, and then we had a post condition. And now the post condition lists another OCL keyword, which is result. And that stands in for whatever it is that's computed by that operation. And what needs to be computed in this case is those links in the CheckedOut association which correspond to items checked out by this particular patron. Now we already are restricted by our context to just the patron of interest for the query. And that patron is then going to be a partner in certain of the links in the association and we want to get the partners at the other end, the loanable items that belong to that patron. So we navigate from Patron, which is our context, through CheckedOut to LoanableItem, and that will be the set of loanable items that are currently associated with that patron. It once again leave us in the situation, what does checked out mean? It means either you currently have them checked out, and haven't returned them. Or you checked them out, you held them too long, you returned them, and the system is remembering that you still owe something on them.

18 – Explanation

The phrase checkedOut.LoanableItem is an example of a compound name in OCL. The checkedOut part is an association which is adjacent to patron, and then the qualification of that is LoanableItem which is also adjacent to the association. So it's as if we're walking through the diagram. And every step along the way, is going to be a name in our qualified name. OCL yomilla as a, in general, treats the names of these associations syntactically, just like it would treat an attribute.

19 – Requirement 4 OCL Quiz

So, that was an operation. Now, try it yourself on requirement four, which is the one that says children, and that was age 12 and under, have a special restriction. They can only check out five items at a time. So, I'd like you to try and write that and I'll give you a couple hints along the way. You're going to have to specify a pre-condition for this one. And remember preconditions indicate what must be true in order for the checked out operation to execute. And be aware that this is not going to be a complete specification for checked out. We saw part of it in the previous exercise. And there's lots more to checking things out. We're just concerned with the part of checkout that has to do with the children. So, see if you can specify for this particular part of it what the constraint is. So, before I get started I had a question in regards to, this requirement almost sounds to me more like an invariant than it does a pre or post condition of checking out. I don't know if there's a check out limit, so to speak, for patrons, but is this one of those situations in which there could be differing ways to express this requirement? So, first off, there's no check out limit for patrons in the requirements. Okay. Other than for children. And I'll disagree a little bit. This is the number of items that are checked out, is something that varies for children or for patrons as a whole. And It

varies depending upon has the operation executed? So, every time that child wants to check something out, it has to be checked. I think that it feels better to have it associated with a particular operation than its invariant because, I mean, as a statement of truth, it doesn't vary. But our need to check it is associated with this operation of checking things out. See if you can do it with operation. So, is this context the patron? Is the child association or a generalization of the patron class? So, recall when we did the original analysis, that we had a choice. We could've had a subclass child of patron. Or we could have, essentially, an attribute. And recall that the issue that arose was little Timmy, today, is 11 years old and having a birthday tomorrow. And they would change from being a child class to a full adult patron class. Now, there are ways of modeling that in UML. But, in general, in object oriented languages, you can't change the class of an instance. So, we decided, in this case, to use an attribute.

20 – Requirement 4 OCL Quiz Solution

Okay, what'd you come up with. So the context for this constraint is the operation checkout which belongs to the Patron class. And we're specifying the precondition. Well how about the arguments, how about the signature? Oh, sorry. So checkout, and then it takes a loanable item as an argument. And then you have the implied argument of the Patron itself. Gotcha. And the precondition for this particular constraint is that if the Patron's age is less than or equal to 12, so they're a child, then that means that they can only checkout up to five books. So, it's going to affect the operation of this function based on that. A couple of corrections. Okay. It's less than five, Less than five. okay, and it's not going to affect the operation, it's going to prevent the operation. Okay. It's a precondition right, and preconditions are essentially preventative. Okay, because I had some confusion in terms of how to specify because your less than or equal to twelve years old, you age is less than or equal to 12, then how's that effect this method in terms of how you express it in OCL. Okay so it prevents it. It just prevents it, okay. And ultimately the expression which comes with the keyword pre is a boolean, okay? So booleans are true or false. In actually coming up with the expression, there are several ways to manipulate the logical connectives on it to get them. The one that's here is that it's in the form of an implication. There's an implies keyword in OCL, and we're saying that if the age and this is the age attribute of the particular Patron, okay, is less than or equal to 12. In this case the 12 is for years not weeks, okay. That then implies that it must be the case that the size of the collection of items currently checked out must be less than five. So a couple of things to say about that. There is an operation, items currently checked out. And remember that it returned a set of loanable items. Right. Now the set is a collection, and collections themselves, collection class, collections have operations. Mm-hm. And one of the built in operations is size. So this is cardinality. Remember we had the in first year logic, we had the vertical bars on things. So this is saying that the size of the items currently checked out is less than five. Two more questions before we move forward. Is the If-then syntax even supported in OCL or if you're going to say something that's got that If-then structure which is like an implication statement, you're just using was. So, we actually had an if-then a minute ago. We had if the book was a best seller, it was only two weeks, and else it was three weeks. Right. Okay, so yes you can have the if-then. The difference is a technical difference having to do with the order in which you check things, okay? So, if you have here a implies, you're first checking the age is less than or equal to 12, okay? And then checking the other part of it which was less than five. Okay, and then the second question is, or it's not really a question I guess. I understand it more now that you say a precondition is almost like if this does not hold, then the method cannot execute. So therefore, when I was looking at the original requirements that

children can check out to five books, less or equal to five was. I guess that was me kind of falling back and saying okay it's less than or equal to five that's how many books I can check out. But prior to this method executing, it can't have five books because then they would go to six if it did execute. [CROSSTALK] An interesting point. If somehow magically, they had checked out 50, and they came and tried to check out the 51st, this would prevent them from doing it. Correct. Now, you can then ask, how could they have gotten 50? Okay, and you could then prove to yourself that they couldn't have gotten to 49, to 48. They couldn't have gotten past four, right? Correct. So this particular constraint is not saying how many they have checked out explicitly, but you could use it to prove that they could never have gotten above four. Okay. Okay, so I guess what we're talking about here is the inner play between an invariant statement that they never have more than four checked out, and the precondition on the operation of which is preventing them from getting in a state which violates that invariant. So wait, the invariant though is thick would never have more than five checked out. What's going on here is that the precondition says that they have to have less than five. Right. And then they can, the checkout on the operation would allow them to add one more, and that would get them to five, so you're right. Okay, okay.

21 – Requirement 4 Explanation

There are several noteworthy features of this constraint. The first is the use of the implies keyword to denote logical implication, which is just a way of saying the restriction is only true for children. It is equivalent to the use of the right right arrow in first order logic. The second feature is the, the OCL use of the right arrow, this one with only one horizontal line. And that's used with collection classes. So in the case where there's a collection on the left hand side and you wish to, access or make use of one of the built in operations in this case size, you can use this right arrow to say you'd like to bring in the size operation on this particular collection class. The other types of collections that we'll get to but all of them have a size operation associated with it

22 – Side Effects

So, so far the two operations that we've looked at have been query operations. That is, they've been asking about the values of attributes. However, in interesting information systems, in addition to querying, you need to have operations that actually change the state of things. And, so this is going to be ultimately implemented with the database and you need the ability to add records, to change records, and that sort of thing. We call particular operations that don't make any changes like this pure, as in pure functions. Because they're similar to mathematical functions, which always compute the same result. Functions which are impure, are said to have side effects. And side effects might be changing the values on the database. They might be doing IO operations, input output operations. Or flashing something on the screen that the user sees. what, whatever it is. So, those are side effects. And, so, now, as the next, next step in our, modeling of these requirements, let's try to specify a more complex situation. One where an operation actually results in a change of state. In this case, we choose to model the actual process of checking out a Loanableitem as indicated in requirement five. A patron can check out books or audio visual materials

23 – Requirement 5 Signature

So first off, what might the signature of this look like? So I think we're still working with this check out method, but now we're specifying a post condition of the method. So the signature is still patron, checkout, a loanable item as as an argument and the implied

argument of patron, but the the post condition is what we're worried about. Okay, so that raises an interesting point. So we just had an example where we were expressing some of the properties of the check out process. This case, having to do with kids. Okay. In fact, you could have any number of constraints about a particular class or a particular operation. What they mean is its as if you were 'anding' all those together. So, this must be true, and this must be true, and this must be true. Well, the thing about the kids must be true, but it's only going to be true about kids. Okay. So it makes sense that you can, what the designers of OCL have done is they've simplified the syntax by saying you don't have to do all those ands if we just have the constraints listed there we can focus on the ones that we want to focus on. So, yes, in this case the signature is going to be exactly the same as what it was before. So how would we express, or could we express multiple pre-and-post-conditions? Is there an operation to string some together or would we need to make multiple constraint statements for the same method. So you can roll your own. So let's say we had three pre-conditions. Okay, you could have them all in the same constraint with 'ands' between them, or you could have three constraints, each with one precondition. Okay. Getting the same thing out of it; the same thing on the post condition side of things. Likewise, an invariant is something that must hold true. If you have five invariants, they all must hold true as if you have 'ands' between them. Okay. Okay, so that was the context in the signature for checked out in this case.

24 – Checkout Preconditions Quiz

Now we're talking about adults. So, the question for you is, there are various preconditions that must hold in order for you to check something out.

25 – Checkout Preconditions Quiz Solution

What do you got? So, I think the first one is a little bit more obvious, or at least to me. You can't check out a book if it's already checked out, it has to be availed to be check out. Or it might be back, being rebound. Sure. Pages came loose and they're being reglued or something like that. So it has to be available. What else? And then also, I could just imagine a scenario that you walk up and you're trying to check something out and they say, well, you have some outstanding fees that need to be paid before we let you check out anything else. So you run a tough library. [LAUGH] My library, they say you want to pay the $0.10 now or hold off on it? And they make me pay it but I can hold off on it.

26 – Requirement 5

So the, these three conditions that Jared has come up with about whether it's available, whether it hasn't been requested by somebody else and the one that we had before concerning age correspond to three conjuncts. You know, three, three possibilities that all must hold. And as we talked about a minute ago we could have three separate preconditions, or we could have one precondition with three, with, with two ands separating the three parts of it. The other thing to note in the expression of it is. We've use some new operation names, in expressing it. So we said, is available. Now, is available is not one that came out of the process of building the [UNKNOWN] class diagram. But that's okay. It's like. Breaking the writing of a a method into, into pieces and calling other methods along the way. Okay it's, it's just divide and conquer. So, we're going to assume we can invent the names of these convenient operations. And, that simplifies the, not only the writing here, but the reading as well, if we're trying to show the requirements to the customer.

27 – Checkout Postconditions

So that was the pre-conditions and as is usually the case the pre-conditions are easier than the post-conditions. So we want to next look at the what it means, assuming that we've past the pre-conditions, to actually check something out. What affect is that going to have on our information system. Okay, so take a minute and think what you expect to happen when the item is checked out and list them in English. So I think my mom was the obvious post condition is now the book's checked out. So a way to say that or express that I thought wouldn't go back to that item's currently checked out operation that belongs to the patron class and say that the set that's returned from that operation should contain this new loanable item that's been checked out. So, this raises an interesting [COUGH] issue. So we're specifying an operation. And we're specifying one operation in terms of another operation. Okay?. And we can do that, that's fine. However, the real change of state here has to do, so the items currently checked out. Is a query operation that reflects the state. We could more directly say what's the change in the state, okay? So what other post condition do you have? So the other condition I have I think is more related to this, I guess, is that a checked out, we have that, a class represented on our email diagram. There has to be an instance of that now for this particular operation that has just occurred. Okay. So we have the association class checked out. Yes. And [COUGH] the effect of checking out something new would mean that there's another link in that association class. Yes. Instance of that. And, if there is another link there, then, by implication, the items currently checked out will return that new item. Correct. So we don't have to state both of those. Okay. We only have to say that the association has one more thing in it and we get the other one for free. However, there's another post condition. Recall the business about requests. Okay, so what would the post condition there be? So, the request queue is now one less than it was prior. Someone has such it popped off the queue, because a checked out operation has occurred. Well, it had better be you, because you're doing the checking out. We can't just get rid of an arbitrary person. The implied patron should be the one who's- Right, okay. So, we already have a precondition which says that this particular patron is eligible because either there's no request in which case there's nothing to pop off. Or if there is a request out there, the first one on the list had better be you. So in that latter contingency we do want to pop off your name, okay from there. So we're going to state the post condition but it's going to be a little tricky to state because it's only in the case where you were on the list. Right, and it sounds like an implication because we just had like another only if the case of it's a child this happens. So only if the case that you were the patron on the checkout request list. Bingo. Okay, cool.

28 – Further Checkout Explanation

So, those are examples of precondition. There's another one. Think for a second and see if you can come up with one other potential precondition. So, for the last requirement, you could be in a situation where someone continually is trying to check out the same vulnerable item over and over again and one of our requirements states that you can renew an item once once. But, if there is another outstanding request for that item, then you shouldn't be allowed to check it out because other people may be wanting to check out the same thing. We're talking about checking out an item. And if you've all ready got it checked out and you want to keep it longer, that's a renewal, not a check out. But clearly, we're going to have to deal with that situation when we get to considering renewals. So, the other possibility is you'd like to check this out, but it turns out that it's the hottest bestseller, and there's 44 people before you in the list of requests. So even though you happened to find it, because it was just put back, it was put back by mistake and they're not going to let check it out because somebody else

has requested it. So, the request queue needs to be empty so to speak for you to be able to. Well, almost. How about if you're the person that requested it? If you're the first person that requested, then the precondition should hold and- Right. Execute. It's not am empty queue. It could be a queue with one thing in it or a queue actually with any number of things in it as long as you're top on the list. Okay. Okay.

29 – Postconditions

So there's going to be these two post conditions, one having to do with the new link in the association and the other one having to do with contingently updating the request list. So we'll consider the first post condition here. What we're really saying is that there's going to be a new CheckOut record. Another way of saying that is there must exist a new CheckOut record, and in fact, there's that keyword in OCL called exists. So, the post condition here is going to say there's going to exist, and this is just like in our first order logic where we had the leftward facing capital letter E. That there's a bound variable in this case we'll call it C. So there's a new link in that association, the association's name checked out, okay? With the properties that the that-links-loanable items had better be the item that we're checking out which was the argument to the operation. And the properties that we wish to hold about that CheckOut, that is the due date, had better be correct. And to get the due date, we can go get today's date, and then we can add in the checkoutPeriod. And remember, we have the checkoutPeriod, that was one of the first requirements we dealt with in OCL. And we can go get that checkoutPeriod by navigating from the CheckOut link to the corresponding loanableItem, and then getting its checkoutPeriod. That says there must exist this new item and it also better be the case that the checkedOut association itself must have a new item in it. And the way we do that in, either first door logic or in OCL, is to make a statement of the quality between the state that existed before the CheckOut operation, and the state that existed afterwards. So the state that exists afterwards is just checkedOut. You can think of that a set of records or a set of links. But need some special notation to deal with the state of the checkedOut association beforehand. And OCL has a keyword, and it's indicated by the @ followed by the word pre. So this is saying checkout@pre means the version before we call the operation, and CheckOut by itself means the version afterwards. And there's an operation on sets, and checkedOut is a set, it's a set of links, that allows us to indicate that another element is associated with that particular set. And that keyword is including. So it's as if we are constructing a new set, which is the old set, plus this one new item. Actually this one new checkedOut link, which we've given the name C. So, we have the checkedOut afterwards is equal to the checkedOut before, and this one other item which we've just added, and that's what's being expressed on the bottom line there. So we're making use of some special keywords and operations in the OCL language to express the equality of these two states. So I've got a quick question. Can we also consider the @pre including kind of be like set union, so we're unioning the set of all of our checkedOut associations plus this new checked-out association. Almost, but this item, I'm sorry, this new checkedOut link is not a set. Its just an element. If we had somehow been able to wrap it in some set thing, then it would be unique. A set containing this new element. And what I think Jared is pointing out here, is we have to be careful in getting the data types right in these things. So, a set is different than an element and if you had a union operation you couldn't put an element on it. And if you had including you couldn't put a set on it. That was the first post condition having to do with updating the checkedOut association. The second one had to do with the requests that were made. We had a precondition that said if there are requests then it'd better be the case that Jared is the one doing the requesting. Now we need to update that request list if it exists, and with the fact that we want to take Jared off that

list. We don't want him to hog the book too long. So, in this case, the post condition is going to make use of another feature of OCL called the LET expression. And the LET expression is just an abbreviation. It allows us to state in one place, an expression, and then use it in several places. So in this case we're saying let the new variable t of type Title be that title of the current item. We distinguish between items, which are things that could be checked out from the things the things that could be requested, which we called titles. But an item always has a title, that's the composition operator that we see in the diagram. And we just want to use the shorthand of t for item.title to save ourselves a few characters every time we type. So the let expression there defines the new symbol, and binds it during the scope of another expression, which is the post condition. And the post condition here is conditional as to be expected. So if that particular title has a request to this particular patron. And then we want to update the request association. So the antecedent of the conditional is t.request.patron and if we didn't have t, we'd have i.title.request.patron, it'd be either three or four steps in the chain of qualified name and includes is a operation on that set of requests. And self is the OCL keyword corresponding to the instance of the particular class. And remember that was way back with the Patron. So we're seeing whether or not Jared is in the list of people during the requesting. And if so, we want to do another one of these statements of equality between the state before and the state afterwards. So, the state afterwards is the request association and the state before is request at pre. And, in this case it's not that we're adding in a new record, it's that we want to remove it. But, it's a statement of equality so were stating that by saying If we, and the key word in OCL is reject. If we reject those particular items in the list which have the following boolean. And the boolean is, the title is t, and patron is self. Then we have a version of the request association without that one record in it and that had better equal to the one afterwards. If there were no requests with Jared at the top of the list, then our result is true. Okay, so we don't have to be concerned there. So the overall post condition constraint has the form of the let binding, and then within that a conditional. And the conditional, we're only interested in the then part of it, which is qualified by whether or not Jared has done the requesting. And in which case we want to ensure that Jared's request is no longer in In that particular association. So do you have a recommendation on this if then statement? because this if then statement, the else is kind of a vacuously true situation. So wouldn't it just be better to use an imply statement to save yourself some writing? Good point. So yeah, we could shorten the situation by doing the implies. I had on other question. Sure. If that's all right. So I guess, are all associations that are modeled in our UML diagram, can they be thought of when we're expressing things in OCL as sets. because it seems like express t.request.patron, this is an association. We're talking about the patron part of the association. The set of all patrons that have requested things, or in which a request instance is there, is modeled in a set. Okay, so you can think of them as sets, but sometimes we can do a little bit better. So we call that in UML, those the ability to order things. And in order things, then we get sequences. So sequences give us a little more power because we can index into them. Okay. Whereas with sets you can't index into them, right? So, now a sequence is a set, but a sequence, because it has this ordering with it, allows us a little more power. And that's going to be based on the model that we have, whether we modeled it that way or as a set. The properties, the requirements, and whether we modeled it that way. Okay.

30 – Derived Data Quiz

So far we have seen how OCL is used to specify invariants and operations. There is another part of our analysis model that OCL can help with. That's derive data. There were two places in the [UNKNOWN] model for the library, where derive data, was used. The Patron's age and

the amount of the overdue fine. Remember that derive data are, they're attributes, like any other attributes but, they're going to be computed along the way, rather than something that is a, is a set piece of data. Okay? And the, the situation here had to do, for example, with the, Patron's age, which is changing on a day to day basis. So, at any moment when we need the age, we compute the value. Okay, so we need to say what value is being computed. And the same, the same holds true for the amount of fine which changes on a day to day basis, and depending upon how long your book is overdue. So let's, let's do a little exercise here. See if you can come up with a constraint in OCL or a Patron's age. And as a hint here, there's a keyword in, in OCL called derive, and you use that instead of pre or post interim variant. See if you can express it.

31 – Derived Data Quiz Solution

What do you got? So, like all of our other statements, we start with context, patron and then talking about the age attribute right now. So this is the first time we've seen a situation were the context has to do with an attribute. We saw it where it could be the class as a whole, or we saw it where it can be an operation. You can also have it association with an attribute, and the attribute here was age. What else? And then our derived key word followed by, I wrote currentDate minus birthDate. So, birthDate is something that we have access to based on our diagram, but I wasn't quite sure how we would get our currentDate. I'm assuming you might have a date, within some software systems there's usually some way to get the current date so. Okay so clearly any information system is going to live within some context in which there's going to be libraries and system calls and that sort of thing. So, we're assuming here that there's some class which I've called Operating System and it's got a operation associated with it called getDate and we can then compute the difference between those two. And, the difference will be what we call the person's age. Now notice that we're also finessing some details here. In this case, the subtraction operation has to work on dates, and we often, when we say age, think of age in terms of years. We did that for the amount that could be checked out. But the subtraction between dates might be years and month, and days, and sort of thing. We would have to get that right depending upon how we're going to use dating within the system.

32 – Missing Pieces

This lesson has illustrated some uses of OCL to provide precise specifications for simple library information system. Even so there are many more things we need to do to complete this exercise. We haven't even mentioned some of the other requirements like numbers one, three, eight, ten, 11 which we would need to specify. We haven't of course at all in this whole exercise mention anything having to do with non functional. Requirements. Along the way we invented some axillary operations like is available, and is available is actually pretty complicated itself and we'd have, we'd have to white that out. We'd have to handle some situations implicit in the requirements such as returning a loanable item, as returning a book we checked out. Paying a fine, cancelling a request, and so on. And, we might have to consider some new some new issues that arise during the course of doing this. For example, if one item in a title is a best seller, need all of them be best sellers, okay? It would make sense in a library if you have some. Designated a book as a best seller that all the items in that title, are a best seller. But we haven't stated anything explicitly that would require that. We would also need to op, add in some specifications for operations for classes date. And money. Money had to do with paying fines. Although in those situations we could presumably reuse those specification in other systems that we're building. Likewise, for the operating system operations. We also

haven't said anything about constructing instances of loanable items and patrons. Although, we did construct instances of associations and we can use similar techniques for constructing h, the instances of the other two classes.

33 – Observations

So, some observations about this exercise. Be aware that there may be more than one answer. And we, we saw some instances of that. Be open during the process of analyses to the possibility of new requirements arising or that there's ambiguities. Or infact, that there, mistakes have been made. And this may mean for the consultation with the customer. The fact that a simple set of requirements can have so many issues, should illustrate the value of performing this kind of careful thinking that is but the kind of careful thinking that is required in order to construct an OCL specification.

P2L09 Behavior Modeling

01 – Behavior Modeling

The structural models that we have been looking at so far express properties of systems that are true at all times. Although, these models are general, they fail to convey interesting behavioral aspects of the systems. That is how the systems respond to external stimuli. UML provides a variety of alternative diagrams that do support the behavioral modeling of systems. We will look at these and later we will also go into detail on the state chart diagramming technique. Which is the one which provides you the ability to. Precisely describe the system properties.

02 – States

First, let's start with two key concepts, states and events. Both of these are abstract but nevertheless useful in describing system behavior. A state is an abstract description of a set of system values at a given point of time. For example, it's raining outside. This is actually some estimation or abstraction over the amount of precipitation in the air at a given point in time. Let's imagine a slightly more complex situation in the imaginary town of Des Cartes Iowa, that has ten streets and eight avenues laid out in an orthogonal grid. And let's say we were trying to imagine the state of the town that had one car in it. The position of that car in this imaginary scenario is at third street and fifth avenue. The number of possible states it could be in is 80, ten streets, eight avenues. For two cars though, the number of possible states, goes up with a square. That is, 80 times 80 or 6400 possible states. The State Space of a system, is a set of possible states, and its size goes up, multiplicatively, with a number of different attributes that we're trying to capture. And the number of possibilities, that each attribute has. This is called the state explosion problem.

03 – Tic Tac Toe Quiz

How about the game of TicTacToe? This is played on a 3 by 3 grid, and each of the grid cells can hold an X, an O, or can be blank. How many different states can a TicTacToe board be in? Forgetting for a moment,whether or not those states are legal states as far as the game rules are concerned.

04 – Tic Tac Toe Quiz Solution

Well if you do the multiplication it comes out to be 3 to the 9th or 19,683 possible states of the board. As you can see it went up rather rapidly.

05 – Events

The second key concept is the event. Once again it's an abstraction, and we'll say it's a single, instantaneous, noticeable occurrence. And also, and also, think of it as some kind of stimulus because the system is going to respond to it. Events in these kinds of systems

can either be asynchronous or synchronous. Asynchronous events you can think of as kind of randomly occurring, they can come in bursts, they can be spread out. Synchronous events are more likely to come at periodic intervals, often the system has some sort of clock or event loop that controls. The current events when they can be dealt with in our approach to modeling these systems the events can serve as the reason or the stimulus for a change of state in the system. That, that change of state is sometimes called a state transition. For example, when you got your envelope through the mail that said you were admitted to college, that's a significant state transition in your life. Events can occur as the results of user actions like hitting a, hitting a button on the, on the graphical user interface. They can occur because of changes in the data, in the, in the state space the temperature getting above 90 degrees. Or they can be queued by the passage of time.

06 – UML Event Taxonomy

UML supports, several different kinds of events which you can use in your modelling. These include, signals, which are asynchronous notifications. Method calls, which are, which are synchronous. State changes in the data. Also called data conditions, and the passage of time.

07 – State vs Event Quiz

Okay, here are a list of descriptions of situations. And for each of these, determine whether it's a state, an event, or neither. Jesse Jones won the Olympic 100 meter in 1936 in Berlin. The color magenta. A building's sprinkler system turning on due to a fire. Your telephone ringing. A rainy day. Big Ben chiming. The International Space Station. The screen saver on your laptop turning on.

08 – State vs Event Quiz Solution

For Jesse Owens, that was an important state in his life. It was a state as far as the Olympics are concerned. And it turned out, it was an important state as far as world events were concerned. The color magenta, however, is neither. It's not an event or it's not a state. It's just the value of an attribute. The building's sprinkler system turning on due to a fire. That's an event, an instantaneous change of state. Telephone ringing is also an event. Rainy day is a state. Big Ben chiming, now I know it takes a long time for it to chime, but let's just think of it as being the start of chiming, that's an event. Once again, an abstraction over that whole sequence of chimes. The Inter, International Space Station is neither. You can think of it as a very, very, very complex object itself. And the screen saver on your laptop turning on, event.

09 – Modeling Techniques

With these two concepts, states and events in mind, let's think a little bit about modeling of behavior. Systems that respond to events are called reactive systems. If you think about it for a minute that's much different than this other kinds of systems that you would normally build, in which it's the system in charge of controlling the order in which things happen. Here is the external world which supplies the stimuli that are causing things to happen in the system, the system has to react to them. In general there are a variety of different approaches to modeling behavior. We're going to go through from the simplest, which is combinatorial systems. Through sequential systems, to the most complex and concurrent systems. In combinatorial systems, we're just concerned with states no events. Sequential systems have states, that is they have memory, but they're linearly ordered one state after another. And then concurrent

systems. Particularly asynchronous concurrent systems have lots of states and lots of events and the events are happening at unpredictable moments of time.

10 – Combinatorial Modeling

Starting at the simplest combinatorial modeling, this is the simplest form of behavioral modeling, and it merely expresses the logic of simple combinatorial systems. In these systems, only the inputs and not the history of previous states determines subsequent states. We'll look for a second at two equivalent forms of combinatorial modeling. Called decision trees and decision tables.

11 – Decision Tables

Start with decision tables. This is a common way for, getting a, getting your head wrapped around a situation where there are various different, states that can affect ultimate behavior of the system. If you, if you think about decision tables in terms of input conditions and and, and, and responses to those inputs. That is combinations of inputs yielding results. That's where the term netwire comes from. The table will have columns. Some of which correspond to the inputs and the remainder will correspond to the outputs. And then each of the rows is going to correspond to a different combination of input values. Let's imagine that we have a workshop and there are three switches, okay. Each of the switches can have on and off as possible values, okay? So we're going to have three columns and eight rows. Where did the eight come from? Well, if we have three switches, two possible values for each switch, that's two to the third or eight. Let's assume that the three switches control two output devices. Maybe an overhead light and maybe a, a power drill. The third switch is a master switch which controls all the electricity in the workshop. Well here's the, here's the decision table for this situation. Partitioned the columns into three input columns for the three different switches and two Output columns for the lights and the motor on the power drill. Each of the rows correspond to one of the possible combinations of the values for the input switches. Due to the third, eight different rows there. For each of the combinations of the inputs there'll be a resulting situation as far as the outputs are concerned. So for example, if the master control switch is off then it doesn't matter the positions of the other switches, both the lights and the motor will be off.

12 – Decision Trees

A graphical version of the decision table is called a decision tree. It's a form of a flow chart in which the decisions are taken sequentially and the resulting output can be seen as a path through the tree. It's exactly the same information as in the decision tree, but you're seeing it in a different form. In the decision tree we'll see in just a second, there are two kinds of nodes. Diamonds denote decisions and rectangles denote the actions that are going to be taken based upon the decisions that are made. The arcs in the diagram indicate the implications when a decision is answered in a particular way, either affirmatively or negatively. Note that in the decision tree you're about to see, some of the nodes have been duplicated. This is a side effect of the redundancy which occurs in the table as well. Here's the tree for the previous situation. On the left is the decision about the master control and the on, on the rightmost is the resultant response from the system in terms of what lights are on and what lights are off. Once again, same information is in the decision table. Note that the two rightmost diamonds contain exactly the same question. And that's it for combinatorial combinatorial logic. As you can well imagine, as the number of possibilities for the input goes up, these tables quickly become unmanage, unmanageable.

13 – Sequential Systems

So then let's move to the next most complex version of behavioral modeling called sequential systems. In sequential systems and concurrent systems that we'll get to, the main difference from common [UNKNOWN] systems is that there's history or memory of what happened before. You were in a previous state and based upon that state and whatever events occur you move to a new state. Systems like this are sometimes called finite state systems because we're going to limit the number of states that they can have. To a finite number. Okay, and if you recall from your theory course we're going to take advantage of finite state machines as a way of doing the modelling.

14 – State Transition Table STT

We can represent these finite state machines in a variety of ways. Let's start with a tabular form called a state transition table. Here the rows correspond to states and there are four columns. One column for the name of the state. Another for the input event which is going to cause a transition. The third for whatever output action is going to be taken upon the transition, and the fourth for the next state. That is, the state transition table is going to capture the idea that a system in a given state, when given a certain stimulus, and when a stimulus occurs. Is possibly going to produce some action or response, and leave itself in a in a state, possibly the same or possibly a different state. To see how this works, let's imagine a garage door opening system. In fact, a, a simplified version of my garage door opening system. Okay. It's gotta motor and that motor can be lifting the door up, it can be pushing the door down, or it can be stopped. There's a button that can be pressed to turn the motor on and off. The door itself can be all the way open, all the way closed, or stopped at some intermediate intermediate state along the way. Importantly, the, is what happens when you press the button, and in this particular garage door situation, okay. What happens when you press the button can be one of three things depending upon what state you're in. If the motor is stopped and you press the button, it starts going but in the opposite direction it was going before. If the motor is going upward and you hit the button, it stops, leaving the door wherever it was at that time. However, for security or safety reasons, if the motor's going downward and you press the button, it not only stops but then it immediately turns on and goes in the other direction. Presumably because something was detected that might be damaged by the door going down on it. And implicit in this is some sensors. One sensor determining whether the door is all the way up, and another sensor saying whether the door is all the way down.

15 – Garage Door Quiz 1

Okay, take a second and try to figure out how many different states this system could be in

16 – Garage Door Quiz 1 Solution

If you do the, the, the work on this there are six different states. It could be open with the motor off, it could be closed with the motor off, this is probably the most common state, it could be, stopped that is the door could be stopped, the motor could be stopped partway, up. Partway down, the door could be moving with the motor on in the downward direction, pulling the door up, upward. Or, the door could be partially opened because it had been closing and the button was pressed to make it, start moving upward. That is the motor is temporarily off. Six possible states for this system.

17 – Garage Door Quiz 2

Second part, of the quiz, how many events does the system respond to.

18 – Garage Door Quiz 2 Solution

Of course it's going to be responding to the button presses, but don't forget that it's also going to be responding to the sensor notification that the door is up or the sensor notification that the door is down. So, three different events

19 – STT for Garage Door System

Here's a state transition table for the garage door opener. Notice that there are eight rows, but only six states. That's because some of the states have two possible, transitions on them, depending upon the events of their arriving. Second column is the input, as to which of those events there are. Notice that the third column has the actions of starting and stopping the motor. And the next state is in the, in the fourth column. Well, this table can obviously get crowded as the number of possible states and events goes up. So we'd like to at least explore the alternative of some kind of graphical view of it

20 – State Transition Diagrams

These views are called State Transition Diagrams, they're essentially represent, graphical representations of a Finite State Machines. In these diagrams we have an indication of a node, typically with an oval, or some kind of rectangle. We have arcs, directed arcs connecting the nodes, indicating that there's a state transition. The arcs can be labeled. The arcs can be labeled with an action and a transition. Typically the actions can be can be optional. Usually, but not always, the transitions are, are not optional, okay? If you recall your finite state machines from theory, there are these epsilon transitions. We'll see in fact that the garage door opener has one of these, but they're, they're not, they're not all that typical. Note that the layout of the nodes, where we placed them in the diagram, doesn't have any semantic import. So you are free to make the diagram lay it out in a way that conveys what you consider to be the important behavioral aspects of the system. In terms of how these diagrams work, you can think of yourself as, at any point in time, being in a specific state. Okay, kind of waiting there until an event happens. When the event happens, the outgoing arcs are examined to determine whether any of them are labeled with the corresponding event. And if so, a transition is made between the current state and the state at the end of that transition. And in so doing the action, if any, on that transition arc is, is executed.

21 – Example Garage Door

Here is the graphical representation of the garage door opener. I've used rectangles in this case, but there are six, six states indicating the six states we, we listed before. The the arc, the transition arcs have two two expressions on them. The first one, the first is the event causing the transition, if there's then a slash, the second one is the event to take, that will happen upon that transition occurring. So imagine, for example we're at the bottom, in the state labeled Door Open Motor Off. And in this case, there's only one outgoing arc. The user has pressed the button and, in this case, the motor starts and it moves into the state where the motor is running downward. The door was open, it was at the top. Hence, the only place that the door can go is downward. Similarly, you can walk yourself through the diagram. The interesting situation involving epsilon transition takes place if you are in the bottommost state on the right, labelled Motor Running Down. If, then, the button is pressed, okay, recall that

the first thing that happens is the motor is stopped and we transition into the Door Partially Closed Motor Off state just above it. But when we go on, okay, remember the, the safety concern. The door was going down and we stopped, and we now want to take it up. So there's a transition to the left in which there is no event causing the transition. Okay? So immediate transition and what we want to do is start the motor going upward and end up in the state at the extreme left labeled Motor Running Up

22 – Example Telephone

Here's another example, a graphical example, of a state machine, using slightly different icons on things. In this case, ovals for the states. We have the telephone being off hook, we have it when you're dialing or pressing the buttons. We have it when it's ringing, when it's busy when you are connected to another party. And, when it's in a rest state on hook, on the extreme left. Notice in that case, that there are two ovals nested inside of each other. This is used to designate what the default or start state of the system is. There are then transitions, these directed arcs going, among the states. One to look at is the one on the top right labeled dialing, where it's a transition from a state to itself. That is, when you're dialing or pressing the buttons, okay, you're doing this several times and, you remain in the dialing state until you've finished dialing. Now we could have had a machine here that had numerous states as part of the dialing process, in which we've dialed the first digit, the second digit, and so on. And in that case, it would be different states. Until we eventually got entered our local number or our our long distance number. That would have complicated the diagram, and remember these diagrams, are abstractions. We abstract over the set of states, and abstract over the events. That's, that's your choice as a designer or a modeler. Notice also that, the diagram is somewhat busy and that there are arcs that have a seemingly redundant labels. This is another example of a situation where we'd like to improve the diagrams by by simplifying them. And that's where we're going when we get to concurrent systems model with statecharts.

23 – Problems with State Transition Diagrams

So with these, state transition diagrams there are several problems that we've noticed. There are too many arrows, there are too many states, and there's no concept of nesting in them. As far as the arrows are concerned, if you've got end states and you've got impossible events. You've got n times m, a multiplicative number of possible arrows. As far as states are concerned we've already indicated that the number of states goes up with the power of the number of possible things that can be going on. As far as nesting is concerned with the example of the dialing the telephone. In, in essence we would like to have done the modeling by having the, the entering of the particular digits somehow hidden within that dialing state.

24 – State Charts

Well, fortunately, there is, at least a partial solution to the problem of dealing with complexity in these, in these systems. I say partial because no matter how nature notation is, you'll always going to be confronted with systems that are more complex than it can deal with. However, state charts, as developed by David Harel. Okay. Our way of coping with this in, in a fa, in a fashion that allows you to do the modeling of the system in a way that help you get an understanding of it. He calls these his improvements to state transition diagrams, state charts. And they provide, several mechanisms for, dealing with the, with the complexity. 'Kay, and we'll be looking at those. State charts are a part of UML. Okay. Tools support them, and, we will be, looking into them in a subsequent. Lesson and giving you a chance to use them yourself.

25 – State Chart Icons

As far as icon are concerned, state chart compromise between the rectangles that we saw in the ovals to a thing called round-tangles, rounded cornered rectangles. They can, they can have labels indicating state name. They can have arcs connecting them which is a, a directed arc that is there's going to be an arrowhead at one end. And the arch's themselves can be labeled with the event causing the transition, possibly with a slash and the action taken when the the event occurs. Also, there's a way to indicate what the default or initial state is. In the case of state charts this is having a small circle that's filled in. You know blackened. And the final state in this case, the final state is one in which there is a concentric outer ring around a filled in a filled in circle.

26 – State Chart Extensions to FSMs

Statecharts add quite a few features. We're going to be looking at the first two because they're the ones that give the greatest benefit. But I'll mention also some of the others, okay? In particular number one thing is statecharts offer nesting or depth, okay? That is, a particular state in a statechart can be its own state machine. And you can zoom in that way. Secondly, they offer concurrency. Imagine that you have two things going on, each of which can have a number of states. Recall from state transition diagrams that in order to model that we have to multiply the number of states. Statecharts allow you to treat those separately, okay? Therefore, only getting an additive number of states rather than a multiplicative number of states. Of course once you've done that, that is separated the concurrent machines into two parts, you still have to synchronize them. That is, they still have to cooperate somehow, and statecharts offers a way of doing that called broadcast events. We'll look at that. And of course you could also use data conditions which are globally available to both machines. We'll look at entry exit actions, we'll look at event parameters, we'll look at history, and of course the default states that we've already seen the icons for.

27 – State Chart Nesting

I have asked you to have a look at, at Harel's original paper. In there, he has several abstract diagrams laying out the various features of the state chart notation. On the left, there is a a state machine that has three states. Notice the default state is the top one and, with the line coming into it, and there are transitions among the states. Totally, there are six transitions there. The version on the right labeled b, okay, is a nested state transition diagram. A new state called D in the roundtangle, surrounds states A and B. A is still the default state for the the state machine as a whole. But notice that some of the lines come out of D rather than coming out of either A or C. Going back to the one on the left, notice there are two transitions labeled f, one coming out from A and going to B, one coming out of C and going to B. On the right, they're coming out of D. That is, there's there's a an abstraction, saying when you leave any state in D under transition f, you go to B. So in that case, we reduce two transitions labeled f to one. There's still the transition from A to C. There's still the transition from B to A. Notice also though that there's a transition from B to D labeled h. In this case, where does it go? Well, notice that we've added a new default state and its transition for C. That is, if a transition comes into D, to the border of D, where is it going to go? It's going to go to the default state, which is C. Well, this is identical to what's happening on the left where there's a direct line from B to C. In this case the line's a little shorter, saving just a little bit of complexity. We could also have the nesting go further. That is A or C or B could themselves have substate machines in them

28 – State Chart Nesting UML Example

Here is an example from UML. It concerns a a machine, that's either heating or cooling. It's some kind of air-conditioning system. And on the lower right is a nested state called heating. Heating is just a two state sub-system. One of the states is activating, and the others is called act, active. Okay, it's got a default state which is the activating state, and it's got a transition which occurs when the activating is ready. And it the action that takes place is to, to turn things on. This nested state is part of the larger system, which at the top level has three important states. It has an idol state, a cooling state, and this nested heating state. The default state for the larger machine is idle, and the final state is a shutdown state. And there are transitions between the various outer states. But notice also the transitions that from the nested heating state, go only to the boundary of that state and not into the internals of that state. Thereby saving duplicate copies of the lines coming from each of the internal states in the heating state.

29 – UML Example Harels Notation

Here's an example of a UML, state chart that illustrates several other features of, Harel's notation. Once again, there are three outermost states, idle maintenance, and active, and the active state itself is, has nested states inside of it. There are transitions from idle to the boundary of active. Two transitions there, and as, as we saw similarly, each of those transitions is going to go to the default state in the internal machine, which is labeled validating. Another thing to note here is that two of the transitions in the nested machine, in the active state, are labeled with text inside of square brackets. These are data conditions. They are tests, logical expressions on the, the the attributes of the object which is being represented by this particular state machine. In the transition between processing and printing on the right hand side of the active state, there are square brackets and inside it says not continue. Okay, continue is an attribute, presumably a boolean attribute, and if it's false, then the transition can take place, otherwise it won't take place. Also between processing and selecting, there's a transition labeled continue which will take place if the continue attribute happens to be true. The other thing to notice about this particular example is that in the bottom, on the left of the active state, there are two lines of text one labeled entry and one labeled exit. These are actions that will take place upon respectively entry and exit from the inner state machine, that is when a transition from idle goes to active. Before anything else happens, the read card action will be executed. Likewise, upon exit from the printing state, when all the other work is done, the eject card action will take place.

30 – Concurrency

Well, that was nesting. The other, important addition to state machines that Harel offers is concurrency. In this case, concurrency is indicated by a dashed line. It separates a larger roundtangle into two other machines, but in this case, it is nesting that goes on at the same time. That is, we've cranked up two machines that are running. Each can have their current state, each can respond to transitions and each can perform actions depending upon the transitions. 'Kay, once again, this reduces the total number of states from a multiplicative combination to an additive combination. Here's from Harel's paper on the left, is the bowl of spaghetti, that indicates the multiplicative combination. Notice the labels of the states are really indicating where you would be in one of the concurrent machines, and where you would be in the other concurrent machines hard to understand what is really going on there. Harel has replaced this jumble by one major state labeled y and left two of the original states, H and I to interact

with it. The y-state has that dash line indicating concurrent actions that are taking place, they correspond to states a and d. They each have default states, they each have their own transitions. But, hopefully, you can see it's a little bit easier to follow what's going on there. Notice also that Harel allows the splitting of a transition to go to two possible states, one in each of the concurrent arms. So the lower right state in the rightmost diagram, I has an E transition coming out of it that's then split into two places. There's also an example of a data condition in here as well. In UML the concurrency looks like the following. There are two states here, there's an idol state and a maintenance state. The maintenance state has the dashed line in this case it's a horizontal line. And two concurrently executing machines, one called testing and one called commanding, each of which are quite quite simple. Each has an initial state and and a final state and some transitions between them.

31 – Synchronization

Of course if you've got concurrent, concurrently executing machines, they have nothing whatsoever to do with them. Why did you put them in the diagram in the first place? They're there because, somehow they're cooperating. And, that cooperation needs to be coordinated or synchronized. State charts provide several ways of doing that. One are called broadcast events, and the other is the data conditions that we've already seen.

32 – Broadcast Cascade Events

'Kay. Here's, here's an example of, of cascade events or broadcast events from Harel's paper. In this case, there are three concurrently executed machines labeled A, H, and D. And there are transitions between them. On some of the transitions there can be an action taking place and that follows the slash. The actions that we've seen so far you can think of as being equivalent to method calls in our object-oriented class model diagrams. But they could also be the issuing of another event. For example in in, in state A, there's a transition between substate C and substate B that the transition itself is labeled with F. But then there's a slash, which says if I'm taking this transition, also issue a new event called G. That is, we've cascaded the event. Now, the events themselves are globally known. So the issuing of the event g here is known to the other machines in the concurrently executing state chart. And this process of cascading the events can go on

33 – Data Conditions

The second way that the differently executing concurrent machines can communicate is by data conditions. We've already, we've already seen this. Remember that they occur within square brackets. Okay? And they contain within them Boolean expressions in which the terms correspond to attributes of a various classes in the overall system model. You can think of these data conditions as being continuously monitored. And that when one of them becomes true, that's like an event were issued saying, look at me I'm now true, I can take this particular this particular transition. State charts, in addition, support the keywords in and not in. What in means is, I'll make this transition if in one of the other concurrent machines I'm in state whatever x is. So if I say in x and in the other machine I'm in state x, then I can make the transition. And similarly from not in. The variables which occur in these expressions, as I said, come from attributes in the system model. And these attributes are globally known by all of the concurrently executing machines.

34 – Special Transitions

UML supports a couple of special transitions that you can take advantage of, each indicated by a keyword. So, here's a two state machine. The transition from the active state to idle state. the, transition is labeled after 2 seconds. Okay? So you can assume that there's a timer here that if you're in the active state 2 seconds later, you'll make the transition to the idle state. Similarly, the idle state has a self transition, okay? That's labeled when we key, the keyword when and then a particular clock time that the system waits until that particular clock time happens before making the transition. We can put this example in a slightly larger context of to ill, to illustrate one other feature of, state charts. So, we have the self transition on idle and we have some, normal, transitions labeled by events in the rest of the diagram. But the transition between idle and tracking, okay? Involves an action, okay? That action is invocating a method, invoking a method, and the method has a parameter p, that is you can pass information in the action calls. Similarly, on the transition itself, the event that led to the transition has an argument p. So, what we're doing here is we're passing on the information. That came in on the event to some kind of method call so it can presumably be processed in the tracking state

35 – History States

The final major feature we want to look at with state charts are called history states. Here's a nested state machine. Where the two external states are the command state and the backing up state. And there are transitions from the backing up state into the command state, two transitions there. And there's a transition from the command state, to the nested state into a circle labeled with a label H. This is an example of a history state. And what it says is, let's remember whichever state we were in, in the backing up machine, the last time we were there. And we left. And, when I'm entered into the history state, I'll go to the state that I was last in. Whether it was collecting, copying, or cleaning up. I could even go so far as to label the circular state H star. And that says, if any of the the states and the backing up state were themselves nested, I could go to the sub-states that were there. This is quite a, quite powerful feature, but it can get you into diagrams that are kind of hard to read because you may have to remember what state you were in, and also look into the, the various nesting levels.

36 – Complete UML State Description

Okay let's, let's summarize this. What are all the things that UML can provide for you in state descriptions? At the top is the word tracking, that's the state label. We can have the entry action, we can have the exit action. UML supports the ideas of internal transitions. These are, you can think of these as self transitions without the entry and exit actions. Okay, UML now also supports the idea of activities. So we have actions and activities. The difference here is that you can think of actions from the point of view of the system as being things which are instantaneous. Typically that means that we're turning something on or we're turning something off. And the time that it takes to do that doesn't play in the rest of the, rest of the system. Activities, however, are things which take time. Okay, the key word there is do, and we're calling some kind of method which is going to take some time. So we're going to be in this state while, while we're following the target. And deferred events are a a, a special situation in, in UML in which the set of events that the system is responding to are, are queued, that is, put into a queue, and only processed at a later time.

37 – Complete UML Transition Description

As far as transitions are concerned, there's a transition goes from a source state, goes to a target state. The transition can be labeled with some kind of triggering event. There can be a guard on it. There can be an action. Okay, that takes place when the transition occurs. And as we've noticed there can be forks in the transition arcs, and there can even be joins.

38 – Relationship to Class Diagram

Okay, you should be getting the idea now that the state charts are quite powerful. There's lot's to them, like the class, diagrams, you don't have to necessarily use all those features in every diagram that you do. How, in fact, do these state charts relate to the class model diagrams? Well the way to think about it is, that each of the classes. In the class model diagram, has attributes, and those attributes form a state space. But each of the classes in the state diagram could have it's own, state chart. And that doesn't mean you're going to be building state charts for each of the classes because, most of the classes have relatively simple states, and are, perfectly well described by the methods manipulating the attributes of that class. In th state charts, there's references to attributes. Those are the attributes of the class. There's also references to actions and activities, and those are the methods of the class. The events in the state chart diagrams are going to correspond to signals. A signal is a dependency in the class model diagram. You can use a stereotype, in this case the stereotype is send, indicate that the move, movement agent class, is going to send a signal, called the collision signal. There's a dependency between those two two classes. So what this is saying is that, as you're doing your state chart modeling, you have to make sure, that it's consistent with respect to the class model diagram. That is, the events have to be named, the attribute names have to be, correct, the method names have to be correct, and so on.

39 – Harels Digital Watch

When I first came to Georgia Tech, David Harel came and gave a talk. And, as in the paper, in the talk, he talked about his digital watch. And he presented a model of the digital watch. He, he in fact had used that as a way of stressing his diagramming technique to see if it was capable of representing the features of the, of the digital watch. From the paper, here's, here's the example of the digital watch, from a high level view. Notice that there are really two states here. One is the dead state and one is the alive state, separated by whether the battery is placed inside it or so on. And then are five concurrently executing sub machines inside the alive state, each of which has their own activities going, going on, and there's even further nesting down in the lower right-hand concurrent machine. The fact that state chargts support nesting means you can pull out any of these sub-machines and consider them independently. You can even provide details in the pulled out version that weren't visible in the top level version. Here's an example of the stop watch state. It makes use of a history state. You'll notice there it had it's own default state for the, for the stopwatch itself in the zero step. There's also a guarded transition using the in keyword that I mentioned before. Here's a pullout on the displays state. Notice that the state itself has a self transition over on the left that indicates that there's a two minute timer. What this is saying is if you were in one of the display states, other than the default state, after two minutes it will, it will flip back unless you were in the stopwatch state, in which case it'll keep, keep you looking at the stopwatch.

40 – Harels Digital Watch Quiz

So using this diagram, here's some quiz questions that ask you to figure out what's really going on there. And for this quiz, you should be aware that the events labeled A, B, C, and D correspond to the four buttons on your stopwatch. First question is how many outermost states does the watch have? Second, what button must be pressed to turn on the alarm clock feature? It's the alarm clock, not the stopwatch. Third, in the ALIVE state, how many concurrent machines are running? Fourth, when the C button is pressed to set the time, which part of the time, that is the day, date, hour, minute, or second, is the first one the user can change? And fifth, if you are changing the time on your watch and you press the B button to indicate you are done, what unexpected side effect occurs?

41 – Harels Digital Watch Quiz Solution

Well for question one, how many outer most states? There's two we've already said this dead and alive states. Which button must be pressed to turn on the alarm clock feature? That's button a. In the alive state how many current machines are running? Well there's five. There's the main one, the power state, the light state and the two that are labeled alarm-st and chime-st. When the C button is pressed to set the time, which part is first, that's, the user changes the second setting first. And then the question about what unexpected side effect. When Harel was at Georgia Tech, he told us about how when doing the modeling of his digital watch. He found a bug in the way that it ran. Whenever you hit that B-button, it turned on the light whether you needed the light turned or not. Going back to the garage door opener for a minute, when I modeled the garage door opener of my house, I found that there was a bug in its system as well. The bug arose when I was going on vacation and I was hitting the security feature. Now this wasn't in the example that I showed you, but. The real state machine for the garage door opener has a security feature and that turns off the remote the remote control so you can't open the garage on the outside and what I was going to do was start the door go, door going down, hit the security feature and duck out underneath. Okay the system had not been designed to change that security feature as the door was going down and I lock myself out. Okay, and it's another indication that getting these reactive systems correct is quite difficult and careful modeling of things is important. To make sure that you don't get into these embarrassing or possibly situations that lead to safety problems.

42 – Summary

As I said, these reactive systems are hard to build. You probably have heard of examples of complex systems getting into deadlock states or otherwise freezing up because of the common [UNKNOWN] blowup in complexity that occurs from all of the internal things that are going on. So, you have to spend some time in getting these things right. And some kind of behavioral modeling technique, like statecharts, okay, can be very helpful to getting that kind of assurance. Besides statecharts, UML provides other diagrams that can be used for understanding behavior. Okay, we've already seen these in our review of UML, activity diagrams, sequence diagrams, collaborations, use cases, communication, timing and interaction overview diagrams. Okay? Outside of UML, there's a couple of other behavioral modeling approaches which I'll just mention to you, we won't go into it. One of those is temporal logic. If you've heard of model checking and model checking tools these are ways of modelling system and asking can I ever get into this certain state that I don't want to get into, okay. Am I ensured that I can't get into it, okay and model checking tools can help you get answers to that problem. Another notation for expressing concurrency is process algebras. These allow you to specify what things can go on

concurrently and the reads and write behavior between the concurrently executing activities. We will be looking further into statecharts, we will do, be doing an exercise that ask you to learn the features. This is such an essential part of getting models right that want to make sure that you have kind of acquired that skill.

P2L10 Clock Radio Exercise

01 – Modeling with Statecharts

Statecharts are a precise way of modeling the behavior of complex reactive systems. Such systems are ubiquitous and errors can lead to safety, security, and usability problems. In this lesson, we will go through an exercise in modeling the behavior of a common clock radio.

02 – Description

The radio is powered by electricity from a wall socket at not a battery. It can be controlled by two manual knobs. One for volume on the top right and one for tuning on the right side. The chosen frequency is displayed on the right side of the front panel with a small vertical white bar. The display features a twelve hour clock on the left front and two small lights. The light in the upper left hand corner of the display labeled AM indicates whether the displayed time is in the morning, when the light is off or in the afternoon or evening when the light is on. The light in the lower right hand corner of the display is labeled wake. If it is lit indicates you have armed the alarm. In addition to the above, the radio has two switches and five buttons. One of the switches on the top edge of the back determines whether the frequency band is set to AM or FM. The second switch found in the center front of the top has four positions. The switch slides horizontally from left to right. In order, the positions indicate whether the radio is on, off, armed for wake by radio or armed for wake via a beeping sound. Four buttons are found on the left side of the top of the radio. They can be used to set various timers in the clock. They are labeled hour, min, wake, and sleep. By pressing the wake button, you can set the time you wish the alarm to go off. By pressing the sleep button, you can set how long you would like the radio to play before automatically shutting off. Using this feature allows you to fall asleep with the radio playing. If neither button is pressed you can set the current time of day. You've actually set these three timers using the other two button on the front left of the top of the radio, they are labeled hour and min. By pressing hour you increment the hour of, of the respective timer, the wake, sleep or time of day timer. Similarly for min, if you wish to be awakened 15 minutes later tomorrow morning than you were this morning you would hold down the wake button while pressing the min button 15 times. The final button is the snooze button. It is the large button found in the center front of the top of the radio. The snooze button is useful when the alarm is trying to wake you up. Each time you hit the snooze button the alarm shuts off for 10 minutes. The final feature to note about the radio is that the alarm will automatically turn itself off after one hour, to prevent the situation where you forgot to turn it off and it was audible all day long.

03 – Exercise Introduction

So this lesson we're going to go through an exercise. Imagine you had to prepare an analysis model of the external behavior of this clock radio using state charts. What steps would you take to do this? Because we are concerned here with modeling behavior, one natural way to

get started is with usage scenarios, also called use cases. A use case is a sequence of steps, each involving an actor performing an action, possibly on an object. Clearly any model of behavior of the clock radio must indicate how each of the use cases is realized. See if you can come up with three typical use cases for the clock radio. Okay. So the first use case I can think of is that the user is setting the time for one of the three different modes. Whether that's the display time or the time we want the alarm to go off. There's a third time but with each of those, I guess there's a separate use case but we'll just say setting the time is use case one. Okay, so, Jared is right that each one of those would be a separate use case and now we're picking the one about just setting the time of day that's visible on the display. And then for use case number two we could change the frequency of the radio so we may be in AM mode and changing the frequency, or in FM mode and changing the frequency. So regardless of the mode, we're changing what station is coming in. Right, so we're going to be twisting that knob to change it. On the side. On the side. Okay. And then for a third use case, let's say the user presses the snooze button, the alarm's going off and they want to sleep a little bit longer. Okay. Clearly there are lots of other ones, like just listening to the radio, turning it on and just listening to it, or getting shocked out of bed by the beeper going off. Lots of possibilities here and ultimately, our state and chart model needs to be able to cover all the cases that might arise, including error cases, including situations where what you had hoped to happen doesn't happen and the radio has to respond in that situation.

04 – Percepts Quiz

Once we have the use cases, we can use them to determine the radios percepts. Recall that a percept is an externally, sensible, by that I mean visual, audible, tactile aspect of a device. That is you can walk through a use case to determine what percepts the user interacts with. Typically a device's output and some of its input comprise its percepts.

05 – Percepts Quiz Solution

Okay, so, I think I'll start then with our first use case that we talked about which is setting the display. We have in that use case, I guess involved, we have the either the wake button or the sleep button. So those are, I guess they're not percepts but they are button that we can interact with. But our percept that we see is the clock time. So we both have the hour and the minute. You're talking about setting the time of day? Yes. Okay. Yes. So, I guess we have the display time as the first percept, and then we also have the percept for, and when we change the frequency with the tuning dial. So this is now the second use case? Yes. Well, let's stick with the first one. Actually you're getting some feedback from the button when you press it, so that is a percept tactilely, but it's not retaining any state. Okay. So we probably don't need to model that as a thing. But we'll model it as an event that takes place, you pressing the button, releasing the button. And then when we're pressing those buttons, want to make sure that I got the inscription right. Would the light come one for wake? Like when you press wake down is there like a little light that indicates that we're setting the wake time or are you just by sheer fact that you're pressing wake, we now know that we're setting the wake time? So good question. I guess theres some ambiguity with the term, okay. When you set the wake time that doesn't turn on the alarm mode. It's setting the time which, if you set the alarm mode, the radio would come on. Right, okay. So the light is not going to come on when you set the time, but only when you slide that switch over either to radio or to alarm. So does that, I think that takes care of use case one then. Okay. And then for use case 2 in which we're tuning, using the tuning dial, we have the percept of somewhere on the radio seeing where we are in the frequency band. Okay, so there's that vertical white bar that moves across. And then we, I

guess we're actually, you know, we're adjusting that dial, so we're not seeing anything. I guess you could see the dial moving, but that's something we're interacting with. You're definitely interacting with it, and in fact, tactilely again, you're getting a feel for it. And this one does retain some state, okay? Although, in the middle of the night you're not going to be able to tell what state it's in. Oh, right, okay. Which is different than the volume knob, where actually there's a little vertical piece of plastic on top of it that indicates its position. Okay, is there like a limit I guess mechanically maybe with that switch too, that we could probably, or with the dial? If you turn it far enough you get the aliens in outer space, but no, it's limited to normal AM and FM radio bands. Right, okay, so we have the vertical bar and then the dial or two percepts that we're interacting with for this use case. And then for our third use case, which was pressing the snooze button, we have the snooze button. Right. Okay, yeah, I guess for that particular use case, I feel like I might be tying in other use cases that are involved with what would set you into actually pressing the snooze button, which would be either be being an alarm mode or in music mode, but those themselves I think, could we put a separate use cases. Should we worry about them as separate use cases? So recall when we talked about use case modeling that we can have contingent use cases that is we can, the UML modeling notation allows use to say that one use case is included in part of another use case. And that allows you to factor out the common part and have separate other parts. However, there is one another percept concern with this particular use case. When you set the, when you hit the snooze button, what's your purpose for setting that button? To turn off the alarm or the music that's going. So turn off the speaker, essentially, and the speaker is certainly a percept. In fact, that's the whole purpose of the clock radio is to have that, to control, that particular percept. So we definitely are going to have to model the fact that when the snooze button is hit, something happens with the speaker. So to summarize the various percepts that are involved here. The speaker, the time display. We just went through those along with those indicator lights on the time display, saying whether the alarm is turned on and the AM/FM percept. There's also, as we indicated, the rotational position of the tuning knob, the horizontal position, that left and right position of the vertical frequency bar. Rotational position of the volume dial, the current setting of the switch which is the AM/FM band, and the current setting of the slide switch, which is on/off, radio, or alarm. Think of it as a mode switch there. The other devices, the various buttons, are in fact that. They don't retain any state. But you can still use them to cause other effects in the clock radio.

06 – Percept States Quiz

So to drill down one step deeper, the next step in analysis is to determine, what stage, each percept can be in. Each percept will in general, be modeled with a finite state machine, and these finite state machines can be though of, as executing concurrently. That is a state chart in this particular situation is going, the models of clock radio is going to be carved up into different, different, concurrently executing state machines, in which, each state machine is, going to, or most of the state machines are going to correspond to the states of the particular perceps. For example, the mode switch can be in four states. And when the user sets the switch to one of these states, this naturally affects the device's behavior. Let's see how this works for one of the clock radio's percepts, its time display. How many different states can the time display be in? For the moment ignore the two accompanying lights, the AM/PM light and the alarm indicator. Just the rest of the display, how many different states can it be in.

07 – Percept States Quiz Solution

Third grade math. 'Kay? Exactly right. However it's unlikely we're going to have it, we're going to want to model a state machine that has 720 states in it. With state charts, we can reduce this to two concurrent finite statement machines totaling 72 states. One for the hour and one for the the minutes. But even that seems little bit over, overkill here. Resi regardless of which version we choose, the resulting state chart would still be crowded and not all that useful. Instead, we will abstract the machine into a single node, labeled clock time. And assume, that the underlying logic for computing the correct signals to cause the lights to display. There are 720 different possibilities work as expected. Deciding just how to abstract the many possibilities for percepts is a key skill in state chart model. We can do the same abstraction on the WakeTime and SleepTime displays, giving us a simple three state machine that looks like what we're seeing here. Note that we have not yet labeled the transitions between the states with events, that step will come later.

08 – Display FSM

What we're looking at, is a finite state machine, a state chart that is carved up into a collection of concurrently executing sub machines. And the one that we filled in, that pivotal state chart is labeled Display. And the Display, if you think about it, can be in one of three states. Either, it can be displaying the current clock time, time of day. The time when you wish to be awoken. And, the amount of time that you'd like the radio to run, as you're going to sleep at night. And between these each status is an oval. And between the states are some transition arcs. And in this case to make it a little bit simpler, we've abbreviated the two arcs, the ones that go back and forth with a single arc with double-headed arrow. But in general those are two transitions and we have to take care that we represent them both

09 – Mode Switch Quiz

Let's turn that to a physical, rather than an electronic percept, the mode switch. Remember, that was on the top, front of the, radio. The sliding mode switch controls, whether the device is off. On playing the radio continuously, on laying the radio only when the wake up time is reached or on beeping, only when the wake up time is reached. Can you model this percept with a finite state machine?

10 – Mode Switch Quiz Solution

So, tell us what you've got. So I just drew a box so I can say we're talking about the mode state machine. And within the mode there are four possible states we can be in. We can be in on, off, music, or alarm. And it seems that the transitions between those states kind of follow this order because the switch itself can't jump all the way from, for instance, alarm, back to on. It's not circular. It just goes kind of back and forth. So, we go on to off, off to music, music to alarm, and vice versa, and go backwards. Straight forward. I hope all of you were able to get that particular solution. Notice that this machine. That is, the machine that describes the mode switch, can execute concurrently with the clock time display. So, that these are concurrently executing machines. You can see the time on the clock at the same time you're moving that switch. Notice, also, that this kind of sequential state machine, going back and forth between these four states, is essentially the same thing that we would use to model the bandswitch between AM and FM. That's in two states and we can go back and forth. So, we'll just go ahead and plunk that one into our composite machine now.

11 – Station Indicator

Another percept of the clock radio is the tiny bar on the front that indicates the current station. This device is physically controlled by the StationKnob, and like the knob, can take an infinite number of positions. It can be modeled as a single state with no transitions. Now, we would expect that the StationKnob and the station indicator are coordinated with each other. After all, when you turn the knob you would expect the indicator to move. And later we'll figure out how, using our state chart modeling notation, we can connect those two, those two concurrent activities together. But for now we'll just treat them as separate machines.

12 – Station Indicator FSM

In our growing diagram, the indicator that is the frequency indicator, is a single state which I have labeled here frequency. And there is no transition, that is the user can't directly move that vertical bar around. It will be the job of the station knob when the user does undergo a particular transition. By turning the knob to somehow affect the station machine, concurrently executing machine. As I said, we will delay dealing with that until a little later.

13 – Speaker

Okay. Clearly it can be on or off and, a part of the abstraction process is deciding that for the purposes of, our, our understanding of the radio here. That, we're not going to be concerned with as if there were any states between the two. Now if we were, went down to the electronic level, there would be some voltage levels and there would be, you know, transitions going on there. But that's that's not what we're concerned with here.

14 – So Far

So far, we have been developing a StateChart to describe the behavior of, of a clock radio. Thus far, we have used seven concurrently executing machines to model the radio's percepts. We have left some placeholders for other machines we will need to complete the diagram. We have also left out the transitions for the time being. Let's begin to look at the events that can provide the impetus for these transitions. Recall that an event is a spontaneous or instantaneous occurrence. That is, we're not concerned with its duration. It can communicate information such as if we turned the dial what position are we turning the dial to. But that the state machines can be sensitive to those events taking place and cause a change of state when they detect them.

15 – External Controls and Stimuli

Ultimately, behavior is about how a device responds to these events. To model this, we need to understand what those events are. For the next step in the state chart modeling process, we will look at how to determine, for each of the external controls, what stimuli or events they can provide to the device. Using the use cases we have developed earlier, why don't you begin to make a list of the user actions that might generate events that we would expect the radio to respond to. And as a hint, go back and look at the picture, and use it to find each of the ways that user can affect its behavior. Okay. So, the first is that the user can switch the mode button, and that's going to change our state machine based on, we have to look at what mode we're coming out of to determine what mode we're going into? So what is the event? What is it that the user does? It's just clicking that little knob over. So, sliding the thing? And, there's really two events here. There's one is pushing it to the right, and one is pushing it to the left. Okay. Okay? And, as a programmer, if you were eventually

doing an implementation for this thing, you would have to deal with both of those. And you want to deal with them separately, right? You have to know. Now we could model those as two separate events, or we could model them as one event, maybe called slide, and have a parameter to it that says left or right. What else? Okay, so we can also turn our tuning dial. Okay. And here again, we might want to have a parameter which indicates the rotational angle or something that indicates the information that is being communicated from the dial to the system, that's actually going to change the vertical bar and the actual tuning behavior of the radio. Right. Okay. Another one is flipping our switch for the AM or FM. And, once again, that's a left-right thing. Okay? I guess, is it not toggle, though? It seems like since we just have two options, kind of like toggling between, do we need a- Well, this particular clock radio that I modeled, that's on the picture there, is a slide switch left and right. Okay. And then I have pressing the snooze button, but I felt like it might need to be split, because you could press the snooze button while it's beeping, like kind of the intended use case for that. But you could also press the snooze button when It's just sitting there on accident or something, or you just fumble- Okay, so let's think about this a second. As far as the user is concerned, the event is pressing the button. Okay. Okay. What's in the user's head is irrelevant here, okay? At least as far as our initial approximation of things. Certainly, we want to consider the possibility of what it means if the user presses the snooze button when the alarm is not going off. And presumably, our resultant state chart will describe for us what behavior will happen. This is one of the real reasons we want to do this sort of thing, is to come up with insights about, well, I haven't thought about that yet. And I need to take care of it. We certainly don't want the radio to go off when it's not playing, just because we hit the snooze button. What else you have? So, this is kind of like three parts to this, but when you press the wake button and then press either the hour or minute button, and then that's going to affect. So is that one event or two? I think the first event, there's one event of pressing the wake button, because that's going to affect our different systems, like displays is now going to have to display our wake time. So that's a separate event, and then there's a conjoined event when you press the wake button, and then you press either the hour or minute buttons. Now you're going to be adjusting the display time, so that's switching modes from your display to wake, for instance. And then you have this separate case for actually changing the time. Okay. So, we want to be clear about this, because this one of the different ways that state chart modeling forces us to think about things. As far as the user events are concerned, they're separate. Okay, so, there's an event of pressing that wake button and releasing that wake button. So another event here is release. Pressing the wake button. And then, there's a separate thing going on, pressing the hour button. The order in which those happen is very important. If you press the hour button first, it's going to change the time of day. Right. Okay, if you press the wake button first, then press the hour button, it's going to change when you wake up. So its important that the state chart that we end up with reflects that difference, because the user intends them to be used differently. The event is pressing the wake button, the event is releasing the wake button. Another event is pressing the hour button and pressing the minute button. And our machine, as we eventually refine it to deal with all these contingencies, had better behave in an expected way as far as all of the precepts in all of those possibilities, all of those possible situations.

16 – From Actions to Events

So, imagine that you were a programmer implementing the internal logic of the clock radio. You would have to take into account all of the different events, okay? And all of the possible combinations of those events. So, ultimately, you have to deal with all those cases. And consequently, it makes sense during the analysis phase, to list them all, okay? To give a fairly

precise description of what are all the different events. What information comes in along you know, its parameters to those events. And ultimately then, what the system, how the system's going to respond to those events. And for the purposes of this exercise, I've expressed that in a table in which we have numbered events, and we have the the description of the event, and then the systems response to that event. So, listed here at the beginning of that table

17 – Outermost Layer StateChart Quiz

State charts can be nested. That is a state may have a state chart nested within it. Let's, for a second, step out from our layer in which we've been modelling and think about the outermost layer of this, of this particular device and work inward. If you recall the paper I've asked you to read by Harrell the digital watch had the same outermost layer having to do with whether the batteries are in the watch or not. What might be the analogous situation here, as far as an outermost state is concerned?

18 – Outermost Layer StateChart Quiz Solution

So, we can either have the radio be plugged in or not plugged in. Sure. So, and that's important, because the behavior is going to be much different depending upon whether you're plugged in and plugged out. So, what would be the user action to switch between those states? So physically plugging in or unplugging. Okay. So we have a two state machine and we have two events that cause transitions between those states.

19 – Adding Events

So, here's a, a very simple outermost state chart in which we have an unplugged state and a plugged-in state. The pulling the plug causes us to go from the plugged-in state to the unplugged state. And I've labeled here, this is not part of the state chart notation, but I've labeled here, what the event number is in our table. So vent number five is the pulling of the plug. Symmetrically, the, going from the unplugged to the plugged-in state is plugging in the plug, and that's, that's event number six, okay? But for the remainder of the, this exercise, we're only going to, consider sub states of the plugged-in state. That is, now, imagine that, the rightmost state here has all of those other concurrently executing machines which we were, talking about previously.

20 – Event Allocation

So, some of these events that we've been talking about and put in our list affect the different concurrently executing sub-machines that we've already modelled. See if you can figure out which ones of these, which ones these are and label the transitions in the finite state machine with their numbers. So, pick a particular event and decide which of the state machines are affected by that particular event happening. Starting with number one for turning the dial for the frequency, that would be affecting our station finance statement. So, we would now expect, as a, in that particular machine, the station of machine that the arc that we have, the self arc that we have is going to be labelled by a particular event, which is event number one, okay? Simple, right? And we can do the same thing for some of these other events and come up with an annotated diagram or a diagram in which the transitions all have arcs on them. In our particular, this particular exercise, we don't have any spontaneous transitions. The spontaneous transition is where you're in some state and you go to the next state without any event happening. Okay, so we ultimately are going to see, we're going to need to have for each of the arcs that we have in the diagram, some have been causing the transition. Remember that our arrows with two heads on them are really an abbreviation for two arrows, one going

in each direction. So I think we meant event four for the, event one I think's changing the volume so that would be the volume dial state machine but for this one were talking about four. Okay so with the transition were putting on the station knob, self transition is for event four. Here are the additions to our machine to handle events one, two, three, four, 12, and 13. So with respect to event one, that's the volume knob that's similar to our station one, or switching the band between AM and FM. Those are events two and three, one going to the left and one going to the right. And similarly sliding the mode switch to the right and left are events 12 and 13.

21 – New Sub–Machines

Some of the events that we've listed suggest that we need to add additional state machines, sub-machines, to our set of concurrently executing machines. Typically, these correspond to internal timers. So you think about it, the clock radio has some timers in it, for example, with the snooze button. You know it has to time ten minutes before it turns itself off. And those timers, even though they aren't directly or explicitly visible to the user, do effect things which are visible to the user. So, we're going to have to model them. So, the question then arises as to which ones those are, how do you come up with that list of things? And one way of doing that might be to consider events eight through eleven which are the events that correspond to depressing and releasing the wake and sleep buttons. So think for a minute about what is going on when those buttons are pressed, and presumably, then, the user is also pressing the hour and minute buttons. And you want me to design another state machine to represent this? Well, first think what the radio has to do to deal with those things, and see if that if that doesn't then suggest that we need to keep track of some history somehow, and if that then suggests that we need to have another state machine there. Okay. So, I think, based on the buttons, say, none of the buttons are pressed, talking about the wake button, and the sleep button. Right. It seems like that needs to be represented by some kind of state, say, nothing, or none, or something. And based on that state, your other machines are going to act a certain way. Like when you press the hour and minute button, the rest of your state machines for adjusting the display and so on and so forth, will act a certain way. But then you have a state where the wake button is pressed, so that's it's own state. And based on it being in that state, your other machines will act a certain way. So, while we're pressing that wake button we are in a state. Okay? Yes. And that state then allows us to press the hour button. Okay? So, pressing that hour button while we we're in that state has a different effect than pressing it when we're not in that state. Okay? So, let's see what such a state machine might look like.

22 – Setting the Time

Now recall that a minute ago I said that we had no spontaneous transitions and yet we have what looks like a spontaneous transition between the none state and the clock state. We'll see that we are going to need that distinction because the user can actually have an event here like hitting the arrow button which causes us to move from the none state into the clock set state. So we'll come back to that when we get to these other events.

23 – Responses to Events

The ultimate value of the clock radio to its user is how it responds to these listed events. Each of the events we have listed should have some effect on the radio state. Consider what happens when you turn the volume knob. Certainly you would expect the sound coming from the speaker to be louder. But there is another response. The rotational position of the knob will also have changed. This is an important piece of feedback to the user, who may be adjusting

the loudness, in the dark of the night. We can fill this in, this information into, into a table of responses.

24 – Stimulus Response Table Quiz

So we have three columns. One, one is our event number the second column is the event or stimulus that we're talking about, and the third column is, just in English, what a, what response we expect from from the clock radio when that event takes place. Also, we have already talked about Events 5 and 6, so they also can be filled in. See if you can fill in for the table for Events 12 and 13, which is sliding that mode switch left and right

25 – Stimulus Response Table Quiz Solution

So if you were in state On, you'd be one shift to the right would put you into state Off and in the same way Off to Music and then Music to Alarm. And similarly for going left. And we can we can list that by saying, using the word if or some conditional. Ultimately in the code, we are going to have to have a conditional statement that indicates these various possibilities.

26 – Stimulus Response Table

And we can continue this process of filling in the table, with the rest of the responses for each of the, possible events that are listed in the table.

27 – Timer Events Quiz

As with the external events we considered earlier, we need to model any events the internal states respond to. The most interesting such event is when the clock time reaches the alarm time. After all, that's why you have a clock radio, right? You want it to go off in the morning and wake you up. What response should the the, the radio, the clock radio have when those times when, when the clock time reaches the alarm time?

28 – Timer Events Quiz Solution

So, the first is, we're going to play either the beeping noise, or we're going to play the radio depending on, I think that we had that setting from earlier, from the other system of which mode we're in. So we have a state machine indicating whether we're in a mode that we expect to hear beeps or we expect to hear radio. And depending upon which of those two states we're in, we would expect to hear different things coming out of the speaker. Right. Okay. What else? And then we are going to be working with our two other internal state machines. The one for starting the snooze timer that'll run up to ten minutes. But that's only if the user has hit the snooze button. That's right. Okay. So we don't expect, as a response to the times matching- Right, right. That that one happens. Okay, well then, just that one system then, of the internal system for our alarm in which, after an hour, it'll go off. So we have to start counting that hour up in order to know when to turn it off. So to deal with this, we've invented a new event, event twenty, when the clock time reaches the alarm time. And we need to list as the responses for that event what Jared was just talking about. So while we're on the topic of setting the internal timers, we said the snooze one, this won't be effected unless we're talking about the specific event where we press the snooze button. If you were to press the snooze button, will that reset the alarm, the internal alarm timer, back to zero? So, recall that we have this timer which is timing up to an hour, for how long the radio's going to play or the beeper's going to beep. And Jerod's question is if during that time the user hits the snooze button to shut them off, whether that timer resets. I don't personally know, but

that's an excellent question that the developer would actually have an answer to. My intuitive reaction is that no, it doesn't reset the timer, it just allows you to snooze a little while longer. Okay.

29 – Internal States

Things are a little trickier than this, however. It is occasionally necessary, even when modeling external visible behavior of a device, to invent some additional internal states. The most common example are timers, such as what we saw earlier. A user can't really understand how a clock radio works without appreciating that there are several internal timers working. See if you can come up with a list of what internal timers are there. Okay, what timers do you have? So, the first one is the snooze timer. So as soon as you hit the snooze button, then in 10 minutes, the- Times up to 10 minutes, sure. What else? There is an alarm time. I think, or you know, the alarms not going to go on forever. So there's like an internal, isn't there a state for, our alarm has started and then at some point the alarm needs to go off so its not playing forever. So, recall the original requirements in which we said that the alarm can only go for an hour, and so there has to be a timer to time up to that one hour. Okay. What else? For the sleep, internally when you are playing music and you have your sleep timer set. Is there something that kicks off? Usually you press sleep on, right, or something like that? And then it'll run until it meets the time you set for when the- Well if you set the sleep timer. Then when you move your slide switch or mode switch into radio or alarm, I guess into radio, it's going to, normally if you hadn't set the sleep time, it would go off. You wouldn't be hearing any radio. So, if you're sitting there and move it to on, radio comes on. Okay. But if you move it to radio. Music Music, music. Okay, it would normally go off. But if you have the sleep timer set, it goes off after that amount of time. And then, another internal timer is for your sleep timer. I guess I want to make sure I understand correctly how the sleep timer, like how it functionally works. So you press the sleep time, and that's like, say you want to play for 30 minutes before it goes off. So to activate that, what mode do you need to be in to start your sleep timer? Okay, so there's two things going on here. Setting the amount of time before the radio goes off and then actually causing that to happen at night when you're ready to go to sleep. So, we know how to set the time by hitting the sleep button and then setting some time there. Now, it turns out on this particular clock radio, we've seen that the possibilities are 12 hours and there's 60 minutes. But, this particular clock radio, you can only set it for, I believe, up to an hour. Okay. So, you're really just setting the minutes part of it. And, I haven't personally tried this feature out, so I don't what the internal logic does in the situation where you say three hours and twenty minutes, even though it only can do this sleeping for an hour. So, that's one thing that as an actual developer, we'd have to decide upon. Okay. Then there's the question of how, when you're going to bed at night, this thing works, and what it is, is if you have set the sleep time, if it's got an actual value you've put in there, when you move the mode switch into radio you would normally expect not to hear anything. But if you've got the sleep time set then you would actually expect the radio to continue playing until that sleep time expires. Okay. So then, I guess, just the internal timer for that. So as soon as you move it into that mode, it goes up until you're to that threshold. Right. And then lastly, we have this internal clock timer for your actual display clock. So yeah, we're clocking the minutes and hours as they go by, and so there has to be some timer to do that particular thing. So we can add these timers and sub-machines, and when we do the results, this is what you see here.

30 – Other Internal Events

For other internal states, their events and responses look like the following. We have, new event 19, which is the alarm timer expiring, we have the snooze timer expiring. And we have the, clock timer reach the wake time plus one hour. That is, we expect things to be shut off. And for each of those we have what we expect the response of the clock radio to be.

31 – Guarded Transitions

For guarded transitions, earlier we looked at situations where the response to an event is conditioned on a sub-machine being in a state. For example, with event 20 we had the response that looked like the following. If in mode music, go to speaker go in the speaker sub-machine to mode playing. This response can be coded as a transition between the silent state. And the playing state for the speaker that occurs when event 20 happens. And there is a guard that looks like the phrase in, in music. And that particular logical expression is in square, square brackets.

32 – Cascaded Events

The second way of coordinating activities would be cascaded events. In state charts the response to an event can be the broadcasting of another internal event. Because all states listen for all events, this mechanism can be used to communicate between concurrently executing sub-machines

33 – Example

For example, when the frequency knob is turned, which was event four, three responses are required. The physical knob ends up in a new position. The radio channel must be changed. And the white vertical line indicating the current station must be moved. The position of the vertical bar is in the province of a different machine from the one that was recording the moving of the knob. Somehow, it must be, this, this other state must be informed of the new station. This can be accomplished by using, by issuing a new internal event which we'll just call event A to which the station machine responds.

34 – Still To Do

We can do this, the same sort of invention of new events were appropriate to make sure that all of the news cases that we started out with actually cause the machine or make the radio behave in a way we like or behave. Well, we're going to stop the exercise here, but there's some things which we would still have to do to get a complete model. Although this model process, we have undertaken seems quite long there's still some things we'd have to do. We'd have to indicate what the default states for each of the concurrent machines is. Recall that for a state machine we can indicate what the state is when we turn things on, and we would need to do that for these concurrent machines. We would have to place the guards on the transitions where required, and we'd have to invent these in, internal events. The results of this process can be seen in, in the diagram shown here.

35 – Validation

We're not quite done however, so far we've been in the, we've been engaged in building the state chart model. Once the modeling is over, the resulting stays sharp but still be validated. There are various ways we could, we could perform this check. We could hold a review that is get a team of people involved in. I'll walk through the use cases and make sure that each of the

concurrent machines is doing what you would expect it to do. You could do model checking. Model checking is a, is an automated technique where you can encode all of the concurrent state machines and any questions or tests. You'd like to determine whether or not the state machine can ever happen in the state machines and then you can run what's called a model checker to determine whether those things can ever happen. And we could even and this is similar to but he could have a separate, we could build a simulator of the execution using some kind of state chart interpreter, which probably others as, as well, including going back to the users with any questions that arise during this validation process.

36 – Statechart Modeling Method

So this exercise that we've gone through with the clock radio, really is a, an example of a process which you could use to do say chart modeling. And here are the step that we went through. We prepared a use cases to start with of typical uses of the clock radio. We determined the external percepts, that is, what the user can see or hear or feel, with respect to the device. We modeled those percepts with states, which may be corresponding to currently executing state machines. We determined the external controls and the stimuli or actions or user actions that could occur. In doing so, we might model with additional states and, or add in additional transitions or events. Then we began to consider the responses that the system have and we did this with a table in which we listed. The various events and the responses of the systems to those events. We added in some external internal states or state machines to handle the timing situation. We provided coordination mechanisms including the guarded transitions and new internal events. And added any additional actions and activities that the clock radio is required to do in order to implement, or, or implement these particular responses, and then we, we validated the resultant a state chart

37 – Conclusion

Clock radios are a common consumer device which people can use without any training and without normally making any mistakes. Never the less they are complex, as the state chart we have produced indicates. In fact, the clock radio that I have at home has a bug in it. If you get awakened by the radio, turn it off and then change the alarm time to a later time. For example, to waking your spouse, the radio comes back on. It still thinks that within, that it is within the hour window. Okay I call that a bug and I think that the original designers of the radio should have detected this particular situation and change that implementation. In any case it is only by carefully modeling and validation that such situations can be avoided. State charts are a device that can help you do that careful thinking and hopefully lead to better implementations, better understanding of complex situations and ultimately better implementations of them.

P3L01 KWIC Exercise

01 – Software Architecture

To get a feel for software architecture, we will do an exercise first described in a paper by David Parnas, which is linked on the class resources page. The exercise asks you to come up with four different architectures that address the same problem. The problem is to design a program that produces a Key Word in Context index, also called a KWIC index.

02 – Key Word in Context

A KWIC index system accepts as input a sequence of text lines. Each line is a sequences of words and each word is a sequence of characters. You can think of the lines as containing titles, something like titles. A line may be circularly shifted by removing its first word and appending it to the end of the line. Thus, a line consisting of four words will have four circular shifts, including the original. The idea being, that we can index into the list of lines using each of the words that comprise the line. The quick index system outputs a listing of all the circular shifts of all the lines in alphabetical order of the key word used to shift the line. The idea is that if you want to look up any of those titles, you can use any of the words that comprise the title to find it

03 – Example of Circular Shifts

For example, if the original title is Gone With the Wind, a good Atlanta title to, to deal with, then there are then there are circular shifts for it, for them, one for each of the words in the title. And we've indicated them here by underlining the word that we're shifting. So we have Gone with the Wind, and with the Wind Gone, and the Wind done Gone, no, the Wind Gone with. Okay, and the Wind Gone with the. And if we wanted to look up using any of those words we could because now we've indexed on it. And when we alphabetize it, we have the one starting with gone and then the the and the wind and with.

04 – Example with Multiple Titles

Of course, the value of doing the quick index is if we have more than one title, and here's an example with three titles. Beyond Gone with the Wind, we have War and Remembrance and The Winds of War. Let's see how that looks.

05 – KWIC Exercise

Here's a solution to that example with three titles. And note that we have in doing this, we have left out unimportant words like and, and of, which are sometimes called stop words. And that we have rotated the output so that all the key words line up in a column. Okay. And when quick indexes are actually published, you can see them look like this and you can just go down the column to find the appropriate word. And so we have, in this case, six lines

in our output. We've removed the stop words from the, from the index list and, and then alphabetized according to the key word.

06 – Diagramming KWIC Quiz

And so, here's our, here's our exercise. Assume that you had to implement quick. Or, more accurately, to design and implementation of it quick. And to start doing that, you want to break it up into pieces. I realize this is a small problem and maybe doesn't really warrant having pieces, but assume that you want to break it up into component pieces, okay, and we're going to represent the architecture you come up with as, with a diagram called a box and arrow diagram. And the boxes are going to correspond to the pieces or components that you come up with. And shoot for something like three to five or six, different, different pieces, components, okay. And come up with a label for each one of those components. After you've done that, decide how the comp, components are going to communicate. And [COUGH] have line between two boxes saying how it's going to communicate. And in particular there's two typical kinds of communication. One is a flow of control, typically A calling B or A passing control to B and the other is a sort of data, data communication that is that one component uses the data in another in another component. Indicate in your diagram which kind of communication, whether it's a data communication or it's a flow communication, control flow communication, and you can do that by using different line style or whatever, or wanting a textural label. And the trip of the exercise is I want you to come up with at least two solutions. Okay. Okay? And so take, take a minute and see if you can't sketch out a couple of box and arrow diagrams. Do I need to be worried about assigning words to talk about the relationships between the boxes and arrows? So there's only going to be two kinds of relationships. Okay, there's going to be, think of A calls B. Okay? And so you could just have one kind of line a solid line with an arrowhead indicating that and the other is, A uses the data in. Okay, you can think of that as a dash line or something like that. Okay?

07 – Diagramming KWIC Quiz Solution

So why don't you tell me about your first, first solution. What components do you have? Sure. So I started with five components, and then I added one towards the end because I realized I might need a distinction for this component. So I have a line, and lines consist of words. So there's two components I have. And, and in- So, so you have a component that holds the data for lines? Yes. And a component that holds the data for words? Yes. Okay. Go ahead. That have index, which is this object that holds all of our titles, which consist of lines and nodes. I guess, actually, I don't, I think titles may not be necessary. I feel like I'm saying the same thing. The index consists of all of our lines. So the index, the word index could either be a verb or a noun. So this is, this, you're thinking of it as a data structure? Yes. Go ahead. And then I have a, a system which contains this index of all of our lines, and then it also contains, or also uses our passes control to a circle or like a shifter that will shift those lines around. Okay, well, which of the components is responsible for doing the sorting? Okay, so I've worked that out. I don't have one. Okay, so one other one. Mm-hm. And so, the, the operation here would be the system passing over control to the, well, tell, tell me how it would work? What would be the, the steps? So the system would pass control to the index, or it would use the index to aggregate through all of our lines, all of our, yes, all of our lines that had multiple words in it. And as it's going through each line circler would then circle it in all the different formations that the line could be in so that the index will grow. So the, the system is doing the calling into the data structure index to get a, get out particular pieces, it then passes those to the circular shifter? Yes. Okay. And then a circular shifter produces

some results that are then passed over to the sorter. Okay. And this order does its thing and presumably there's an input process at the start of this and there's an output process at the end of this. Yes. Well the way you phrased one part, the circler doesn't necessarily have to know about this order. So the circler its, I'm thinking its only job is rearrange the lines and then now you have this index all the different arrangements in your lines. That index could then just work with the sorter and the circler doesn't have to know about it. Sure. And now, what is it that breaks the file into lines and the lines into words? I guess the, the system in this case would be the one that gets the file and then gives that off to the index or populates the index initially.

08 – Components

So, it sounds as though what you have is the system component is responsible for causing the input to be read in, causing it to be parsed into pieces getting storing those pieces into the line and word data structures, and organizing index. Okay? And then calling the circular ship to do his thing, and then ultimately calling the sorted view to do its thing. Now sometimes when you put a lot of responsibility for organizing steps and behavior and algorithms inside of one piece, you may want to break that piece into, into parts. So, this particular solution is similar to Parnas' approach which he called the Shared Data Decomposition. Well, we're the system into components based upon the functions they compute. And all components share access to the data, which is stored in, in, in memory. So you have a component several data structure components which are then accessible to the circular sorter and to the the circular shifter and, and to the sorter. And this solution solutions like this typically contain some form of what's called a master controller routine, which you have labeled as systems. And it's responsible for invoking the others and knowing what steps are, are in the process and that the typically in a situation like this, control flow dependencies, is, are organized or realized by function calls.

09 – Shared Data

And he has a, a diagram which has these pieces in it. It's somewhat similar to yours and it differentiates between subroutine calls which are indicated here by the lines with the arrowheads, the big arrowheads. And accesses to the memory which are lines with the smaller arrowheads. And he also breaks out system IO, that is the reading and the input and the writing and the output. So that's solution number one. Take a minute now and see if you can come up with another solution. Okay, so for my second solution, I've tried to decentralize some of this, because it sounded like the system was just too, too heavy. So, the system is still comprised of these components for parsing something to circle what we parsed, and sort and then display. So I have the parser, the circler, the sorter and the displayer, but I'm trying to treat it as if it's like a running through a process in which the system doesn't have to negotiate everything. So . So, a step at a time? A step at a time. Okay. So we start with the parser. The parser does its job, passes its output to the circler. The circler, then, creates all of our different, you know, shifted versions of the lines. That gets passed to the sorter. And then, the sorter sorts it alphabetically and passes that on to the displayer. Okay. This sounds very similar to, what Parnas calls the pipe and filter, solution to things, so let's take a minute and look at that.

10 – Pipe and Filter

In the case of pipe and filter, we break the system into independently executing components called filters. The filters are connected together using a FIFO, or first in first out queue, and

these queues are called pipes. So we have pipes connecting together the filters. Each of the filters is going to take a single input, which has become called the standard input, and it's going to produce a single output, which has now been called the standard output. The filters share the assumptions that the inputs and output consist of sequential files containing lines of ASCII characters. Can you think of a situation where you have a pipeline of filter components where it's non-ASCII? I mean, I guess you could pipe together Linux processes that have binary output, but. Yeah, but what- Useful. Well, okay. In terms of, like, a user interface, you wouldn't be seeing anything that might be useful. Well, it turns out that in doing image processing, okay, you put various filters along the way to deal with the processing of the images. Also in situations where there's sensor data and you want to filter out noise of certain kinds or select certain frequency bands and so on. It's binary data, but it's going through a filtering process. So although this does work in other situations, it's most familiar and most used in situations where there's text files. So as you've indicated, there's going to be filters having to do with circular shifting, and alphabetizing, and reading things in and putting things out. And one of the essential elements of this particular approach to solving things is there's no common data storage elements. We're just passing the solution along as we go.

11 – Pipe and Filter Diagram

And if we lay it out graphically, it looks like a pipe and filters. The filters are the components along the way and the pipes are the little lines connecting them. And, and, and this, this form, will indeed, solve the problem and yet it's much different than the previous one. Well, Parnas also laid out two other solutions, which I want to briefly describe to you and they're probably many more. Is there a problem necessarily because the pipe and filter doesn't have any central storage location that we're not keeping that data? Do we need to have good like logging systems, for instance, if we use that type of approach because the data isn't persistent, maybe like it would be with a shared data model? Well, okay, so I haven't really laid out what the requirements of the problem are, okay? And even in the shared data solution, that's in memory and it's going to go away when the process is over. Now, we could imagine adding in or being more explicit about what the requirements are and whether we need to persist them, okay. And let's, let's come back to that in a couple of minutes. Okay. Okay? First I'd like to go over a couple of other solutions that Parnas proposes. The next one to consider is called the Abstract Data Type or ADT solution and this is breaking the system into components based upon important data structures. So when we had the shared memory solution, that was breaking things into components based upon functions and likewise, the pipe and filter was more breaking it into functions. Here we're thinking in terms of the data first. We're going to hide the represent, representations of those datas behind abstract interfaces. That is we have a function called interface to it and how exactly we store this stuff away is all hidden from the other components. The components holding the data, of course, are also going to have some operations available to them. In a sense, this is a precursor to an object oriented approach. It's not, it was, it was first developed before object oriented languages became popular but many of the features that ADTs have, have been incorporated into object oriented solutions to things

12 – Abstract Data Types

In the ADT solution, we're going to have components for lines for characters, we're even going to treat the circular shifter, instead of being a verb, it's going to be the circular shifts data structure and there's an operation for computing the circular shifts. And similarly for the alphabetized versions of things. So we try to make everything into a data structure, the

components into data structure, and then have operations for computing the values in that data structure. We have input components, like before, output components like before, and a master controller that invokes the other components.

13 – Abstract Data Type Diagram

In the diagram, we have the master controller invoking the inputs and outputs. But the other communication is based upon more or less a need to know when the output needs some value, to be produced it looks to it's source which is the alphabetic. Alphabetizer or the alphabetized shift component. And it looks to the circular shift component, which looks to the information that was stored during the parsing process.

14 – Implicit Invocation

Parness' other solution is a little bit more subtle. Now, in this case we're going to coordinate the communication between the components using a technique called registration broadcast. Components requiring services which, we're going to call clients, express interest in state changes in components providing them which we'll call servers. And that requesting notification is called a registration process. When a server component announces that something, detects that something has been changed and announces it, it's going to announce it to all the registered clients and that's going to be broadcast. In this particular approach, servers don't know the identities of the clients. The clients called them and said, call me back, but I don't know who you are that I'm calling. And the uni, unit of notification here is the event. We have essentially the same components as before, it's just that their mode of interaction has changed, and, and is now implicit invocation based upon something happening.

15 – Implicit Invocation Diagram

The diagram is similar to the diagram we just saw, except now some of the arrows are going in different direction to indicate when, when the various components are being notified about the events that are there. So now we have four solutions, and we, we might ask the question why do we need four solutions. That's an excellent question and it depends upon ultimately how, how this particular program, this particular solution is going to be used.

16 – Shared Data Approach Quiz

To get us there, let's think for a moment about the strengths and the weaknesses of the various approaches.

17 – Shared Data Approach Quiz Solution

I think one advantage that we may have is because all of the different parts are sharing the memory, sharing the data. The porting may be easier, or like the interplay between components might be easier, but it seems like maintaining any kind of change for the system in the long run is going to be more difficult than if you had the components kind of isolated and the functionalities isolated. So, [COUGH] we have shared data, it's all in memory. That's going to be quite simple for all the components to get out the information. It's also going to be very efficient. Okay. There's no there's no function calls involved in getting that data, you just go and get the data, okay. On the other hand, if we wanted to change the way that that data is represented, every one of the components would break, all right, because they all have to know how to get the data out. So we have a plus with respect to efficiency and simplicity, and we have a negative with respect to resilience to changes in representation.

18 – Evaluation

With respect to the three other solutions, we have advantages and disadvantages as well. As far as the ADT, Abstract Data Type solution, it's very good, as far as maintainability and reuse. Those particular components could be used in other applications by just, just plucking them out. Remember, they've hidden away details. On the other hand because things are hidden away you have to invoke them through function call interfaces which might be more expensive, so you might pay a price in performance. With respect to implicit implication, okay it has also maintainability advantages. If you change the representation because the, the, the clients and the servers don't know much about each other you only have to change them in one place. You don't have to change change the others, which also facilitates reuse. On the other hand, because it's implicit invocation and you don't know who you're talking to a lot of times, it's sometimes difficult to think about or control what's going on. And if you had to do some kind of debugging it might be tricky to know, you know, which of the components was responsible for some kind of problem. Also, as with the ADT solution, because you have these more or less abstract interfaces between things there may be a performance hit. With respect to pipe and filter, pipe and filter is very intuitive, easy to think about. It's also easy to reuse because each of the filters along the way, you can plunk out and put into another, another solution. On the other hand if we wanted to make changes, such as making the system interactive pipe and filter wouldn't work at all. It, it's going to stream things, stream things through. also, it's not particularly space efficient because you have no no place for you to store the data, which means you might have multiple copies of that data floating around as you're processing. So, each of the particular solutions has advantages and disadvantages, and in any particular sit, situation you want to look at what's important to you. Is performance important? Is memory footprint important? And, pick a solution that has the particular advantages that you need and avoids the particular disadvantages that might bother you.

19 – Enhancements Quiz

Another consideration is what's going to happen next. If you're building a system and you've done a good job and that system is successful, you're customers are going to want more. Fact of life. They're going to want enhancements and you can't really anticipate very well in advance what those enhancements are. Okay? If you could, then you could build your system in the first place so that it already had the enhancements in there, okay? So as another little quiz here see if you can list three ways in which this particular quick indexing tools might be improved. Three, three kinds of enhancements that you can imagine the customers wanting.

20 – Enhancements Quiz Solution

So, I think, I guess in today's day and age, somebody, a lot of customers I could see wanting some kind of GUI interface to be able to see this index. Okay. And then also, if we're going to have a GUI, there needs to be a smart, intuitive way to search through what we've just sorted, the keywords that we've just sorted. And, I think finally we want to a way to have that data persist. So if we wanted to add more titles later, remove some titles out, because they're outdated or something then supporting that as well. We certainly wouldn't want to have to go through the whole parsing, sorting, cer, shifting process, anytime anybody want to be using this. Mm-hm. Okay. So, in fact, there's a lot, there, there are all these needs and there's lots more. For example it may be the case that the form of the input changes over time. People might want to have input that if we're talking about titles, comes out of some bibliographic databases in a different format. We might want to use, re-use some of these

components in other applications. That's, that's a, a form of, of evolution as well. We might want to for performance reasons, or, or other reasons change the processing algorithm, so that we do the shifting of lines as they're read in or we wait until, doing the shifting until they're all read in. We might want to shift lines on demand we might want to use an incremental rather than a batch sort. That is, have some kind of sorted pre-structure that we add each title to as it comes in rather than when you get them all in and do a sort. You might want to add new functionalities such as we indicated before, in terms of stop words and eliminating those. We might want to support deletions, like, like you mentioned. We might want to use external stores, that is along the lines of persistence we might imagine the database on disc that holds these either in their original format or in some partially processed format. We might want to change the data representation. Imagine that we are moving to a different library to support our our in-memory storage. And we, so we might need a new representation of the lines and, and so on. Variety of changes. And the question then is, of the various approaches to the architectural breakdown of things, which ones are resilient to which changes? If you could anticipate the changes coming in, you could pick an architecture that, if not already able to provide that particular change would be able to easily adapt to that change.

21 – Reusability Quiz

So let me posit a hypothetical here. Which of the four styles you think would be able to deal with a change having to do with the reusability of the components?

22 – Reusability Quiz Solution

For the components to be reused, and potentially shifted, like you can shift their order around, or plug-and-play, it sounds like a pipe and filter solution would be best for that. Sure you can take any one of those filters and plug them into another application and as long as it had a single input and a single output and it was line oriented ASCII or ASCII characters you can you can imagine very easily very easily using it.

23 – Data Change Resilience Quiz

Well, how about this, which of the four styles would be least able to cope with a change having to do with a different data representation?

24 – Data Change Resilience Quiz Solution

With the shared data model, because everything is being shared by all the components, and you're kind of pre-assuming what it's going to look like, the data format, any change to that is going to cause a widespread change throughout your entire system. Sure. And I gave that one away before a little bit. But whenever you have shared assumptions, if you violate an assumption, everybody that depended on that particular assumption is broken.

25 – Deletion Quiz

And which if the four styles would be best able to cope with the change having to do with this interactive deletion of titles.

26 – Deletion Quiz Solution

I'm not sure after like, I can't think definitively for any of the, the types where it may be more difficult for that change to happen. Think about adding an operation into into the architecture to do the deletion. Is there one architecture which would be easy to find a place to do that deletion? With the, the abstract data type solution, because we have these well defined interfaces and what they can do, if all we need to do is add a delete operation to a particular, I guess, interface within that system, the ADT model seems to support, will be able to support that type of change. And it had a line data instructor right there, a line ADT. Mm-hm. Just an operation to delete a line. Okay? Right. It goes into a single place and because of the abstract interface, nobody else depends upon that operation taking place.

27 – Lessons

So to take away from this, the bottom line, is that there are a variety of different architectural styles that can be used to solve the same design problem. And, in order to figure out which one to use you should be aware that each style has it's advantages and disadvantages, and depending on the particular requirements, changes you have to deal with, you can pick the one that's best suited for your particular situation.

P3L02 Overview of Architectural Styles

01 – Introduction

With this lesson, we begin the second major unit of the course on software architecture. Which is you recall, is the highest level of expression to a design problem. In actual practice, software architecture and industry usually amounts to the preparation of a slide for display that contains some boxes and arrows depicting the major components of a system, and how they're connected together. We want to take a more principled look at what this essential aspect of software design is all about. Here is such a diagram. The boxes depict major components of the system, and the arrows indicate some form of dependency among the boxes. It might be the flow of control, it might be the flow of data. The point being is that the, there's no actual semantics to the diagram that is universally accepted.

02 – Informal Definition

Let's start with an informal definition of software architecture. It is the organization or the breakdown of the system in to component subsystems or modules. Architecture is almost universally done in layers. That is, there's a most abstract version, and then the components of the abstract version are broken down into sub-components, subsystems and so on, until we get to a level, a low enough level where things can actually be implemented. So, for architectures as we mentioned a second ago, often also make use of a stereotypical architectural styles and we'll be looking into those styles in this lesson.

03 – Analysis to Components Quiz

Here's a quiz. Assume we want to determine the components of a software system based solely on an analysis model. Given this situation, mark each text box below either true or false. First box, analysis models can adapt well to changes in customer requirements. Second question, analysis models should represent the approach that will be taken in design. Third, analysis models are resilient to changes in hardware. Fourth question, analysis models include all components required by a system.

04 – Analysis to Components Quiz Solution

Here are the answers. First question. Analysis models can adapt well to changes in customer requirements. This is true because analysis models are constructed before design, therefore they should not be affected by design constraints and can adjust more easily to changes in customer requirements. Second question. Analysis models should represent the approach that will be taken in design. This is false. We actually want to avoid mixing analysis and design together or else we might bias the design approach that is taken. Third question. Analysis models are resilient to changes in hardware. This is true. Analysis models should not make assumptions about the running environment of a system and can adjust to changes in hardware. Fourth question. Analysis models include all components required by a system.

This is false because there will likely be additions to a system for collection classes or other types of utility classes that an analysis model would not specify

05 – USP Definition

Here is a somewhat more formal definition from the definers of the unified software process. I'm not going to recite it for you, but I will mention several key elements. One is architecture is all about decisions, choices that the architect makes about how which, which components are there, how they interact, and how the non-functional requirements are being dealt with. As far as the components are concerned, these are structural elements, and they're interfaces, that is what they provide to the rest of the world, and what they require from the rest of the world. Components interact, they, they engage in collaboration with other components, and it is the composition of these structural and behavioral elements into larger, and larger subsystems that form the overall architecture. This structuring, this, this composing may be guided by architectural styles, that provide guidance as to or bring in experience that others have had with building similar systems. The decisions about software architecture, concern not only the structure and behavior but other important elements, such as usage, performance, comprehensibility, understandability, economics, technology constraints and trade-offs and aesthetic concerns.

06 – Other Definitions

Some other definitions we want to, to pull from during this lesson are, one from Dwayne Perry and Alex Wolf, that involves elements, forms and rationale. Rationale being the set of dis, the, the decisions and reasons for making them, that the architects have agreed on. Obviously, architecture involves the fundamental organization, components and relationships. This comes from the IEEE definition. Another element from Verhoff is the determination of what makes up the components here based upon hiding away those things which are hardest to change. That's a little bit different way of thinking about a system. But if you imagine what the system is going to be like several years after its initial release, it's going to change. And those changes have the potential of breaking the system in unexpected ways. So by hiding away those tough decisions we can help reduce the overall maintenance cost downstream. And then for the rest of this lesson, we're going to be guided by the Garlan and Shaw paper which is listed on the class resource page, and they talk about architecture in terms of its components, its connectors, and its configurations. What do they mean by these three terms? Well, a component is a computational or a data element, plus its interfaces, which they call ports, interfaces to the rest of the system. The interfaces express what the component requires or needs from the rest of the system, and what it provides. Recall from the UML component diagrams, this is exactly what the interfaces, represent. A connector is a, essentially a communication protocol among components, although it may have code associated with it for enforcing that particular protocol. It is its, its major element of defining the character is that protocol. And then configuration is how you put those pieces together. You plug a connector into a component. You plug the other end of the connector into another component if it's a binary connector, and the ports then can talk to each other using the connector. That overall topology for the pieces is called a configuration.

07 – Components

A couple of takes on components one from Richard Taylor, a software component is an architectural entity. It's concerned with a unit of the system's functionality or its data. Once again, key here is the interfaces that it provides to the rest of the world, and, according to

Taylor, the dependencies on its required execution context. What that, what that means is, what does it take in order to enable the component to run in a manner that it should? Szyperski offers the following, that a component is a unit of composition. We're going to take components, we're going to put them together with contractually specified interfaces. This means that the interfaces are explicit, the other components know about it, they agree to it, and that these interfaces when you're putting things together in configurations can be checked and enforced.

08 – Selecting Components

The obvious approach to selecting components is to say, well what is it the system is supposed do or compute and break that down into, into pieces. However there are many other factors that might go into deciding which components are going to be part of your system. Of course, required functionality is, is most important there, but it may also be the case that you already have some existing reusable components from your libraries that you want to build into your system. And that may have a important role in deciding on the overall component structure as would the physical machine architecture, that is the architecture providing you multiple cores in which case how can you take advantage of those in, in breaking down your computation. Another element you might not have thought of is your staff, that is the people who are going to do this. Conway's Law say's that the ultimate structure of a system depends upon the structure of the organization building it. So it's well to take into account that if you have three people helping you lay out the architecture you're likely to end up with three major components. Another important element is that for real systems they're going to have long lifetimes. And the trajectory of that lifetime, the direction in which it's going to move as we saw in the Brohoff definition, could and should strongly influence the components into which you divided.

09 – APIs

A word about API's. I've mentioned it requires and provides part of the components description. This is sometimes called the component's application programming interface or API. If you look at documentation for systems at the level of Javadoc for, let's say a bunch of software you may download. The description of the, of the classes and what their methods are and so on, are this, is that, is that classes API. It's going to include the names by which you can refer to the elements of that particular unit for example, the method names, the arguments you, you provide to that component, and, and their types, the return value and so on. The API could be specified in a particular programming language. If, if it's, if that's the case, it's called the language binding. It might be described that a higher level of abstraction, such as using OCL. And a little later were going to look at specialized notations for describing APIs at the architectural level called Architectural Description Languages, or ADLs.

10 – Connectors

That's what I wanted to say about components. In a sense components are easier because, you're going to, devise them in terms of the functionality, and the functionality is, dominates what's in the requirements, specification. Connectors are trickier. Okay. Connectors, are, where the designer has to make some specific choices about how to deal with problems. Taylor's definition is a, a connector is a, a software connector is an architectural element tasked with effecting and regulating interactions among components, the piping between those components. The key way that I like to look at connectors is they provide a protocol for interaction among those components. A protocol is a kind of a language saying who speaks in what order. What

information is passed back and forth, and what to do if something goes wrong. We're going to later in the course, devote a whole class to discussing connectors.

11 – Example Connector

As an example, the simplest connector I can think of is a procedure call and return. This is a pair of messages. The first one you're calling some method, and second you're getting every turn value passed back. This is an asymmetric relationship, okay, that is the caller, okay? One of the two roles for the, for the connector is caller and the other's callee. The caller waits, once he's issued the call, for the callee to re, return. Okay? It's a synchronous relationship because the caller blocks or. Stops any further computation. The connector also allows for the passing of information in terms of typed parameters and the second message may include a return value, also a typed value.

12 – Configuration

Assuming that we have the components and the connectors, now we need to wire them together. And we call that a configuration. It's a set of specific associations between the components and the connectors of a software system's architecture, according to Taylor.

13 – Terminology

There's some other terms related to architecture that I"d like to mention for a minute just so that if we come across them later, you'll know what I was intending. The first one is conceptual architecture. Obviously, the word conceptual connotes that it's vague or high level. The reason is fake or high level is that it's often produced very early in the development process, in fact before you may even have a complete idea of what the requirements are. Conceptual architectures are often produces a way to begin the planning process. Okay, by having an idea of what at a very high level the components and connectors are going to be, you can begin to block out what the teams might look like and how long it's going to take to produce the ultimate program. A pair of other terms to be aware of is the As-Intended versus the As-Built architecture. During the planning process, the architectural planning process in which the architectural team decides on what the architecture is going to be and produces some documentation for that, the result is the As-Intended architecture. However, during the course of actual construction of the program, something else may be built, and we'll call that the As-Built architecture. There are several reasons why the As-Built may not match ideally with the As-Intended. It may be the case, for example, that during the course of refinement, the development team comes across a available component, whether it's open source or from another group, that can short cut the development process by providing some needed functionality. But that additional piece may not match identically with what was intended in the architectural plan. This process by which the As-Intended becomes the As-Built is sometimes called architectural drift. And if it happens during software maintenance, that is, after the program is released and the maintenance team then is dealing with bugs and enhancement suggestions, the term is sometimes called architectural erosion. This may arise because the maintenance team, under time pressure to get the fixes out to the customers, may not make the ideal fix that would be done if, if the original development were done in a way that was aware of this particular problem or enhancement. and, perhaps also didn't go back and make the appropriate changes to the architectural documentation.

14 – Architectural Views

In another lesson later on we're going to be looking at architectural views. But to anticipate that I'll just mention that architectural description is not just a diagram, it's a set of decisions. And in fact the, in order to fully communicate that set of decisions, many diagrams and/or textual documents might be produced. We call these architectural views. Because the set of decisions may be large, and there may be many different aspects to it, okay? Over the course of time various different kinds of diagrams and tables have been developed and found useful and so we want to be aware of what those are so if you are confronted by a situation where you need to convey some aspects of the architecture, you have you're aware of the various diagrams and, and textural processes you could apply.

15 – UML Diagram Quiz

In earlier lessons, we reviewed UML and recall that there were lots of different diagrams that UML provided. Some of those might be useful for conveying aspects of software architecture. See if you can list some of the UML diagrams that might be appropriate and place them in the text box.

16 – UML Diagram Quiz

Well it turns out that most of the different kinds of UML diagrams could be so used. Of the 14 diagrams I've listed here nine of them that might be, might be appropriate. Of course a given diagram, like a class model diagram, might be useful at a very low level that we wouldn't even call architectural. What it also could be used for lending out what the major classes of a system are. As with some of the other diagrams they can convey the structural elements, but also some of the UML diagrams can convey the behavioral aspect. For example, sequence diagrams and communication diagrams. And at a most abstract level in terms of dealing with a systems overall usage and how its going to be broken out into different aspects of functionality, use case diagram could be useful for those circumstances.

17 – Architectural Styles

Most of the remainder of this particular lesson is going to be concerned with architectural styles. As with buildings, software systems come in, in, of different types. Okay, we call those types architectural styles. Taylor's definition of an architectural style is a named collection of decisions. Those decisions are appropriate in a particular circumstances that is dependent upon you know, the system specifications and its major concerns. The design decisions constrain, what are the possible components and interactions and by using. The architectural style. You get various benefits from it on the ultimate system you're you're building and the process of building it.

18 – Arch Style Quiz

For this quiz, I'm going to list some decisions and you tell me what architectural style it might apply to. Don't worry, it's a fairly commonly used architectural style. For this particular kind of system, we're going to have software components that are physically separated, that means on different machines. Some of the components are there to request services from other components that provide those services. By so doing, this allows for scaling. In the situation where the number of requests grows over time. We're going to have that the service providers are unaware of the identities of the service requesters unless those, service requesters provide that identity information. We're going to in this particular architectural style it isolates the

requesters from each other. They're aware of the service provider or providers but not each other and we're even going to allow for multiple service providers the number of which may grow dynamically depending upon the demand for the services that are there. Can you name this particular architectural style?

19 – Arch Style Solution

Of course, it is the common client-server architectural style. The server is usually a database server. They may be some business logic associated with it, and the clients these days are typically on web browsers making requests over the internet to the database in business logic.

20 – Architectural Styles cont

The benefits that Taylor eluded to included, include the fact that what we have done by documenting an architectural style is encode our experience on it. For example with the client server there are certain ways of, of dividing things up and, and connecting them together. That, work better than other ways. We also know, with client server what kinds of problems can arise and, how we can best cope with those. And having that knowledge then allows us to, reduce our overall development effort because we're not stumbling down blind alleys. Architectural styles can also be encoded into Standards. Standard sometimes call reference architectures. And those Standards can then support the validation process, the way that we check whether our architectural solution is, is a good one. Architectural styles can also support Reuse. The fact that there are, all kinds of, client server. Systems out there means we maybe able to make use of standard components such as [UNKNOWN] database server. And because different styles provide different ways of, of structuring the development process, we may even, even be able to. Use the, the style to guide us in, in what our groups should look like and the steps that they should take when validation can come and so on. What I'd like to do now is have a look at some of the different architectural styles that have arisen over the years

21 – Catalog of Styles

What I have here is essentially just a big, long list. And I'm not going to go into all of them, but I will make some, make some comments that are appropriate to a few of them. The idea for throwing these at you is that as I said, the key to software design is having experience. Experience means being aware of possible solutions, and here's a catalog of solutions that have been applied in certain circumstances in the past. In the KWIC quick exercise that you undertook you saw the abstract data type architectural style as did the, as you did the batch sequential one. Blackboard architecture is one in which the various components post their results and their requests on some kind of common data repository, and the other components look at the repository and see if there's anything they can react to. The fourth one here, the big ball of mud is not really any architectural style it's an absence of one. It usually arises of because of the process of architectural erosion, or because the team didn't even have an architectural design process in the first place. We also have already have mentioned client server. We'll talk about component-based systems later in the, in the course. This use of the term component is somewhat different than the one we've been using in this particular lesson, but we'll make that clear when we get to it. You may not have heard about coroutines. So I want to ment, take a, take a second to mention that. With subprograms or subroutines we mentioned that there's this asymmetric relationship. There's a caller and a callee. With coroutines, it's a symmetric relation. Okay? A can call B and B can call A. Okay? Moreover, if A calls B for a second time, B continues from the point that it was last at when it returned from the first call. These are called coroutines. A primary example of coroutines. Think about printing out

formatted data. With printing out formatted data such as with print F, typically you have a a list of formatting information and a list of data items. And you, the implementation proceeds by taking a, the first formating information and the first data item and then connecting them together. Then getting the second format information information and the second data item, and there may be some loops involved, some formatting information may allow for multiple occurrences. Moreover the data provided maybe in the form of a loop. So we're really going back and forth between these two streams of information, and a coroutine is a perfect a perfect style for dealing with that kind of situation. If you've got a sequel database and you've got some experience with this, you know that you can include in your standard sequel some other functions that you've written. If you do that sometimes the architectural style is called data centric, you, that is, you're using stored database procedures. In this course we won't be getting into domain modeling very much but there is a architectural style called domain driven design. And here, by a domain we mean a kind of application program. So think about, for example, tax processing software. With tax processing software, there's certain vocabulary that everybody's familiar with such as deductions. And there's typical ways of solving problems. So, if you've ever used your TurboTax or other tax preparation software, you know if you change something over here, other things will get changed automatically for you. That style of data flow updates is inherently part of the tax preparation software application domain. And so by organizing your tax appropriation software using this particular domain architectural style once again you can save yourself effort. We're going to be looking more extensively at implicit invocation. And, and also in the Garland and Shaw book that's listed on the resources page there's a very nice section that talks about all the possible options for implicit invocation architectural style. Another very popular one is layered architectures, in which each layer in the system acts as a virtual machine, providing capabilities to the layers above it.

22 – More Styles

Historically probably the first architectural style that became pervasive was called the Master Control. That issued right at top level routine that was responsible for organizing the use of the lower level routines. Some of the other ones listed on this particular list listed here. Are more recent message bus is an analogy with the bus, the hardware bus that organizes computations on a chip message bus often means asynchronous message passing over some common data channel. With your smartphone or other mobile devices there are a set of constraints that you have to deal with. That you wouldn't have to deal with in other kinds of applications. So architectural style in support of mobile code. Where there might be remote remote evaluation and you have agents of various places on a network is an example of mobile code architecture and style. The term object-oriented architectural style is a little bit different, than, object-oriented programming or object-oriented programming language. But with the object-oriented architectural style, we still have objects, but each of those objects have an independent existence that is they're running all the time, they have their own thread of control, and they're sending message to each, messages to each other. Assynchronous messages. This will allows them to cooperatively address a, address the problem being solved. Peer to peer network, you may have heard of. Here there are equal parties sharing responsibility for providing whatever services. Plug in architecture. If you are familiar with some interactive development environments like Eclipse, you know that there's a whole registry of available additional functionality that you can plug into Eclipse. And the mechanism for doing that is a very powerful way of adding extensibility to systems. Pipe and filter you've seen before with a quick a quick exercise. One you haven't seen probably, is process control. Think here, nuclear reactor. Think here, your speed control on your car. The situation is you have some ongoing

hardware process and you'd like a corresponding software. Application to control that process. If the process is going too fast, you want to slow it down. If it's going too slow, you want to speed it up. This is called process control, and its key element is some kind of feedback loop. From the artificial intelligence world, there's production systems. These are essentially a collection of rules, and the conditions under which the rules fire. It enables the modeling of systems where we don't have a clear idea of what the control flow needed to implement the system. A very popular one these days is, is Rest. Rest stands for representational state transfer and you could think here it's those internet applications that are using HTTP. that, often have a client server type relationship, and that are stateless, that is each of the, user requests are handled independently, and some potentially some caching going on to improve performance. Service oriented architecture or SOA, is where we have carved up the functionality of the system into separate services. That is, from the users point of view, a service is a unit, a self contained unit of, of, of functionality and that means that we have to imagine the architecture of the system as being able to support a set of, a set of services. These are typically done in support of enterprise type applications, and often with internet connectivity between the user requests through some browser, and the ultimate service being provided by some server. Shared nothing is a term for a distributed database with no sharing across across the nodes. I don't have, personally have, a lot of experience with that one. Stay transition systems, on the other hand, are very common, particularly if yo have a situation where the system is driven by events, assynchronise events, and has to react to those events. A typical example is, if you've got a GUI and the user is providing the events. But it could also be some kind of real time system where the events are coming from the outside world. Shared memory, we saw from the, from the quick exercise and finally we have table-driven interpreter. For certain kinds of applications where the requests take the form of simple expressions in some kind of language we can deal with those requests by having a, an interpreter. The interpreter is essentially taking the request parsing it, and then invoking the, whatever procedure is required to deal with that sp, specific request.

23 – Style Issues

Although it may sound like, all the problems have been solved by just selecting the correct architectural style, there are still some issues. One important one is that for real systems, big systems it may require more than one architectural style. We call that a Heterogeneous system, you can imagine for example that the. System might have some client/server elements that might have a GUI make it reactive. It may be the combination of a variety of, of, of of different approaches and the systems, you still have to have an architecture so that, so you have a single concept for how the system is going to work. Secondly. Some situations, although they call for having an architectural style, style are very domain specific. For example, imagine in military context of a particular kind of airplane. It may be the case, it is the case that that airplane comes in a variety of variants. However, the control systems for the airplane is pretty much the same across variance. That is, it shares more than it differs. In this situation, we call it a Domain-specific software architecture, or DSSA. Another term sometimes used is reference architecture, that is the reference architecture describes what's common. And then for any particular variant, the architecture responsible for saying what those variants are, and how they're going to be dealt with. The third issue is one of semantics. It's easier for me, it's easy for me to lay out and say, a client server is XYZ. But what exactly does that mean? It's important, and as the field of software architecture evolves, to get more and more precise definitions of what these styles mean. Which will then enable, reuse of, of existing solutions.

24 – Architecture Description Language

I hinted earlier in the lesson about architectural description languages. These are, as it sounds, notations for describing architectures. They provide an extra modicum of formality and precision, that goes beyond just having a diagram. Moreover, by having that extra precision, it then enables some tool support. For example, building to diagram the tool. Diagram a solution in such a way, that the diagramming tool can check for whether you've done things well. Analyzers, okay, for determining the structural properties and behavioral properties of your system. And even simulators. I've listed here some of the popular architectural description languages. The one that we'll be looking at a little bit later is called Acme.

25 – Architectural Evaluation

The final thing that I wanted to mention in this overview of software architecture, is evaluation. It doesn't do you that much good to develop this fancy architecture, if it's not the right one. Okay? So, we need some process by which we can judge the correctness, completeness, consistency. And other quality aspects of the architecture we've produced. Because of its, the importance of software architecture on the ultimate product being developed, it's key to get it right. Because if we make a mistake, it's very costly to make a change. So, some approaches have been developed for dealing with architectural evaluation. One of those, is architecture review boards. That is, for large systems which are developed by multiple teams. Or, maybe systems of systems. Making a change can have impacts on various unexpected places in, in the ultimate system. And so it's good to have the stakeholders, particularly development stakeholders, sit down and evaluate the impact of those changes. Some organizations have, formalized this into a periodic meetings. That review suggested for architecture, suggestions for architectural changes. Also some, so, some evaluation techniques have been developed. Software architecture assessment method, or SAAM. I'm going to show you a slide on this. It's, is a relatively informal one. More formal one developed at the Software Engineering Institute, is the Architecture Tradeoff Analysis Method, or ATAM.

26 – SAAM

Here's a sketch of SAAM. Assuming that we have already gone through an architectural design process and if we produce some artifacts like diagrams. So we generate that architecture. We also generate some scenarios. Now these are not primarily usage scenarios, or we could think of them as, as elaborated usage scenarios. Where instead of looking at it from the outside, external view, we're looking for it internally. So for example if, if the external request is by the user to compute some result, the generated scenario here that we, the team provides is which elements of the architecture are required. To, be involved in providing that functionality. And in what order. That is, we essentially are going to, walk through the diagram and see how that particular usage of the system impacts the architecture. This is particularly important if what we're talking about is a new scenario. That is imagine the system that was architected one way and we want to add a new a new, a new functionality to it. Imagine also that maybe there are thoughts on different ways of doing this. So if we had alternative proposals. Each are provided in the form of some kind of diagram, we could walk through the two diagrams and see which one, in which one, there's more impact of the change. We might want to go with the solution which has the, the lesser impact. And so imagine this all happening in some kind of design review meeting, in which we systematically go through that proposed changes. The proposed architectural responses to them. And use the gathered information to come, to come to some kind of conclusion about the way forward.

27 – Summary

Well, the intention of this lesson is to introduce you to the topic of software architecture. That we're going to be looking at in more detail in subsequent lessons. The ultimate goal of course, is to produce high quality systems and reduce the cost of producing them. The key way of doing that is the early detection of problems. And the key way of detecting things early is to try to layout in advance, what the, what the system is going to look like. You want to have explicit recognition of what the issues are, explicit rationale for how they are being handled. We want to be able to on the productivity side to deal with any existing assets we can apply toward the solution. And we want to be able to construct an architecture at a sufficient level of abstraction. That it can be used to convey all these idea quickly and effectively to all the stakeholders involved.

P3L03 Architectural Views

01 – Architectural Views

Building architects use sketches and blueprints to convey the architecture of a proposed building. Similarly, software architects use a variety of notational devices called views for the same purpose. It's well to remember though that an architecture is not simply a picture, it is a set of these important design decisions made during the course of thinking about how the building or the software system is going to solve whatever problem it's suppose to solve. Specifically, software architecture conveys the set of components that will together compute a solution to the problem while satisfying or not violating any of the require non-functional constraints as specified in the requirements for the system. In this particular lesson we will look at a variety of different graphical and textual approaches for conveying software architecture views. We will start with Philip Kruchten's 4 plus 1 architectural views paper which I asked you to read. And then we will add in some different views that go beyond what he was talking about, the feature view, non-functional requirements, bug reporting, context, and utility views.

02 – Logical View

The first and probably the most popular of Kruchten's five views is the logical view. This conveys the structural breakdown of the computational, communicational, and behavioral responsibilities of the system. There's lots of ways of conveying logical views. Probably the most frequently used is a box and arrow diagram. We'll see a slide in a second. You've already encountered some of the UML diagrams that can be used to convey the logical view, including the class model diagram, interaction overview, and collaboration diagrams. And also, we will be seeing the components and connectors that are part of architectural description languages. Here's a random box and arrow diagram. It contains boxes which indicate the, the the functional components of a system, and arrows connecting them indicating some kind of dependency between the boxes. Box and arrow diagrams have the benefit of being familiar. And usually can convey a whole lot about the high-level architecture in terms of a single slide. However, they have the disadvantage of being imprecise. Just what is it that those arrows mean?

03 – Developmental View

The second of Kruchten's views is the developmental view, and here we're concerned with the source code. The logical view had to do with the system, primarily with how it's going to execute. The development view has to do with the source code. And the units of source code which might be considered for modeling in the development view include packages, classes, subsystems, libraries, files, and so on. UML package diagrams, UML component diagrams can convey this sort of thing, as can the mechanisms provided by source control systems such as CVS or, or, or, or some of it's SVN or, or Github or some of the more modern modern systems allow the developers to break the system into modules. Here is a reminder of some of

the slides that we saw when we were reviewing UML diagram types. Here's a class diagram with packages, here's a package diagram, and here's a component diagram where the rectangles correspond with the components. The the arrows correspond to specific dependencies among the components indicating that a component supplies what is needed by a component at the other end of the line.

04 – Diagram Types Quiz

Quiz for you. Match the diagram in the first column with the text in the second column that most closely describes the meanings of its lines.

05 – Diagram Types Quiz

The lines in the component diagram are going to indicate that one of, one of the components makes use of or calls upon the resources of another. As far as the package diagram is concerned, that's an indication of importing, particularly the names, or a sub-component of, or is merged with. And finally, the class diagram, of course, the particular lines on the class diagram correspond to generalization, association, or dependency.

06 – Process View

The third of Kruchten's diagrams is the process view, and here we're very specifically getting into what concurrently executing processes or threads exist and how executed, execution is divided among them. Primary means for conveying this is the UML deployment diagram. Here's an example that we saw before where there are two major concurrently executing components and some indication of how they're communicating with each other

07 – Physical View

The fourth of Kruchtens' views is the physical view, and this is very close to the previous one. Here however we are concerned with how the processes are allocated to the various execution units. And we can use the deployment diagram that we just saw, or we can use a sequence diagram here. Here, as a reminder, is a sequence diagram. The columns correspond to objects, which could be running on separate processors. Recall that objects are like classes, except the names are underlined and there's usually a colon separating the name from the class name. Going down a column corresponds to the passage of time. And the lines that cross it indicate messages being sent. So here we have the, the coordination of three particular processes dealing with the handling of a transaction. Collaboration diagrams had the same content as in sequence diagrams, but they're laid out differently. Here's the corresponding collaboration diagram for the previous sequence diagram. Same objects, same messages. The numbers indicate the orders in which the messages are sent.

08 – Use Case View

The plus 1 in Kruchtens' 4 plus 1 is the use case view. Use cases are important execution sequences from the external actors or user's point of view. We'll have a look at the use case diagram that UML offers. Some other UML diagrams that can be used to convey individual use cases. And we'll even see some structured text that can convey a use case. Here's a use case diagram from UML. Each of the ovals corresponds to a use case. The stick figures correspond to external actors. Some of the lines indicate are, are labeled and indicate that the particular use case is used for a special purpose such as being included in another use case or being shared

among several use cases. Important thing is that the use case diagram in UML conveys a set of use cases, not an individual use case.

09 – Context View

Going back to our historical overview of, of modeling. Recall OMT, the object modeling technique. And it had both structural view and behavioral view, but it had a functional, and the functional view made use of what are called data flow diagrams. UML does not contain that particular functional view. It uses use case diagrams instead. But I wanted to show you the, the data flow diagrams because I have found them personally useful in describing describing systems. In particular, a data flow diagram conveys systems activities and the ordering in which they occur. Data flow diagrams can be nested, and the outermost data flow diagram is called a context diagram. In the context diagram, there's a single oval which denotes the system as a whole. That oval can be connected to various external actors. In the case of the context view, those aren't stick figures. Instead they're rectangles. And the actors can be individual users, or they even can be external systems which are communicating with the system that you're modeling. The lines that connect the actors to the system are called dataflow lines, that is there's some communication of data between the actor and the system or the system and the actor. Here's a context diagram of a system that plays chess. The external actor is the human opponent and there are three flows of data. The human opponent can submit moves to the to the chess playing program. Similarly the chess playing program can communicate moves back to the human opponent, or the chess playing program can produce a diagram of the current board situation.

10 – Individual Use Cases

I wanted to spend a moment talking about individual use cases. Individual use cases are extremely valuable in understanding expected user behavior. They're also used in, in the development shops that use agile methods in doing some of their planning and scheduling. What it essentially is is a story illustrating a specific act, interaction between a user and a system. I've listed here a story about buying from Amazon. It's written out as just narrative text. And there's no structure to it, but nevertheless, it's a way of expressing, in a very user centered form, a system requirement. We could take this same story and also express it in a structured way. Here in tabular form is the same story, in particular there are three columns,. The column on the left is the actor, okay? And there are two actors here, the user and the Amazon website. There are various actions that could take place. And sometimes, optionally, there are objects, okay, that are used by the actor to perform the action, or produced by the action, and those are in the right-most column.

11 – Feature View

Well that's it for the four plus one views, but I think there are some other views which are sometimes useful in understanding, or conveying your understanding, of a system that you're trying to build at the architectural level. One that I've found quite useful is a feature view. A feature is a conceptual unit of system behavior from the user's point of view. Your camera has a zooming feature available to it. Or maybe it doesn't, okay? Typically, features are something which manufacturers provide as options, that you may have to pay extra for. Feature modeling is used for describing a set of features that a a collection of related applications provide. There's feature diagram, which I'm going to show you in a second, that conveys the set of possible features that might be configured into a particular product in that set of products. The diagram is graphical. It has icons and so on, but it can also have intra-feature constrains which aren't

shown in this particular diagram. Here's a particular feature diagram showing the features of a car. At the top is the, what's called the concept or the main manufactured item and then under it are its features. So the car, this particular car has four features. Okay. Three of the features are required. The body, the transmission, and engine. And whether or not it pulls trailer is optional. Whether it's required or optional is indicated by the little circle at the end of arc connecting cart to its features. If the circle is filled in it's a required feature, and if it's open; then it's an optional feature. Features can have sub-features. So, a transition sorry, a transmission can have an automatic transmission feature, or a manual transmission feature. The or that I just said, is indicated by that open arc connecting those two lines coming out of the transition. Similarly there are two kinds of engines, electrical and gasoline. In this case the arc is filled in though, indicating that you can have a combination. That is you can have a hybrid engine. We could imagine a constraint here going between automatic transmission and pulls trailer, saying that the pulls trailer option is only available if you have the automatic transmission and not the manual transmission.

12 – Feature Diagram Quiz 1

As a little test for you, take a second and figure out how many different possible cars are expressed by the feature diagram that I just showed you on the previous slide.

13 – Feature Diagram Quiz 1

The answer is 12 cars. To find this answer we can multiply the possibilities together. First, we know a car body is required, and it has no sub-features. Then for the transmission we have two options. Either the transmission is automatic or it is manual. For an engine, there are three options. Electric, gasoline, or a combination of electric and gasoline called hybrid. Finally, for the pulls trailer feature, we can either have the feature or exclude it. Thus, we can multiply these numbers together: 1 times 2 times 3 times 2, which equals 12, for the 12 possible car configurations.

14 – Feature Diagram Quiz 2

Second question is, which of the following are valid configurations for cars? Car having a sunroof body, automatic transmission, gasoline engine, and pulls trailer. And second, a sedan body, manual transmission, gasoline engine, and electric engine. Third, coupe body, automatic transmission, manual transmission, gasoline engine. And fourth option, number d, a hatchback body, manual transmission, and pulls trailer.

15 – Feature Diagram Quiz 2

Well, A is certainly allowed. You can have the sunroof and the automatic transmission, gasoline engine, and pulls trailer. And so is B. But in C, you can't have both an automatic transmission and a manual transmission. And notice, in D, we don't have an engine at all. That's not much of a car.

16 – Non Functional View

The second view not listed by Cruchen I consider to be quite important, and that's the non-functional view. In coming up a software architecture, it's youre responsibility not only to describe a system that's going to compute what it needs to compute, but also compute it in a way that satisfies those non-functional requirements. Satisfying non-functional requirements is hard, it often involves some kind of tradeoff, okay? So you need to be expressing what the

options are and how you decided to trade things off, and I'm suggesting here some tabular text. Here's an example, if you imagine a web browser and you say, what's it's major computational responsibility? Well it's displaying web pages. Quite simple. But if you actually look at the code for a web browser, it's filled with code having to do with managing caches. There are page caches and connection caches and image caches and so on. Okay? Why are there caches? Well, caches are a technique for dealing with performance constraints and non-functional requirement, okay? So it's important to relate that information, first of all, that there is a performance requirement, and second of all, what technique is being used to address that performance requirement, the caches.

17 – Non Functional Requirements Quiz

Here's a little quiz for you. Along with performance, can you name three other important non-functional requirements with which a web browser must be concerned.

18 – Non Functional Requirements Quiz

Well, we know if we're using that web browser to buy something from Amazon there are security requirements. Extensibility, we know that web browsers can have plugins to support different presentations of different kinds of web content and portability. We'd like our web browser to run on a variety of different platforms. I'm sure there are others.

19 – Bug Reporting View

Another view which you may not have previously thought of as a view is I call the book reporting view. If you've got a, a system you're developing most shops will have some kind of tool for reporting bugs. These are bugs during development or these are bugs after delivery, and those bug reporting tools often have fields which you fill in which indicate which component does the bug relate to. I'm suggesting that those components had better correspond to the architectural components in your other views otherwise there's the potential for some kind of confusion. Okay? Similarly, with respect to features, you would like the person who's handling the bug to be aware of, well, this particular bug arose when the following feature was being used by the, by the user. Here's a screen capture from a get based web browser interface into a source control system in which bugs are being listed with respect to a particular component of that system.

20 – Utility Views

The last view I'd like to talk about is not really a view but a miscellaneous category indicating that there's a lot of other information having to do with system structure that hasn't been conveyed by the other views we've seen. Okay? This has to do with supporting software in files which are, are, are part of the system but maybe aren't part directly of the executing software. Okay? And I'm suggesting that this information be conpeyed, conveyed in, in tabular text it includes things like installation scripts, log file analysis, statistical processing, the make files that are there the configuration files for different configurations for the system, any documentation. Okay, the project manifests which describe system structure as part of a, a large delivery package. And any supporting tools. Okay, when you're building the system all of these pieces have to actually be there to contribute to the build and hence they're part of this kind of grander architecture of a system.

21 – Conclusion

In putting this all together, an important point to get across is there's no such single, tangible thing as in the architecture. An architecture is a set of decisions. You can convey some of the information about those decisions with various views, whether they're graphical or textual, okay? And their sum conveys the architecture. For your purposes as a developer, you need to select the appropriate views depending upon the structure of the system, that is the complexity and the application domain of the system and the particular people that are going to be looking at those diagrams or reading that text.

P3L04 Text Browser Exercise (Arch)

01 – Text Browser Case Study

Today, we'd like to revisit the TextBrowser example that we started the class with earlier. And this time we want to come at it from the point of view of it's architecture. In particular, we want to suggest a process that might be useful in actually performing an architectural design. Note that this lesson uses UML diagrams and constraints to describe system architecture. And there are, there is a paper on the class resources page by Stirewatt and Rugaber. That that comments on several of the points we're going to be making.

02 – Text Browser

We call the following problem that we considered earlier. You have a source of textual data, a Document, with a file system interface, which we call the FileManager. You have a resizable viewing window resource capable of displaying lines of text which we call the ViewPort. And you have a controlling device which we call the ScrollBar capable of selecting a discrete value via handle, where the purpose of it controlling which lines have actually appear on the screen. The objective of, of this case study is to specify the properties of a text browser, choose an architecture, and assemble the components together

03 – Exercise

To construct this diagram, you should take the following steps using a UML class model diagram as notation. First of all, indicate external actors but only one activity. That is, there's only going to be one class in here and that's the class of the text browser itself. Indicate in the external stimuli or events that can affect the text browser. And indicate how the text browser communicates it's results back to the external actors, it's percepts. So you want to take a crack at producing such a, a, a diagram that lays out the text browsers relationship to the external actors. Okay, so I have I think kind of what are the things that you can do to the system. Okay, let's, let's start with [COUGH] what are the actors, the external actors? So for the external actors, I've shown drawn the user using the text browser, but also I believe we mentioned previously that the operating system is also an actor, even though you may not think of it visually, like, see that happening. Okay. So there's two external actors in here. The end user and the operating system which is going to supply the actual file contents. There's the the classic self which represents the system, okay. And you also have some events okay. What events do you have? So, an event that the user would, start with the user, what the user can do with a text browser, view the text browser, can move the handle that's in the tray, and then also resize the text browser. So which one of, which ones of those are events? events? The movement of the handle's an event, the resizing of the window's also an event. Viewing's kind of like a continuous process, there's no kind of instantaneous event that's happening. Well but the system is not doing the viewser, viewing, the use is doing the viewing, so. Correct. That's actually a percept of the system right. Okay. It's something that the system provides for the

user. In addition to the actual lines of file, what else is the system communicating back to the user? We're also communicating back the size of the handle. Something that we're seeing. That, that says something about the proportion of the file that's visible. Correct. And we're also communicating back to the user the height of the view port. Okay. Obviously the the user has control over that height. And the system is going to give feedback by showing a view port with different heights on it. And you mentioned also that the contents, and that's clearly a percept as well. So we have the height of the viewport, it's contents, the size of the handle, and the position of the handle on the tray is another percept. Correct. Okay, and we have the two events, the resizing event and the scrolling event. We have the two external actors and we have the class itself. Okay, so we can take the informal sketch that you've done, and we can lay it out using some kind of UML tool into a precise UML diagram that's got the single rectangle indicating the, the class labeled text browser. It's got some attributes which include the height, the the contents that are visible, the handle size, the handle position. Its got some external actors for the user and the file system. Its got some events indicating the user what the user can do to affect the text browser, and I've also put into the image into the diagram some real mode comments which are the rectangles with the dog eared corners on them. Note that there are some certain subtleties that haven't included in here such as how we're going to deal with zero length files and so on.

04 – Phase 0 Preparation

We divide this process into phases. Phase 0 is the preparation phase. And to begin, we will look at the TextBrowser from the outside in to determines its properties that have to be implemented. We will specify these properties using a context diagram. A context diagram portrays a single system. And the actors that interact with the system and the information that is passed between them.

05 – Phase 0 Summary

So, to summarize Phase 0. The goal of Phase 0 is to understand the system being built in terms of its relationship with its environment. And this means understanding important external actors and the interactions, the event interactions that they can have with the system.

06 – Phase 1

After you have a good idea of what the system looks like, from the outside, and how it is expected to behave, you can begin the architectural design process. This involves decomposing the system in to it's components, and allocating responsibilities to them. Now we did this before, when we did this exercise, but that was an analysis model. Here, when we're talking about act, act, architectural design we're actually going into the solution phase of things. Now, one of the features of object oriented development is that it's often the case that the particular elements that we come out of the analysis with, our pieces there, translate into pieces into the architecture and ultimately into the implementation. In addition to decomposing the system into it's components, we have to allocate responsibilities to the components, for handling those direct and indirect effects of the events, okay? And we're going to express that using OCL invariants and pre and post conditions. To indicate how those responsibilities are being satisfied

07 – Phase 1 Steps

So this is phase one. The steps we have here are the following. First of all, we're going to decompose the system into components. Secondly, we're going to allocate responsibilities.

Those include event handling, the delivery of the percepts and the guaranteeing of the properties that the system is required to have. And third, we're going to specify the component properties as OCL and variants, and preimposed conditions. So at this point in time can we assume components to be objects, when we're using OCL the syntax looks like we're referring to objects. So that's a good question. And it doesn't have an easy answer. Let me break it into pieces. So we're using to specify architecture we're using UML and OCL which are object oriented notations. Okay so that's one answer. The other answer is that object oriented programming languages have other features like inheritance. Okay and delegation of messages and so on, which we're not going to be involved with here. We'll get to those later when we talk about actually designing the objects, okay? But for now, we're thinking at the architectural level and we're going to think of these components insofar as UML and OCL are concerned. Okay.

08 – Decomposition

So the first step was decomposition. In order to decompose the systems into components we begin with the analysis model that we produced earlier. Note that the elements in the analysis model are good candidates to serve as architectural components. However in general, we might need to make some adjustments or add new components in order to deal with non-functional requirements.

09 – Phase 1 Diagram

So, in this particular slide we have the results of our analysis model. There were classes corresponding to our three major elements and then there were associations among the elements. There was one binary association having to do with displaying the contents. And there were other ternary or three part associations that indicated how the misc, how the three components worked together to make sure that the scroll bar affected the lines on the screen, and that the scroll bar handle and the position of the scroll bar handle were all right. Now, in analysis, UML supports the idea of associations. In design, there are no associations, in programming languages there are no associations. Instead there are in UML what are called dependencies. So part of our process here will be going and taking these associations and translating them into dependencies. Also, the comments which describe the guarantees in the previous, picture have been translated into or, they, they, they have the OCL constraints which we developed during the analysis phase.

10 – OCL Postcondition Constraint

As a reminder, what's now examined as one of the OCL Postcondition Constraints, specifying what happens when a user moves the handle. Remember, there are direct and indirect effects. And this, in here, we're talking about the direct effect. This particular constraint says that when the handle is moved, we expect the handle position to be in a different place. This says just what we would expect. After the user moves the handle, the handle is in the expected position. Note several things about the specification. For the first, because it's a direct effect, it's very simple. And that's exactly the kind of thing we like to have in an event handler. Event handlers have to be very fast, because there's lot's of events, and therefore, they have to do simple things and we'd hope that the OCL expression, which is specifying that, is similarly simple. Second it doesn't say anything about what is happening to the viewport. That's an indirect effect. This is the responsibility of that display association. And we will get to how we're going to deal with that in awhile.

11 – Another Postcondition

Here's a similar postcondition for resizing a window expressed in OCL. The context part gives the signature, and it's saying that the new size is, is an integer. The precondition is that that new size is greater than or equal to zero. And the post condition is that the height of the window, okay, which was the percept, is going to be that new size. So about as simple as you can get it.

12 – Third OCL Constraint

Here's the third constraint, having to do with the displaying of the document. In this case it's an invariant, it's an indirect, indirect guarantee. And it's relating the contents that are visible in the viewport to the data that's provided by the file manager, as far as it's document is concerned. And it's saying in, in, in effect that the sequence of lines that we see are determined by the handle position of the scroll, scrollbar, that is the top line we see is determined by the handle position of the scrollbar, and we're going to see subsequent lines that and the number of them is the, is the height minus 1. That is the number of additional lines plus the top line, that gives us height, which is what we would expect. What this says is that what we will see in the view contents percept is that subsequence of lines that the document, from the document starting with the line indicated by the scrollbar's handle position and continuing for height minus one additional lines.

13 – Phase 1 Summary

To summarize Phase 1, the purpose of Phase 1 is to divide the system being built into components. The analysis diagram is a good place to start. Similarly, the associations in the analysis model indicate the kinds of interactions that will have to occur among the components.

14 – Phase 2

In Phase 1 we divided our system up into components responsible for handling events and providing percepts. We also saw how guarantees were specified by invariants. Note that although the example invariants we saw was, were attached to associations, other invariants, invariants might be directly provided by the components themselves. For Phase 2, we want to determine the systems architecture. What this means is determining how the components will interact. This determination takes the form of selecting an architectural style. An architectural style is a generic pattern of interactions that can be used to to address non-functional concerns such as performance, reuse, or reliability.

15 – Phase 2 Steps

Here are the steps we'll employ for Phase 2 in the TextBrowser example. First we're going to choose an architectural style. For the check, tech, TextBrowser, we're going to choose a layered, implicit invocation architectural style. We're then going to assign the components to layers in the layered architecture. Typically users events are at the bottom and percepts are at the top. Now we're going to determine dependencies among the layers, and we're going to update the OCL into a, an equivalent, but what's called a constructive format in which there's a single variable on the left hand side.

16 – Text Browser Arch Quiz

Okay, so, layered implicit invocation is just one of many possible architectural styles, that you could think of. Can you think of some other alternatives that we could use here.

17 – Text Browser Arch Quiz Solution

Sure, so just relying back on kind of experiences I've had with web development and also some other GUI programs we could probably use a model view controller type of architecture? Sure, so model view controller is a typical, it came out of originally, the small talk world and is provided as a set of classes in Java Swing for example, very commonly used for GUI. What else do you have? Maybe something where we have, I guess what were using right now is kind of like registering events and callback. Does that? Yeah sure. So, I don't know, I think that what we've described here and then kind of in NBC are my first two, the ones that I gravitate towards. Okay, you can always find a way in these situations thinking things in terms of client server. That is, we have the file manager serving the file. We have the GUI components serving what the user is seeing and so on. So client server's always a possibility. Later on we'll talk about some other architectural styles, one that might be applied here is shared memory where all of the components are accessing the data out of a single place. Or another style, if you've ever been involved with Microsoft Windows Com architecture, component based architecture, we can try that as well. But for this exercise, we're going to stick with this layered implicit invocation architectural style.

18 – Layered Implicit Invocation

For this style we will organize the components into layers. For the higher level components register their interest in lower level events and are then called back when the events occur. In particular, the upper level components don't know the identity of the lower level components providing the events. The lower layers are going to handle the external events propagating status changes upward. So when the user moves the handle, that particular event ultimately has to be handled so that the user sees something different. So the event has to be propagated to higher layers into the architecture. Which then handle it to, to affect all those indirect indirect implications of, of the requirements of the system. Well, I should say that the upper layers receive these notifications and they prepare and present the results. This propagation of events is implicit, so we call it implicit invocation. Event announcement is made without the source component knowing the recipient which is, which reduces the coupling between all the components.

19 – Benefits and Costs

As with all architectural styles, there are benefits and costs. The benefits of the Layered Implicit Invocation architecture include im, improved reusability. Because the lower level components do not depend upon the upper level components, you can use them in other situations. So you can imagine taking. Our handler for the resized window and using that in other applications as well. There's also reduced complexity, because the, there are fewer the, the actual components know less about each other and everything is implicit but the complexity of the system can be reduced. Making it easier to understand and maintain. The, there is a cost, however. The cost is slightly increased overhead, that is performance overhead because of the extra levels of indirection. Whenever you have an indirection, okay, that means that there's a two-steps in, in resolving that. You make the call, and then, the call has to be. There has to be a call back and so on. For phase 2 after we have selected the style, we will assign the components to layers, determining the dependencies between the layers, and update the OCL. In particular we will insure that each constraint, is an equality with a single variable on the left hand side.

20 – Assigning Components to Layers

So first off we want to assign the components to layers. And it turns out that our simple rule, having events at the bottom and percepts at the top won't quite work for the TextBrowser. Both the ViewPort and the ScrollBar handle events and provide percepts. So they both can't be at the bottom. And they both can't be a the top. So for the purpose of illustrating the layered architecture, we will arbitrarily play the ViewPort on the top. Its percept is the most central one to the, to the user.

21 – Phase 2 Diagram

Here's a UML diagram we might come up with. On this we have the viewport at the top, the scrollbar in the middle, and the file manager at the, at the bottom. Notice that the, whereas in the earlier diagram we had associations among the components, here we've converted these into dash lines which in UML correspond to dependencies amongst, amongst the components. Dependencies are something which the ultimate implementation languages have many mechanisms such as procedure calls to implement. So we don't have to worry about vague concepts of associations. We can deal directly with, with dependencies

22 – OCL Updates

Next we have to worry about updating the OCL. Recall that in our previous diagram, OCL was associated with components and associations. The component OCL was used for specifying event handlers, and the OCL that, and that OCL will remain unchanged in the architectural diagram. In particular here are the two constraints that we had for dealing with event handling. The first one has to do with moving the handle, and there we saw that the handle has a new position like we expect, and similarly for the resizing the window, where the height of the window is the, is the new size. There was, there was also OCL annotating the associations. [COUGH] As we move from analysis to design, we will replace these associations with dependencies and as part of this process, we must assign each association's OCL to appropriate layer. Here are the three constraints that specify the associations in the analysis model. There was one for scaling the handle, there was one for displaying the document, and there was one for making the lines visible. This is just a repeat of what we, what we saw before, and you'll notice that the, in all three cases we have a single variable on the left hand side. This doesn't necessarily always have to be the case. You could well imagine a constraint, in which we said a plus b equals c plus d. That doesn't have a single variable on the left-hand side and we'd have to subtract b from both sides, or a from both sides or whatever to get that, okay. It could also be the case that there are inequalities. That is the constraint might say that a must be greater than b, okay. That one is going to be we'll have to think a little bit about how we can implement that. In fact, if a is greater than b that's even easier than saying that a has to be equal to be because any value of a that's greater than b will, will satisfy the result.

23 – Resize Window Quiz

Let's take these, these three constraints one at a time, and decide where it would be appropriate to assign responsibility for managing that. Before when we had associations we associated the, we, we had the constraints connected with the associations, but we don't have associations anymore. We have, just have components. So lets take them one at a time. The resize window indirect effect, that particular association, which component do you think would might be the appropriate one to be responsible for managing those indirect effects?

24 – Resize Window Quiz Solution

Sure. We could do this intuitively or we could be a little bit more, systematic about it. And one way of getting a hint on things is to actually look at the constraints. And you'll notice that for the constraints that we had the answers that Jerrod gave were also already on the left hand side of the equations. In fact, they were the in the context part of the constraint, they were the class that that came first. So that's a hit you can use to try to decide who's responsible for maintaining them. Now, in actual practice, any component could be responsible, okay? Or we could introduce a new component to be responsible. In fact that's what were going with this how do we manage the indirect constraints that are more than one class? Will that means some kind of interaction among the classes or components, and who's responsible for managing that. That turns out to be a tricky part of this. And we want to handle that in a systematic way as well.

25 – Constraint Placement

So, we've now updated the diagram, which we placed the constraints and associated the, the indirect constraints, and associated the, them with the, the components. Now we can begin to think about, well, implementing those components is going to involve being responsible for making sure that those constraints are, in fact, satisfied.

26 – Invariant Maintenance Quiz

For the two direct effects, which we, we use the term event handlers for, it makes sense to have methods responsible for doing that. And, with methods, we have pre and post conditions. And, it's pretty straight forward, which components are receiving the events, or the ones that responsible for, for dealing with them. For the three association, constraints that we've now assigned to components. Those were invariants. And remember that, OCL has pre-conditions, post-conditions, and invariants, and, as the system, as the user interacts with the system and makes a change, let's say, to the scroll bar handle, it's a responsibility of the system. To make sure that all the inter, indirect affect take place. That process is called invariant maintenance. An invariant is something which is always true. We've temporarily made it untrue. So we have to reset the system to a consistent state, we move the handle, we have to change what the contents on the screen. So that's and example of the, maintaining that particular invariant. Taking that a step further, when the user scrolls, remember we had three invariants. Which of those invariants do you think might break?

27 – Invariant Maintenance Quiz

I believe it's displays document. Is that the, that's the invariant that when it's held to be true showing us the correct sequence in lines so that's going to be broken as soon as we move the bar. So we move the bar. Remember the direct effect was that the bar moves. But if our system has any meaning at all that does what we expect it to do, it better be the case we see different lines. Okay, and we've already decided that the components have responsibility for dealing into that and in particular here the view port component has responsibility for dealing with that. Although it's going to have to interact with the other components. Right. Okay. So that's the job the implementation has to do. When the user scrolls, the event causes a move handle method to be invoked. Move handle causes the value of the handles positions attribute to change. That's a direct effect. Because the displace document constraint refers to handle position, the value of the view contents will change as we expected. Hence the invariant will have to be re-established.

28 – Invariant Maintenance Strategies

Key question between, going between analysis and design is how we are going to maintain these invariants. What maintains means is how, once the invariance is broken, you will propagate the knowledge of the break to the appropriate components so they can take steps to re-establish the invariant. We can call these invariant maintenance strategies, or we can just say. How are you going to implement this? And in particular, we're going to have examples of three invariant maintenance strategies. The first one is aggregated responsibility. That is, a single component is going to be responsible for managing this process even if it has to invoke or make use of several other components. That single component's responsible for handling the external events after delegating the inherent maintenance to the appropriate subordinate components. That's Strategy One. Strategy Two is the opposite distributed responsibility. Each component knows about the dependent components and anything that it's responsible for managing. And invokes them when this state changes. And third invariant maintenance strategy which is called the mediated responsibility or mediated strategy, okay? It involves a special implementation element called a mediator, and it's provided, it's one mediator for each invariant. The mediator knows all. Okay, it knows about the both the independent and dependent participants and the invariant. The independent one is the one that gets informed when the initial event takes place. And the dependent ones are the ones that have to respond to it. Three strategies, aggregated in one place, distributed, or a new component, a mediator, responsible for dealing with it.

29 – Centralized Strategy Quiz

Notice that these three strategies differ as to where knowledge of the participants is held. In general, there's a spectrum between centralized solutions where knowledge of all invariants is in one place, and completely decentralized strategies, okay? Which of these three strategies is most centralized?

30 – Centralized Strategy Quiz Solution

It's gotta be the aggregated strategy. Everything's- Well certainly the knowledge is in one place there, but with the mediated, the knowledge of a particular constraint is in one place as well. Right, but they don't have to have knowledge about each other, so it's not central in that sense. Right, so the set of invariance is distributed into separate mediators, okay? But for any given invariant, it's centralized in one mediator. In one mediator. Okay.

31 – Decentralized Strategy Quiz

And then the flip side is, which of these three strategies is most decentralized?

32 – Decentralized Strategy Quiz Solution

Distributed. Everybody- Sure. shares the responsibility.

33 – Tradeoff Between Locality and Complexity

In deciding among the possibilities the tradeoff between locality and complexity needs to be considered. In the centralized choice, there's a single place that handles all invariants, but that implementation of that place can be quite complex. On the other hand, complete distribution. While allowing each invariant to be handled in a simple way, can lead to solutions that are hard to debug. Because of the many moving parts involved.

34 – Example Continued

If the user moves the scrollbar handle, the invariant is temporarily broken, because the displayed lines no longer represent those that exist at the requested position in the file. Now for this particular example, let's look at each of the three strategies and see how it works. First off, aggregation.

35 – Aggregation

One of the components that the owning component, that is, the one that is aggregating things, and let's say in this case it's the ViewPort, has pointers of instance variables to the other two. It owns them. When the scroll, ScrollBar change request that is the direct event first is in, notified, or announce to the Viewport. It delegates a responsibility to the other components to handle it. First off, it has to find out from the ScrolBar what the new position is. The Viewport then determines that it needs additional content from the FileManager in order to reestablish the invariant. It makes a request to the FileManager for the required lines, and then uses its own method to display them. That is, the viewing window has aggregated the responsibility for maintaining this invariant.

36 – Aggregated Responsibility Quiz

Let's go with the second event, the resize window. When you play through the steps of how that's handled, assume that the event comes in to the, the window, it's the aggregator. What does it have to do to in order to re-establish the invariant

37 – Aggregated Responsibility Quiz Solution

So it needs to get the new reported height. Okay. From the event that has occurred, and it doesn't necessarily need to aggregate anything down to the file manager. It just has to change the size of the view port itself. Well, if the viewport size changes, there might be more lines. I see. So it has to be responsible for getting those lines if necessary. That's true. And I guess I overlooked that it would also need. Okay, so first things first is it needs to get the new location of, or based on the height it's going to get the new position of the scroll bar. And that scroll bar will indicate if it needs, well, the scroll bar doesn't move. Okay, so this raises an ambiguity, and it's good to raise ambiguities. So the act, if the size of the window changes, the height of the tray for the scroll bar changes. So, at least in absolute terms, the position of the handle may change. But its relative position may or may not change. Okay, so go ahead. Then based on that It needs to request from the file manager any additional lines of support [CROSSTALK]. Okay. And what was our other indirect effect here? If we now have, let's say, a smaller or larger window? The proportion of the skull bar walls have changed. The handled proportions can change. Can change. Okay. So, the if we're aggregating all of this, the viewport is then responsible for informing all it's other components of what it needs. And asking them to make whatever changes they have to make.

38 – Distributed Responsibility

The second possibility, the second strategy was Distributed Responsibility. The ScrollBar receives the change requests and determines the new value, that, that is the relative, relative position in the ScrollBar tray. It also knows that the ViewPort depends on this information, so it makes a method call passing the relative position to the ViewPort. The ViewPort compares the relative position received to the current value associated with the top displayed line, and realizes that it cannot satisfy the responsibility. It formulates a request to the FileManager for

the additional lines. The FileManager component returns the lines to the ViewPort for display, thereby reestablishing the invariant. That is, Knowledge of the invariant is distributed among three components that delegate partial responsibility to each other when needed.

39 – Distributed Responsibility Quiz

Give a shot with resizeWindow now on how that might be handled in a distributed fashion.

40 – Distributed Responsibility Quiz Solution

Okay, so, it seems very much. This is going to be broken up into pieces. So, our first distributed responsibility here is that the height is now changed, and it gets reported to the viewport. Okay. The viewport will look at its current height and make a determination of whether it's grown, shrank. Essentially, then having to distribute to the scroll bar, the handle that needs to move, and also if its proportion needs to change. Okay. That in turn, will be distributed down to the file manager from the scroll bar. It could be that the first distributor responsibility goes to the scroll bar, waits on something to come back. So, actually I think that's the case. Okay, so Jared has raised an interesting point here. We know we're eventually going to have to go to the file manager to get some more lines. Is it the scroll bar that informs the file manager? Is it the viewport that informs the file manage? Because both of them are dependent, right? Right. I think it's the viewport. Because our last example kind of did a similar thing when we just moved the handle. It eventually came back to the viewport. The viewport reported down to the file manager, I need these new lines. It seems like it's going to. Let's recap here. We're going to resize the window. The height is going to come to the viewport. It going to then report, or send a message to the handle about where it needs to be and its proportion. That will come back to the viewport, and the viewport will request from the file manager any nuance that it needs. Okay, and one of the takeaways from this is that distributed is just that. There's different choices as to who's going to tell whom else about what's going on, okay? So although each particular event is, each particular segment of code is probably fairly small, consisting of some method calls, that are distributed all over the place, which might make them hard to understand.

41 – Mediators

Third strategy is mediators. In mediator situation, a new implementation element is introduced for each invariant. And it's called a Mediator. Each Mediator is responsible for maintaining one invariant. And knows what the dependent components are. The independent event receiving component knows only that they must inform the mediator, when their attributes change value. For example, when the Scrollbar is adjusted, it alerts the relevant Mediator. Which in turn, requests the new position from the Scrollbar. The Mediator realizes the new content is required from the FileManager. Requests it, and pass it to the ViewPort. That is each Mediator has knowledge and responsibility for the maintain, maintenance of one invariant. And by the way, Mediator is an example of a design pattern. Which we will explore later in the course.

42 – Mediated Responsibility Quiz

Let's take resize window again and say let's say we had a mediator for dealing with that. How would that mediator do his thing.

43 – Mediated Responsibility Quiz Solution

So the mediator, is essentially like watching over the, the height change of the report and when that happens it will be reported to the mediator. And the mediator will make note of it. And then it will, I guess, request from the handle, its new position and of proportion based on that resizing. And then, it will then go to the file manager because this has changed and request for the new lines to be passed on to the b port. Mediator is, is a very object oriented solution. That is, there's an object responsible for that invariant, okay? And you could imagine at, at run time introducing new invariants and turning them on and off, and so on. Because essentially associated with, with an object. Okay? In our case the text browser we had three invariants to maintain. We would have three mediators. Each one of them would express the knowledge or implement the knowledge having to do with updating that particular invariant.

44 – Summary of Process

So to summarize this overall architectural design process, phase zero was specifying properties, and this involved constructing the context diagram. Indicating the external actors but only one activity the system itself. We indicated external stimuli or events. That can affect the system. We have the external actors being the user and the file system. We indicated how the system communicates its results back to the external actor. Those are the percepts. And then we specified in English, the behaviors that you want the system to have, and we used as starting point for that, used cases or scenarios that we developed when we looked at the exercises at the start of the term. In Phase 1 we componentized. Which meant decomposing the system into components, reallocating responsibilities to them, for handling events and for delivering percepts, and we assigned responsibility for the vary, guaranteeing the various properties. In Phase 2 we chose an architectural style, and that in turn specified how the components will interact. We chose a layered implicit invocation architecture, we have assigned the components to layers, we determined the dependencies between the layers, we updated our guarantees, we selected an invariant maintenance strategy, and we've assigned, in doing so, that assigned responsibility for maintaining those invariants.

45 – Conclusion

This lecture has presented an architectural design process using the text browser as a case study. The main result is a breakdown of system functionality into components. Also, the components are assigned responsibility for maintaining important system invariants. However, we haven't yet dealt with with the non-functional requirements, which is a major concern of software, software architectural design. We will address this issue in a later lesson.

P3L05 Non-Functional Reqs & Arch Styles

01 – Non–Functional Reqs Arch Styles

Software architecture prescribes the high level structure of a system in terms of it's components and how they interact. A key determinate of course of the architectural is the functionality the system must provide. Also important is that the system must satisfy a set of possibly confic, conflicting nonfunctional requirements. This lesson will provide guidance on the latter aspect. The material in the lesson is drawn from the book by Bosch listed on the class resources page.

02 – Qualities

Beyond the functionality that a system provides exists certain non-functional qualities. These qualities are often crosscutting. What this means is that their implementation is often spread across the entire system. For example, consider re-usability. It is not enough that one of a systems components is reusable, it means that most of the components and possible the application as a whole must be easily applicable in other contexts. Because of the crosscutting nature of non-functional qualities, providing them often has a strong effect on the system's overall structure.

03 – Non Functional Qualities Quiz

For our first quiz, because there are many different non-functional qualities that a system might have, see if you can name three of them and put them into the text box.

04 – Non Functional Qualities Solution

Here are a few that you might have mentioned. The ones that are underlined, we're going to consider further in this lesson. In particular, performance, security, safety, reliability, and maintainability.

05 – Functional and Non Functional Requirements Quiz

For a second quiz, imagine a program that plays tic-tac-toe against a human opponent. I've listed a set of four requirements, put an F next to each of the requirements that is a functional requirement and an N next to each that is a non-functional requirement. First, the system should check for illegal moves by the human. Second, the system should respond to human moves within 5 seconds. Third, the system must be written in Java, and fourth, the system should allow the human to use either a X or O as his or her marker.

06 – Functional and Non Functional Requirements Solution

For the first requirement, concerning illegal moves, that's part of the functionality of the system. So your answer should be F. However, the second requirement, that the system should respond within five seconds, is non-functional. It's not what the system is computing, but it's how or with what quality the system is doing the computing. Third, the system should be written in Java. Once again, this is a non-functional requirement and the fourth one, the system should allow the user to use either an X or an O, that's functional. That's part of what the system actually computes.

07 – Quality Catalog

So let's take a minute and exam each of these five non-functional requirements that we'll be using in this lesson. After we do this we'll see how each of them influences systems built with different architectural styles. For each quality we will provide a definition. The ways in which the quality might be measured. And some devices that are used to provide this quality. First off for performance, the Software Engineering Institute's definition of performance is, that attribute of a computer system that characterizes the timeliness of the services delivered by the system. And we can measure this in a lot of different ways, in terms of response time, throughput, system capacity, system utilization. And there are many devices that programmers can use to improve performance for example caching, concurrency, memory management, and so on. The second quality we want to look at is maintainability, and this is the extent to which enhancements can be readily added to a system. This is sometimes also called evolvability, flexibility, adaptability, and so on. The measures and we have talked about these earlier are coupling and cohesion, and there's many devices such as encapsulation, publishing your interfaces, use of sub-classing, indirection, and wrapping. The third quality is reliability and this is the likelihood of failure, of system failure, in a given period of time. That is, the continuity of service that the system provides. The typical measure for this is called Mean Time To Failure, or MTTF. Some of the devices that can be used to provide a high reliability include redundancy, fault tolerance, and recovery blocks. Software safety is the extent to which a system protects against injury, loss of life, or property damage absence of catastrophic consequences. It can be measured in terms of the complexity of the system, the time coupling, and by fault tree analysis. Some of the devices include hardware interlocks, in the case that the particular system has peripheral hardware devices, and false containment strategies. The fifth quality that we'll, that we will look at, it is security. And this is the extent to which the system protects against unauthorized intrusion or provides confidentiality. Security is measured by levels such as confidential and top secret. And sometimes formal proofs are used to guarantee that the system obeys whatever its confidence level is. Some of the devices include authentication, authorization, security kernels, encryption, auditing and logging, and access control mechanisms

08 – Applications Quiz

Those are the five qualities we're going to look at as a quiz for you. The first column contains a list of the five qualities and the second column contains some sample applications. See if you can match these up, that is, what application in column two is a best match or best represents the quality listed in column one.

09 – Applications Quiz

While weather prediction is a good tester performance. The finer the grid on which the weather is computed, the higher the quality of prediction made, and so having many, many computations on a fine grid, which might stress the performance of a system is the best way to get good results. As far as security is concerned, online banking. Certainly, you don't want anybody interfering with your bank accounts. And so, having a highly secured banking system is important. As far as safety conce, is concerned, I want the cruise control software on my card to be highly, highly safe. Maintainability, I'm thinking here of the Twitter API. That is we know that there are many, many applications that are based upon the twitter API, and over time if we can maintain stability and maintainability of that API, those applications won't be broken. As far as reliability is concerned, I want my traffic light controllers. So, at every intersection you come to where there's traffic lights those traffic lights are controlled by some kind of control box, and we want that control box to be, the software on it to be as reliable as possible.

10 – Architectural Styles

Now that we've had a look at the five qualities that we're going to be digging into, let's also look at the architectural styles we're going to compare them with, okay? In particular, we're going to examine be examining the effect of the five selected non-functional qualities on system architecture for each of these five styles. The five we will look at are pipe and filter, layered architecture, blackboard, object-oriented software architectural style, and implicit invocation. So first, let's take a minute to recall the features of those particular architectural styles.

11 – Review of Architectural Styles

The definition of pipe and filter from Wikipedia is a chain of processing elements called filters arranged so that the output of each element is communicated by a pipe to become the input to the next. Layered architectures according to MSDN, the Microsoft Developers Network, is the grouping of functionality into distinct layers that are stacked vertically on top of each other. Communication between layers is explicit and loosely coupled. Blackboard architecture, out of the artificial intelligence world, according to Wikipedia, is a common knowledge base. Is iter, is iteratively updated by a diverse group of specialist knowledge sources, starting with a problem specification and ending with a solution. The object oriented architectural style according to MSDN is the division of responsibilities into individual reusable and self-sufficient objects each containing the data and the behavior relevant to the object. Note that object oriented architectural style is somewhat different than what we talk about an object oriented program or object oriented process for developing programmers. Finally, implicit invocation, according to Garland and Shaw, is a component can broadcast events. Other components in the system can register interest in those events, by associating a procedure that should be called when the event is detected. When the invent is announced by the system, the system itself invokes all of the procedures that have been registered for the event.

12 – Pipe and Filter Performance

Let's begin by looking at the performance issues when using the pipe and filter architecture. On the one hand, the pipe and filter style can enhance throughput because the filters can run in parallel, that is, concurrently. So you're overall system throughput can be reduced. On the other hand, an individual filter may be slowed down if it, if it is waiting for its supplier.

Moreover, if the hardware only allows one process to run at a time. There maybe significant overhead due to contact switches among the filters.

13 – Pipe and Filter Maintainability

So that was a quick examination of the affect of one quality, performance, on one architectural style. Let's look at the affect of another quality, maintainability, on pipe and filters. On the positive side each of the filters in a pipeline is an independent unit and this enhance, enhances encapsulation and reusability. On the other hand some changes, like the format of the data that's going through the pipe line may affect, all the, all of the filters, thereby increasing their coupling.

14 – Pipe and Filter Other Qualities

We'll now take a quick look at the other three qualities and their effect on pipe and filter. For reliability, okay, the reliability of a pipe and filter system may be reduced because, the reliability of an overall system is only as good as its weakest, weakest link. That is, if any of the filters in the pipeline or if any of the pipes break down for some reason, the whole application breaks down. Safety may also be reduced because of the multiple dependencies. On the other hand, it's easier to verify because all of the output comes from a single source. Security benefits because of the simplicity of the architecture increases opportunities for authentication, encryption and implementation of security levels.

15 – Layering Qualities

Let's now take a look at layering. By and large, security is enhanced because it is straightforward to add a security layer between the system and its environment. As far as the effect of the other four qualities on layering, performance may be reduced because the response to external events must be passed up and down the layers, which may in, may also increase context swapping. Maintainability, on the other hand, might be improved because of the stable interlayer protocols and interfaces would lead to well-defined and reusable components. It may even be possible to replace an entire layer or insert other layers. Reliability may be reduced because an event may be handled in multiple layers. That is, making it hard to find, when something goes wrong, what the responsible layer is. However the higher layers may have an oversight capability to provide the necessary redundancy to improve reliability. As far as safety is concerned, similar to security, it maybe easy to insert safety monitoring layers.

16 – Blackboard Reliability and Security

Let's take a minute to look at reliability in blackboard architectures. In a blackboard architecture, the independence of the components can increase system resilience. That is, the system may continue to function in a degraded fashion if one of its components breaks. On the other hand, because there is no overall definition of system behavior, it may be difficult to identify the cause of a problem when something goes wrong. As far as an advantage is concerned, access control and the blackboard architecture's enhanced because there's a common data repository. On the other hand, the flexibility of a blackboard architecture allows for the dynamic condition of new components which may reduce confidence in overall system security.

17 – Other Blackboard Qualities

With respect to the three other blackboard qualities, performance, because there's a lack of well-defined control flow, may lead to redundant and administrative behavior. For example, the polling of a repository. Maintainability is enhanced because having independent com, components can lead to in, increased flexibility. But if we make changes to the common control paradigm wherein the blackboard components are updating the repository, or if we change the repository's data format, this can have pervasive negative effects on maintainability. As far as safety is concerned, because the blackboard is a common repository accessible to all the components, if you somehow get bad data un, that might lead to a safety problem, this can easily spread to the other components.

18 – Object Orientation Maintainability

Let's now have a look at the object oriented architecture style and its relationship to maintainability. The object oriented architecture style is a very powerful way of organizing a system. And maintainability is significantly increased when using this style, because of the independence of the components, both their encapsulated data and the hands-off, message passing style of interaction. On the other hand, objects will have to refer to each other. Okay? There needs to be some way of identifying other objects in the system. This can increase the intercomponent dependencies, thereby reducing maintainability.

19 – Object Orientation Security

I'd like to examine the security issues that arise in object-oriented architectural styles. On the positive side, the encapsulation inherented in obj, inherent in object-oriented systems, can reduce vulnerability. Negatively, the many relatively small and independent objects increase system fragmentation, thereby meaning many more possible points of infection. Moreover, the relatively unconstrained message passing paradigm can ease the spreading of a problem throughout the system.

20 – Other Object Orientation Qualities

As far as the three other system qualities in an object oriented architectural style, performance has problems because of the many small objects linked to multiple context switches. And delegation, okay, whereby one object may refer to others to provide its functionality, this can increase indirection, which can reduce responsiveness of the system. With respect to reliability, decentralized control in object oriented system reduces opportunity for oversight. But encapsulation can reduce vulnerability to undin, unintended interactions. For safety, the correspondence between the real-world entities which the system is is modeling, and the objects that the programmer has developed can improve the intentionality of the system and accountability, thereby enhancing the safety of the system.

21 – Implicit Invocation Qualities

Our fifth architectural style is implicit invocation. And here let's first examine the question of reliability. Well, if you're approached to event delivery, in an implicit invocation style it's centralized you are more easily able to deal with unexpected events, thereby improving reliability. On the other hand, because interactions are implicit, overall system understandability is reduced, potentially compromising reliability. Or the other four system qualities, performance, might be compromised because of the extra communication due to the bookkeeping and indirection can lead to context swapping problems. Maintainability, there may be increased reuse

due to the independents of the components. On the other hand, as far as safety is concerned, increased interaction complexity may make it harder to ensure safety. And with to respect security as we saw with object orientated, object orientation, the fragmentation of an implicit location architectural style can cause problems, but encapsulation can help to mitigate them.

22 – Side Effects Quiz 1

Here's a quiz on architectural style. Match each of the architectural styles with a negative effect it could have on system design. The four architectural styles are pipe and filter, blackboard, object orientation, and implicit invocation. The negative effects include increased system fragmentation, reduced system understanding, increased coupling, and can promote the spread of bad data.

23 – Side Effects Quiz 1 Solution

Here are the answers for this quiz. First off, the answer for Pipe and Filter is C, increased coupling. This is because some changes, like the format of the data going through the pipeline could affect all of the filters, reflecting their tight coupling. The answer for Blackboard is D, can promote spread of bad data. Because the blackboard is a common repository accessible by all components, if it somehow gets contaminated this bad data could readily spread to other components. The answer for object orientation is A, increased system fragmentation. Object orientation can increase system fragmentation because of the many relatively small independent objects that may exist. And finally the answer for implicit invocation is B, reduced system understanding. This happens because server components are not aware of clients which can change dynamically.

24 – Side Effects Quiz 2

Here's a related quiz on non-functional requirements. Please match each of the following non-functional requirements with the side effect using it might engender. The four non-functional requirements are reusability, reliability, security, and performance. The possible side effects are increased system fragmentation, reduced system understanding, increased coupling, compromised delivery schedule.

25 – Side Effects Quiz 2 Solution

The answer for reusability is A, increased system fragmentation. High reusability means high cohesion. That is, that each module has a single purpose. This might lead to more modules, and hence, to more connections among the modules. The answer for performance is B, reduced system understanding. Performance requirements often are dealt with via introducing special cases or Achaean data structures, which can make the code harder to understand. The answer for security is C, increased coupling. Security means data security, and data security is provided by controlling access to the data. This means that in order for modules in order to access the data, they need to go through some form of data access control. Which is typically provided by a centralized control module, to which all the other modules must be coupled. Finally, the answer for reliability is D, compromised delivery schedule. Increased reliability typically means extra code to check for potential problems. Extra code means extra coding, checking, documentation, and so on, which can lead to difficulties delivering on time.

26 – Summary

To summarize, non-functional qualities, not just the five that we've looked at but the whole, whole set of them, can dramatically affect the architectural software system. Moreover, real world systems often have multiple conflicting non-functional qualities. This means that you as a designer have to make tradeoffs among them. For each of the quality requirements of your system, be sure to take into account both the positive and negative impacts that it will have on the overall system architecture.

P3L06 Connectors

01 – Connectors

Recall our terminology for describing architecture in terms of components, connectors and configurations. The components are mainly determined by the system's functionality, with a few invented to handle non, some non-functional requirements. Now it's time to take a look at connectors. The main material for this lesson comes from the [UNKNOWN] paper, which I have asked you to read. There is also some material in the optional text by Taylor [UNKNOWN] on software architecture foundations. According to Shaw and Garlan, connectors are responsible for mediating the interactions among the components. They establish the rules that govern component interaction and specify any auxiliary mechanisms required.

02 – Atomic Elements

Let's come at this from the bottom up, in terms of the atomic elements out of which connectors are built. The base element is called, in the meta-paper, ducts, as in air-conditioning ducts. Okay? This, this is the channel, has no associated behavior, with it. It could, for example, be some kind of internet connection. Or it could be all on one machine in terms of the underlying electronics. It could be provided by the program language implementation, for example in a virtual machine. It could be the operating system through some kind of system call. Okay? Or it could be an inter machine communication, for example with sockets. Ducts provide the mechanisms for transmitting the data, and it could also be control information among the components. Connectors go beyond ducts, by providing the protocol used by the ducts for doing that communication. That is the sequence of interactions. In addition to ducts, connectors may include internal mechanisms. For example, some storage,uh, like, like would be used for buffers. Or computation, computational elements such as might be involved if you, if the, if the connector were fighting some kind of translation capability.

03 – Pipe and Filter Quiz

To get started, here's a little quiz. For our friend the pipe and filter architectural style. Which is the component and which is the connector?

04 – Pipe and Filter Quiz

I hope you said that Pipes are the connectors and Filters are the components.

05 – Service Categories

In addition to [UNKNOWN] mehta talks about, meta at all talks about services. Now these are not, services in the sense of services that the, user sees or even service in the sense of client server. Okay that words, it's a unit of computational service. These are services that connect or provides the overall architecture. According to Mehta, it's a service category that represents the broad interaction role the connector fulfills, and they lay out four different

categories of services including Communication. That is the transmittal of data. Coordination which is the trans, transfer of control. Conversion that's when some kind of a translation is going on, particularly among data formats. Okay? And Facilitation that is some kind of mediation or operation optimization activity. And they use the abbreviations, T,O,X and F respectively on, in their diagrams to abbreviate those particular four categories of services.

06 – Services Quiz

For this particular quiz I provided some types of services and asked you to say what categories those fill in. See if you can provide one of the four letters next to each of the service names to give its major category of service that it provides.

07 – Services Quiz

Well, for data buffering, that's really a facilitation. It makes the system work better, but it's not in itself a major communication of either data or control. Acknowledgement, guaranteed delivery, multiplexing, transactions, scheduling, synchronization, all about the collaboration aspects of, that a particular connector could provide. As far invocation is concerned, there is a collaboration aspect there, but there's also a data transmission aspect as far as the parameters are concerned. Dynamic reconfiguration is a, an advanced technique for making a system run better at runtime. So that's a facilitation operation, as is load balancing.

08 – Variety of Connectors

So we've talked about the atomic elements, the ducks. We talked about the different surface categories. Now we'd like to actually look at a variety of different kinds of connectors.

09 – Procedure Call Connectors

Let's drill down a bit into a very simple connector, the procedural call connector provided by all, all your friendly programming languages. First of all there's a flow of control, that is, control is with the calling routine and then control shifts to be with the called routine. So that's a coordination role. There's transmission of data via the parameters, so that's a communication role. Procedure call is so common, so pervasive a part of programming languages that it's basis for all of the composite connectors, that is when we make a complex connector out of simpler connectors. For example in the Java method call we have the caller, the callee, there's exactly one, at any given time there's exactly one caller and one callee. So we can say the fan in is one and the fan out is one as well. Sounds simple right, but there's all kinds of variations. For example,. How are the parameters transmitted? Well, there's call by reference, call by value, call by name. There could be default values, keyword parameters, inline parameters. If there's a return value, things could be provided by invocation records, by a hash table. And even the order of valuation is a variant. Are the arguments which might themselves be procedure calls, method calls evaluated left to right, right to left or whatever. Some languages provide for multiple entry points that is when you call a method you may enter that method in different places. Is the invocation explicit as it would be in a method call, or is it implicit as it might be in an object oriented line in which there's delegation, and the ultimate callee is known specifically by the caller. Usually we think of procedure calls as synchronous but there are also situations where procedure calls can be asynchronous. We mentioned the possibility of different fan ins and fan outs, okay? And then there's the issue, the variation that is allowed as far as accessibility is concerned. We know, for example, programming languages allow for only private access that is within the same class. Or protected access, to the method from

the particular class or it's parents, or children classes, and then public acc, access, where any other, the caller can be anyplace

10 – Event Connectors

The second kind of connector we'd like to look at are event connectors these are, these are also very common. In fact book on the resource page. They devote a very nice section to describing all the different variations that might exist with event connectors. First of all event connectors are responsible for a flow of controls so that's a coordination role. They may also pass parameters. Typically this might involve time stamps, or actual data, so that's a communication wall. Event connectors, once an event is detected, generate messages, method calls. After detecting the event or some combination of events that it's, it's prepared to detect. Event connectors are particularly relevant for distributed, asynchronous applications in which we need to know when certain things happen. The set of event connectors that exist is dynamic. That is, the application itself can turn on or turn off the ability to detect certain events. Some of the variations that might exist among implementations of event connectors include cardinality. That is how many different components can produce the event? How many different observers of the event might exist? And might there be patterns of events? In which, we like to be able to detect currents of the pattern. How were the events actually communicated? Is it via best effort, exactly once, at most once, at least once? Do we have a priority among a set of events? Do we always handle the outgoing ones before the incoming ones? Are there different priorities embedded with the event that are handled in a certain fashion? Synchronous, asynchronous, or based upon certain time out? How is notification handled? Is it polled? That is, does the potential receivers have to periodically look to see whether the event occurred? Is there a published, published subscribe interface in which a particular component registers it's interests in events. And then, gets told when events happen? Is there a central updating mechanism that, a registry that receives all of the events and distributes them to the known, known parties. Or is, are there queues sitting there that everybody's responsible for looking at. Causality refers to the circumstances determining the actual issuing of the events. Are there, absolute, absolute event occurrences? Or, could could the events be relative to other situations that is conditional type events. And what's is the ultimate, generator of the event. Might it come from hardware such as pa, page faults, interrupts, or traps. Or are they software signals or triggers or even, inputs from the, from the GUI.

11 – Data Access Connectors

The 3rd main category of connector are the data access connectors. And this is as the name indicates, this, is where the connector is responsible for dealing with access to some kind of data repository. Hence, there is a communication service provided. Moreover, the, the access connector may provide some kind of translation. Surfaces could be character set translation, or something at a higher level. Hence, there's a conversion service being provided. Some of the variations and data access connectors include locality that is. Are the connectors specific to a particular thread, or to a particular processor are they global? What kinds of access are allowed? Is it simply query retrieval, or might there be changes allowed? What's the availability of the data access? Is it transient? Is it persistent that is could it be long lasting as with the earlier connectors accessibility in terms of private, protected or public. With respect to, to life cycle who is responsible for doing the construction or building and who's responsible for cleaning up when things are. Over think your instructors and destruction, and as far as, cardinality is concerned who, who's responsible for defining the messages and who's responsible for receiving them or using them.

12 – Linkage Connectors

The fourth category of connector is a little bit different than the other. This is linkage connectors. And they're responsible for describing the structure of the system. That term linkage here, you can think in terms of link editors, if you've ever heard of those ways of organizing or constructing or putting together the pieces of a system. Linkage connectors are responsible for establishing the ducts and enforcing the interaction semantics. Hence, they provide a facilitation service. Because they're responsible for putting the system together but not for actually running the system, they may disappear after setup is complete. The unit of linkage might be a module, might be a file might be an object. And related to this are tools like configuration management tools and the make command for actually building a system. There are semantic issues with respect to the granularity of the pieces and the semantics that is the, what are the protocols among the pieces. Among the variants that are involved with linkage connectors are whether they're implicit or explicit. For example, implicit might be something like make where you merely state a overall target that you're trying to build and the other building steps are done, done for you. The granularity that is what, what unit is being put together. Could it be variables, procedures, functions, and, and so on? And then of course, the semantics, the cardinality in terms of defines and uses provides and requires, and a key one is binding, that is when does all this happen? It might be at compile time, it might be at run time, or might even be before compile time if, well, part of your construction process involves things like templating or generics.

13 – Stream Connectors

Streams are another popular form of connector. They're primarily concerned with data transfer, that is communication services. Common examples include pipes TCP sockets and proprietary client-server type protocols. Some of the possible variations available with stream connectors include delivery guarantees. Whether the stream itself is bounded, that is, it only has a certain capacity or whether it's unbounded and whether it's capable of buffering the information. What its units of transmission are, that is, is it bytes or is it something more higher order, more structured? Is it stateful or stateless? Is it named or unnamed? Is it available only locally or is it, is it more remote? Synchronous or asynchronous or timed? Law or structured? And what's the cardinality? That is, is it definitely a one to one type connection or might there be multiple receivers? Or might it even be end to end where there are multiple riders and multiple receivers?

14 – Arbitrator Connectors

A powerful category of connector are arbitrators. These are primarily responsible for facilitation services, but because they can redirect control there's also a coordination service they provide. You might be able to use them to negotiate service levels. That is how much resources are being devoted to a particular problem. Hence they support reliability and atomicity, scheduling and load balancing, trapping of faults and even synchronization. Some of the variations for arbitrator connectors include how they handle faults. Typically with a simple arbitrator scheme, there's a single decision made but more complex systems might involve a voting scheme. That is if there are three arbitrators around, they would have to vote on the course of action and the majority would rule. How concurrency is dealt with. The mechanism is it semi fours, a rendezvous, monitors, locks there's lots of approaches to this. And whether it's a light weight approach or a heavy weight approach. Variations involved with transactions such as whether they're simple or they're nested. Whether the arbitrator is there if you need

it or is required. And whether the arbitrator supports reads, writes, or both. And then a major category of, of capabilities and variation of arbitrators involves security. Authentication, authorization, screening, durability that is how long the particular decision last. Is it a single session, or is it a multi session? And then the scheduling of the arbitrator activities.

15 – Adaptor Connectors

Another category of connector is called the adaptor connectors. These are responsible for putting together components that were not designed to be put together. This often means there's some kind of translation going on. It might be converting protocols and policies and hence there's a conversion or transformation activity provided. Some of the variations include invocation conversion, that is is there a dress match, mapping? Is there marshalling, virtual memory translation, virtual function tables? Are there conversion as far as wrappers or packagers? Is there protocol conversion or even is the presentation conversion, that is is the output. The actual form of the output determined by some kind of, computation engine such as XSLT.

16 – Distributor Connectors

The final category of connectors to look at it are called Distributor Connectors. They're role is also primarily facilitation. They identify the interaction pads and they rout things among them. In a sense, they are assisting other connectors. A primary example here is DNS, Domain Name Services. Whereby various components on the internet can talk to each other using names rather than strictly by addresses. Some of the variations that are possible with distributor connectors include naming. That is are they structured based. Can they be hierarchical or flat? Or are they attribute based? What's the delivery policy? Is it best effort, exactly once and so on? And is the mechanism unicast? That is point to point or multicast or broadcast, and then routing, okay? Is there a bounded list? Or is it more ad hoc? And is the path static, cached, or dynamic?

17 – Summary of Connector Types

So there's a variety of different kinds of connector types, and these, these different types provide different categories of services. Many of them are familiar, but some of them may be a little bit, new to you. The point as with, earlier lessons is to be aware of what's out there in case you need it in designing your systems.

18 – Connector Type Quiz

A quiz for you to try with respect to these different types of connectors. I've listed different mechanisms, and I ask you to determine which kind of connector type each one of them belongs to. So remote procedure calls, schedulers, buffers, shared library configuration, or SQL.

19 – Connector Type Quiz

Well the simple one was remote procedure calls or examples of procedure calls. Schedulers, are really arbitrators, right? They're determining what's going to happen and when. Buffers are used with stream connectors. Shared library configurations is a linkage connector. And SQL is a data access connector.

20 – Composite Connector Examples

So far the connectors we have been talking about have been simple connectors. It's also possible to put connectors together; that is, to compose them, make it more complex connectors. One example I'd like to give is a science data server. Whether you're aware of it or not, there's lots of data being generated out there. Think of all the land sat photographs being taken. Using different frequencies of light they can record all kinds of information that's stored away on data servers. Then there are clients for these servers. The clients may be synchronous or asynchronous. That is, they may have specific requests or they might want to get a stream of data themselves. Okay, and we need to be able to build a composite connector to provide this overall capability. It may involve event connectors, data acc, certainly will involve data access connectors might be streaming of data, and it might be distribution. There are different policies that we might want to enforce for delivery. We might, we almost certainly will have multiple producers and consumers. There's going to be almost certainly some data transformation going on and the access may be public or it may be, may be private, okay? It may be transient, or it may be persistent, persistent depending upon the policy of the data server. It might be packaged into streams or it might be packetized, okay? And there might be a naming registry involved so that they information can be accessed via a specific query. Another example of a composite connector are various FTP applications, such as Globus, bbFTP, and GridFTP. These combine procedure call, data access, steam, and distributor simple connectors. Therefore moving and distributing large amounts of grid data. Could be hierarchically or flatly named. Typically synchronous. Using web protocols such as SOAP might involve time-outs, authentication would prevent unwanted access to the data. There's parameter passing. The data might be transient or it might be persistent might be public or it might be private. And it might be at the level of byte stream or the underlying bytes might be raw or structured. Probably one exactly-once deliver, a bounded buffering, and unicast, that is, point to point transmission. The third example of a composite connector is client-server based distributed distribution connectors that involve things like REST architectural style, HTTP protocol. Remote messaging vocation. CORBA, FTP, SOAP, this particular kind of connector, kind of composite connector might use procedure call connectors, data access, stream and distributor connectors. Involve revoke procedure call, would name parameters, persistent and transient data, and naming registry, and typically unicast type connections. The final example of composite connector is peer to peer based data distribution connector such as BitTorrent. Here there is a combination of arbitrator, data access, stream, and distributor connectors. Typically peer-to-peer is, major facility is controlling the flow of, of things. And that is, control flow redirection. If one of the peers is not available, you go to another peer to provide the information. There may be negotiation of protocols, scheduling and timing. There may be voting involved, depending upon circumstances might use either rendezvous or transactions. The data may be transient or persistent. typically, streams are what's what, what's used. They may be buffering involved. And for these types of applications, typically it's at least one semantics. You don't want to deliver more than once

21 – Connector Design

I'd like to take a moment and talk about the design of connectors. As with components there is a design step required, particularly if you're building your own connector. Starting from the overall architecture, of course you determine your components and the required interactions among them. Then for each interaction, determine what required services that interaction needs. Once you've done that you can select a con, connector type that provides that service. Each of the connector types has a variant of dimensions, variation dimensions, that you can

choose which of those variants you would like to have. And from that, define your connector. And then you need to validate. And we'll have one in a minute we'll, we'll describe some of the rules you can use for checking whether or not the, the choices you made will work out. And in doing this you may actually have to define your own connector. You may not find one in the catalog or in available libraries that you can use.

22 – Validation Rules

One category of validation rules are requirement requirements placed by the value on one dimension on the values of another dimension. For example, if you've got event connectors that require delivery notification, then you also need to be concerned with the cardinality rules, synchronization rules, and mode rules. Might be situations where it's not a strict rule, but it's some kind of caution. Certain combinations may be unstable or unreliable. That is they're dynamic, the dynamics of that particular situation may not work in all circumstances having to do with concurrency and locality. There may be restrictions. 'Kay, certain combinations may be invalid, for example, passing by name and transient can't be used together. And there might be prohibitions. Total incompatibility of dimensions, such as streams and atomicity

23 – Linux Case Study

The meta, paper concludes by having a case study using the Linux operating system. Case study was concerned with higher-order connectors. In particular, a operating system like Linux provides several major elements in support of all the applications they're going to run on the system. One key component is the file system. Another is shared memory. And the third is process, support, support for processes. As far as the files are concerned, the underlying hardware doesn't have files. Right? The. It has bytes. There may be a mechanism by, for, for blocking those bytes into groups. But the operating system self provides some kind of facade, that makes it look like there are files there. And in so doing, it ne, needs to deal with contention. If there are multiple, accesses to the file, are you allowing them to take place concurrently or is there some kind of synchronization. required. And it's the operating system's job to provide these arbitration, adaptation, and coordination connectors to get a composite file facade connector. Shared memory, is a data-access type connector but there are issues there, synchronization as well. And finally, in terms of process scheduling, okay, processes are all about controlled access to resources so an arbitrator type connector is required there.

24 – Summary

It's easy to think that after you have determined the components of a system that your job is done. Just as vital is determining how those components will correctly interact. Treating connectors as first-class part of the design process can further that end.

P3L07 Acme

01 – ADLs

The topic for today is Acme, which is an architectural description language. As awareness of the importance of architecture has grown, so too have the tools available for working with them. Examples include tools for drawing, simulation, analyzing, reverse engineering and reporting on software architectures. A necessary step for tool interaction is a standard way for representing software architectures, that is for an architectural description language or ADL. In this less we will look at one particular ADL called Acme.

02 – ACME

Acme is a extensible ADL developed at Carnegie Melon University and University of Southern of California's ISI. It was specifically designed to facilitate the interchange of architectural descriptions among tools. And it comes with a variety of tools of its own. Including Acme Studio, which is a graphical editor by which you can draw architectural diagrams. AcmeLib which is an API in languages such as Java and C++ for interacting with these artifacts. And AcmeWeb which is a document generator. More information about Acme can be obtained by following the link on the class' resource page.

03 – ACME Features

The primary contribution of ACME is that it defines a vocabulary for talking about architectures. Other features include an extension mechanism enabling tool specific sub languages to be imbedded in it. For example, a simulator might wish to deal with timing information. ACME features generics, families and types for defining architectural styles. Further, ACME is defined in such a way that ACME descriptions can be converted into first order logic suitable for use by automatic reasoning tools such as type checkers.

04 – Architecture Vocabulary

Here is Acme's vocabulary. Some of it should be similar to you from the Garland and Shaw paper that you read, and terms that we have been using to describe architecture so far. First off is components, these are computational elements and data stores. Along with them come connectors, which are communication and coordination among. The components. Ports are com, component interfaces, possibly including protocols. And roles are similarly connector interfaces. So you can imagine plugging together components and connectors using their their po, ports and roles. The Acme term for configuration is a system. It is the set of components and connectors you have for a particular architecture you're describing, the actual configuration specified via what Acme calls attachments of the ports to the roles. I'll also be talking about representations, which is Acme's way of describing hierarchical decomposition across multiple levels of abstraction. And rep-maps which are specific bindings between those

levels of abstractions. Those seven terms constitute all that Acme requires of you to understand in terms of building architectural descriptions.

05 – Simple Architecture Example

Here's a simple ACME description for a client server system. At the outermost level is the keyword system indicating that what's being provided here is a configuration describing this particular architecture. The system has a name, in this case simple CS, and then contained within it, are some other statements describing the constituents of that particular system. In this case, there are two components. One is called client and one is called server, and there's one connector. Also there are two attachments for plugging them together. The first component client has one port out of which requests are sent. The second component server has one port for receiving a request. The connector, which is called RPC, has two roles. One of which is called the caller role and one of which is called the callee role. Then, in a natural fashion to attach these together, the client's send request port is connected to the connectors caller role. And similarly the servers receive request port is connected to the connectors callee role, hence we now plug together. The client, the server, using the connector in such a way that the ports and roles are plugged into each other. The system is thus configured in such a way that the client can send requests using its port through the connectors so the server receives those requests and can then act upon them.

06 – ACME Quiz

To see whether you've understood this idea, here's a little quiz for you. Take the ACME code you've just seen, and add a means for communicating errors from the server back to the client.

07 – ACME Quiz

Well we still have the same two components. We haven't added anything there, but for each of the components we've added another port for dealing with these error messages. In particular for the client, it's now got a port which is called here err-trap, and symmetrically the server has another port, this one for sending the alert. We're going to add a new connector for dealing with this communication channel. It's called error, okay, and it's got two roles. One being the source of the error and the other being the sync or receiver of that error. We still have the same two attachments we had for, for dealing with the main flow of communication, but now we've added two more for dealing with error flow. In particular, a clients error trap is connected to the connector sync and the receiver's alert port is connected to the connector's source role.

08 – ACME Graphical View

Although what we just looked at is, looks like a programming language, syntax, text and so on, if you use ACME, you'll likely to use it through its graphical editor, in which case, you can draw all of what's going on. Here is a screenshot from, the ACME studio tool. It, it contains two rectangles which correspond to the components and one circle which corresponds to the connector, and you can see that the two components are both plugged into the connector and hence, can communicate with each other. If you build your description using the graphical editor, you can then generate the code we've just seen in the textual form.

09 – Decomposition

What we've seen so far, is capable of dealing with simple, one-level architectures. However, humans deal with complexity using divide and conquer, breaking things into smaller pieces and then trying to put them together somehow. For software architecture descriptions there are 2 kinds of such Decompositions, Horizontal and Vertical. Horizontal de, Decompositions are done at the same level of abstraction. We understand the human body in terms of its digestive, respiratory, immune and so on systems. However, we can also decompose vertically by going deeper into the abstraction hierarchy, that is we understand the respiratory system in terms of Lungs, Trachea, Diaphragm. The Lungs in terms of Alveoli and gas transfer for example. We have already seen how, Acme deals with Horizontal Decomposition in terms of components like connectors and so on. now let's look at it's support for Vertical Decomposition.

10 – Representations

Acme supports vertical decomposition by allowing any component or connector to be represented by one or more lower level views. Note that there are two things going on here. One is, levels of abstraction, that is a view can be at a lower level representation of something of a higher level. Also, is the fact that you can have multiple views of the same higher level thing. Recall from our discussion of architectural views, that no single view is likely to provide all the information we need. And hence having multiple views, allows us to develop different representations, that can each add something to our understanding. Each view in your Acme description is called a representation. Within representation there's a mapping between levels called a rep-map, short for representation map

11 – Example Representation

Here is a simple example of a representation. It is a decomposition of a single component, which is called here, the component. The component involves a, two ports: one for dealing with easy requests, and one for dealing with hard requests. And then, there's the representation, which describes the details. In a sub-system, called details that has two components. One fast but dumb component and second a slow but smart component. The binding section then pastes these two levels together. Easy requests are mapped to the fast but dumb component port P and hardRequests to the slowButSmartComponent also called Port P. Hence we now have the same system described at two different levels of abstraction, the lower one allowing us to go into the more details than the upper one.

12 – Extending ACME

What we have seen so far are the basic features of Acme. That is the vocabulary of keywords and descriptions that you can build from the keywords describing basic architectures. However, Acme has some additional features that allow you to go beyond this basic vocabulary. In particular, because Acme was designed in support of interchanging architectural descriptions between tools. And each tool may have its own vocabulary that goes beyond what's needed for simple interchange. Acme has a mechanism for embedding within it tool specific terminology. This additional text is not interpreted by Acme other than for syntax checking. But is passed along to the various tools. And they can do their own work on it. This extension mechanism is called Acme's property sublanguage.

13 – Properties

A property in Acme is nothing more than i, than an identifier that can be associated with a value. That is, you're giving name value pairs that are then become part of the syntax syntactic description of your architecture. Examples of uses of such name value-pairs include. Visualization properties, that is, if you are not satisfied with what Acme Studio gives you, but you have other tools available to visualize architectures, you might wish to communicate information about those additional properties within an Acme description. Temporal constraints. Archi, architectures describe systems that actually run and may have timing considerations with them, and you may wish to use tools that can take advantage of this, such as simulator tools. You might like to have more detailed checking, on the data being communicated via the ports and roles, and so you might have a type checking tool. The particular communications between the components across the connectors to other components, constitutes a protocol. And you might wish to enforce that protocol. Do checking on that protocol, and so on. And hence you could use the property language to describe the protocol. If there's scheduling constraints you could put those in. If there's resource consumptions constraints you can put those in, and so on.

14 – Properties Example

Here's an example of extending our previous top level description with some properties. We still have our client and server components. We still have our RPC connector. We still have our attachments. However, we've added some property statements within the descriptions of the components and connector. In particular, the first property is labeled Aesop-style, and it's some kind of style ID. Second one is Unicon-style. Now, be aware that Aesop and Unicon are other architectural description languages with their own tools. By the way, I've indicated here a comment using C++'s slash, slash commenting style. For the second component, the server component, there are two properties, and these are not intended for particular external architectural style architectural description language. The first one is labeled idempotence, and it's got a Boolean value indicated as true. The second one is an integer, including a maximum concurrent clients that this particular component can, is capable of dealing with. As far as the connector is concerned, there are properties for synchronization. For the maximum number of roles that that connector can have and for a particular protocol, in this case, using the Wright, as in Frank Lloyd Wright, architectural description language.

15 – Families

Another feature of Acme that goes beyond the basic vocabulary of what are called Families. And Families are what you would use within Acme to describe architectural styles. That is they're defining new terms that describe sets of architectures. You can encode style rules as properties, that describe how to use a particular family.

16 – Example Family

For example, here is a brief description of a family called the pipe and filter family, which you should now be familiar with. There's a type of component, not a component but a type of component, called the filter type. Another type of component called the pipe type. And we might then use. This, this additional vocabulary in defining a system. And you notice that in the declaration of the system, there's a type given to it. In this case, it's the PipeAndFilters-Family. And we're now going to define specific components that correspond to the types we've

defined in the family description. So filter 1 is a, a type, filter type and likewise is filter 2 and then connector is of the pipe type.

17 – Open Semantic Framework

The third advanced feature of ACME is what's called its open semantic framework. ACME has a simple vocabulary, a simple syntax, and also, a very simple semantics, but there obviously could be lots of information you'd like to encode in an architectural description. And, encoding all that complexity you would like to also have some way of checking it for whether it's valid. ACME itself and the ACME tools don't provide that checking, but they do provide a way for you to essentially export your description in such a way that it can used, be used by external checking tools. That's what the open semantic framework is. In particular, the description that you've either drawn, using ACME Studio, or you've typed in, using the editor, can be used to generate a description in first order logic. That description can then be used by an external tool, which takes as input the first order logic. And proves whatever it needs to do, such, such, for example that the particular architecture you've described obey's certain rules. There's an example here for the client server situation. And it's essentially a an English language keyword version of first order logic. It says that there exists a thing called client, a thing called server, a thing called RPC such that client is a component, server is a component, RPC is a connector, and that they're attached in a way we've described. This, this contains exactly the information we saw in the ACME description. But now it's in a form that can be dealt with by a fair improver, or automatic reasoning system.

18 – Acme Features Quiz 1

Here's a quiz to see if you have understood the various features that ACME provides. In column one there's a list of the names of those features, and in column two there's a definition for them. See if you can match the term with its definition.

19 – Acme Features Quiz 1 Solution

Well, what's a role? That is, what's number one. A role is an interface for a connector, which is answer B. Open semantic framework. That's an export format for use by automatic reasoners. A family, well, that's a means for defining architectural styles. Properties. Properties are name value pairs for exporting information and ACME descriptions through external tools. Rep-map. That's a binding mechanism when we're doing vertical decomposition. Finally, port. Port is a component interface.

20 – Acme Features Quiz 2

As I've tried to indicate, ACME is relatively simple as far as ADL's go. There are lots of other features that other ADL's have that ACME does not. So I'd like you to think for a minute, what other features it might be nice for an architectural description language to have that ACME doesn't, and see if you can list them here.

21 – Acme Features Quiz 2

Well, clearly you'd like to have a description of what those components do orf what those connectors do. ACME doesn't provide this, but you can well imagine some additional syntax for describing those kinds of behaviors. Also, ACME doesn't have representation for functional properties. So going back to our original understanding of systems having function behavior and structure, ACME tells you a lot about structure, but it doesn't tell you much about function

behavior. Also, ACME doesn't directly provide a way for connecting code with architectural elements, and you can imagine that going in either direction. That is, you might like to take an architectural description and automatically generate stub code for it. Similarly, if you've got some existing code, you can imagine a tool that can do some kind of analysis on that code to generate an architectural description that can then be imported into ACME and you can visualize it using the ACME Studio Graphical Editor. Also, of course, is the fact that ACME doesn't say anything at all about non-functional requirements other than what the property sub-language allows you to describe on your own. The essential role of non-functional requirements in any architectural description means that it would be of value to come up with some way in a standard fashion, trying to characterize these particular requirements.

22 – ACME Limitations

The main goal of acme is to enable architectural descriptions to be expressed in a way that can be used by a variety of tools. That is, it's an interchange format. Because of its limited goals, it lacks features found in more elaborate architectural description languages. Nevertheless, it should give you a feel for the importance of architectural description and the role that architectural description language can play in describing these architectures.

P3L08 Refinement

01 – Complexity Abstraction

As you know the world is filled with complex problems that we try to, that we strive to solve. And our primary weapon for dealing with this complexity, is abstraction. That is we hide details so we can concentrate on the big picture. For example, when we have to pick up supplies at the grocery store for dinner. We break the problem down into traveling to the store, locating the items, paying for them, and so on. It is only when we actually driving to the store that we worry about things like what turns to take. This lesson is about managing this complexity by carefully refining a design from an abstract version into an implementation

02 – Levels of Abstraction

Abstraction implies thinking at different levels. For example, there is the level of solving our hunger problem, where we think about food items and where we can obtain them. An entirely separate level has to do with our physical body movements, such as insert the key into the ignition or packing the grocery bag. For abstraction to help us solve a design problem, a key property must hold. The work done at a lower level of abstraction must contribute to the solution at the higher level. It doesn't help us to solve our hunger problem if all we pack into the grocery bag are paper towels and people magazine.

03 – Divide Conquer

In summary, solving a design problem at one level of abstraction means dividing it into subproblems. Solving each of the subproblems at a lower level of abstraction and ensuring that the lower level solution, indeed, contributes to the higher level. That is, divide and conquer.

04 – Horizontal Decomposition

I want to talk about two different kinds of dividing. First off, horizontally dividing things. You can think of design as a three-dimensional picture puzzle, where you have to cut out the pieces. At any one level of abstraction, you carve out the pieces and fit them together. This is called horizontal decomposition

05 – Vertical Decomposition

However, solving a problem at one level is not enough. For each piece at that level, you have a whole picture puzzle at a lower level. This ladder is called a refinement of the former. Devising the puzzles at a lower level, and ensuring that they satisfy the needs at the upper level is called vertical decomposition. Of course the above holds true for each of the top level pieces, and for however many levels are involved in your decomposition. So what we really have are a collection of refinements at one level addressing a, a problem at a higher level.

06 – Proper Refinements

All non trivial design involves refinement. Moving from a high level of understanding to an implementation. Multiple levels may be involved, not just two, depending on the complexity of the problem. To execute a multilevel design properly, three properties must obtain. The top level must faithfully represent the requirements. Each level must be internally consistent. And third, each lower level must faithfully represent the level above it.

07 – Property 1

First, Property 1. Property 1 holds of any program that we want to implement. It must satisfy it's requirements, obviously. You check this property using traditional methods such as software testing, group reviews, and customer acceptance criteria

08 – Bank Account

In this lesson we are going to do an example of a refinement. We will start with a very simple bank account application that allows deposits, withdrawals, and queries about the current balance. Here are some possible requirements. The user can make a deposit of any positive number of dollars. The user may make a withdrawal of any positive number of dollars so long as at least that number of dollars is currently held in the account. Their user may request the current value of the bank balance, which is defined as the net value of all the deposits made minus the sum of all the withdrawals. And initially the bank account is empty.

09 – Bank Account Class

Now, let's take it a step further and produce a UML analysis model of those requirements. In particular, let's come up with a particular class called the account class and you can make some assumptions. You can assume that all of the deposits and withdrawals are made in amounts that are positive integer number of dollars. And you also do not need to include an attribute for the current bank balance. We will add that as a refinement later. Recall that a class description UML has a rectangle with three boxes. The top box is the name of the class. Here we're going to call it account. The middle box has any attributes of that class. And here the attribute that makes sense is some history of the users' transactions, the, the withdrawals and deposits. I call that particular attribute transactions. And I'm treating it as a sequence of integers, because we already assume that all the. wi, deposits and withdrawals within in integers. The class also has three operations, corresponding to the requirements. One is deposits that takes an integer value and doesn't return anything. Second is a withdrawal which takes a. Integer value and doesn't return anything. And the third is the query asking for the balance, which doesn't take any arguments but returns an integer value.

10 – Bank Account Quiz 1

So far, so good. But there's one more step to go, and that is we have to say things a little bit more precisely about what's going on. So I'd like you to give some OCL constraints for each of those three operations, and for any class invariants. Confirm to yourself that the class definition and specifications together satisfy the problem requirements. In other words, satisfying property one.

11 – Bank Account Quiz 1 Solution

There are four constraints involved. The first one has to do with the deposit operation, and the precondition, as was stated in the requirements, is that the amount being deposited is possible. But the post condition is a little bit more subtle. In particular, we decided to keep a record of the deposits and withdrawals in terms of a sequence of integer values. The order of the sequence corresponds to the order in which those deposits and withdrawals are made. So the post condition for the deposit operation is that whatever sequence you had of transactions at the start of the operation, at the end will have one more added to it, which has as its value the amount that was deposited in that step. Similarly, for the withdrawal operation the precondition is the same, however it had better be the case that, as the requirements stated, that the amount being withdrawn if the account has enough money in it that you can make that withdrawal. The way that we express that part of it is that the sum of the transactions made so far is greater than or equal to the amount being withdrawn. Post condition is similar to the post condition for the deposit except the amount being appended will be the negative of the argument indicating that the amount has been withdrawn. The third operation is the balance query. This has no precondition, but the post condition is the results returned by the operation are equal to the sum of all the transactions so far. That is, we go through them and add and subtract as appropriate to get to the current balance, and return that value as the results of this operation. Finally, one other constraint to be aware of, and this is the case for all such models that you produce, you have to set up the initial conditions. OCL has the keyword in it to indicate that, and in our case the initial condition is that the list of transactions or sequence of transactions is empty to start with.

12 – Property 2

The first property that we looked at is concerned with whether we understand the problem we are trying to solve. Property two is concerned with whether we have solved it correctly. We can state property two as follows, each level in a design must be internally consistent. Technically, what this means is that we must make sure that for each operation. Defined at that level. Each operation leaves whatever invariants are there true. That is, if the invariant was true before the operation took place it's still true afterwards. The operations don't break anything.

13 – Notation

One of the objectives of this lesson is to give you some tools to think carefully about dealing with complexed, that is the management of these levels of abstraction. So to do that, I am going to introduce notation. Okay, this notation is taken from first OR logic. To state property too precisely, we are going to need a little of that notation. And throughout the rest of this lesson, we are also going to use the term abstract and concrete to refer to the upper and lower levels of a refinement. First off, we are going to talk about the abstract operations and we are going to use P1, P2 and so on up to Pn and Pi we'll use to indicate a typical abstract operation. We're going to have abstract states. The set of abstract states is the capital letter S and a typical abstract, typical element of the abstract state is the letter s. And we'll also put an apostrophe after the s if what we're talking about is the abstract state after an operation. We have invariants, we use inv to indicate invariants and invA will stand for abstract invariants and invC will correspond to the concrete invariants. We have pre-conditions and post-conditions and we'll just append the word pre and post to our Pi's to indicate. Whether it's a pre condition or a post condition, and those pre and post conditions can have states

associated with them and arguments. Finally we'll use symbols like the ampersand to indicate and, and the rightward facing arrow to indicate implies.

14 – Valid Operations

Property 2 says that each level must be consistent. In other words, that Operations preserve invariance. We will use the term Valid to denote Operations with these properties. That is, we wanted to be to, want to ensure that all our operations are valid. Using the rotations we just introduced, we can express Property 2 precisely by saying that for each operation. The following must hold. Well, first off, that we have an abstract invariant over state s', and we have the preconditions for s, and we have the postconditions for s. If we have all three of those things, then the invariant, the abstract invariant must hold on the state afterwards. Remember that that's the state s' with an apostrophe on it. You have, invariant true before on the state. You have the pre-conditions all set so the operation can run. If the operation runs and leaves the post conditions the way we expect. And it better be the case that the invariants are true on the state resulting. Now that said for the model that we've done so far there is no invariance. Okay, but we're going to need this later when we get to the refinement so I thought I'd introduce it now.

15 – Implications of Property 2

So, we've got some, some first order logic. What does this really mean to you as a developer. Say you have a spec, you could have written it, could have been given it, or whatever, and you want to implement it directly. So we're going from, from spec directly to implementation, just one level of refinement. Property two says that you have an obligation to make sure that each operation in the spec doesn't break anything. Any invariant. You can do this. You can insure yourself of this by testing, by code reviews or even by proving. But somehow you, you have to do it.

16 – Property 3

Property three is where things get interesting. Okay? It states that each lower level of refinement, must faithfully represent the level above it. Okay? But this is going to be a little bit tricky to express, and involves some subtleties, we're going to have to get at. It's the most involved of the three properties. And, the question one asks is, how exactly can we determine whether a level implements the piece that is abstracting it? Before answering that, let's go back to our example bank account and add a level of, refinement to it.

17 – Bank Account Refinement

In the abstract version of the bank account application, each time we wanted to know the bank balance, we had to add up all the previous deposits and withdrawals. And if you've done any kind of implementation out there, you know that that screams for having some kind of temporary variable that holds the intermediate results so you don't have add those all up every time. So let's add that in as refinement to our to our spec. That is we're going to add another level of refinement, in which we're going to implement this particular optimization. Okay, that will reduce the number of computation that we had to do, at the expense of adding this temporary variable in. For the class model itself, there's just one additional line. And this is in the attribute box, and it adds in a variable called runningTotal, which is an integer.

18 – Bank Account Quiz 2

Of course, you also have to update your operations to update this temporary variable during the course of making deposits and withdrawals and so on.

19 – Bank Account Quiz 2 Solution

So as far as a deposit is concerned, in addition to appending that particular amount into the transactions list, we have to update the running total. And, as you recall from when we talked about doing OCL invariants when we're updating state. You have an equation, in this case, it says running total equals running total at pre. That is the value before the operation started, plus the amount of the deposit, which is A. Similarly for withdrawals, we have to say that the running total that is at the end, equals the running total at the start minus the amount being withdrawn. As far as the balance operation is concerned, here's where we save those computations. No longer have to do the sum, instead we can just return as the requested value the running total. As far as initialization is concerned, in addition to initializing the transactions to be an empty sequence, we have to initialize the running total to be zero. And finally, here's the invariant, here's what we have to ensure is always true. Which is that the value of the running total has to, at all times, be equal to the sum of the transactions. Otherwise this optimization that we are proposing to add in our refinement is not going to be correct. 'Kay, note that, as in this example, it is typical for invariants to relate the values of two or more attributes.

20 – Property 3 Details

Well now that we've got the exercise part out of the way let's return to the details of property three and see what it means to faithfully represent that have the refinement faithfully represent the specification. In fact we have to check three things. There's three criteria that we have to look at. First of all, is the refinement adequate? That is, is our refinement rich enough to represent all of the abstract situations? Second, is the refinement total? That is, are we sure that the cron, concrete level can't get into some state that doesn't correspond to a possible abstract state? And the third criterion is, does the refinement model the abstract level. That is, is each abstract operation correctly implemented by a concrete one? We'll look at all three of these criteria separately, after we introduce a little more notation.

21 – More Notation

First off notation wise, they correspond to the concrete operations. We're going to have variables Q sub 1, Q sub 2, so on, to correspond to the abstract ones which were the Ps. And similarly to correspond to the abstract state which was indicated by s. We're going to have the concrete state indicated by t as a typical concrete state element. And capital T as the set of concrete states. And then, the interesting one is that we're going to introduce a function, a retrieve function. Here, I'm using retr as an abbreviation for it. And it's going to map from concrete states to abstract ones. That is, retrieve on some t, concrete state t, is going to be corresponding to some abstract state s. And it's always going to be the case that, t goes to a single s. However, it might be the case that different ts go to the same s. You kind of expect that because at the lower level we have more details. And so, we can get into more states. But those states are going to map into a fewer number of abstract states.

22 – Adequate Representation

The first property three criterion to consider is adequacy. That is, the lower level implementation must be rich enough that each abstract state is represented by a concrete state. State it precisely, for each abstract invariant, if the invariant is true in a state. Then there must exist a corresponding state in the implementation in which that the corresponding concrete invariant is also true. And in logic, okay, for every, every abstract state S, okay, if the abstract invariant is true for S, then there must exist some state T down in the concrete. implementation. And some, some, concrete invariant over that state T, such that if you were to apply the retrieve function to the concrete state, you would get the abstract state.

23 – Note on the Exercise

So thinking about the exercise for a second, what does adequacy mean? Well, there's no abstract invariant. And that really means you don't have to worry about anything for this particular property, for this exercise. Or you can think about the in, the abstract invariant as just being the Boolean, value, true. If we make this simplification then the above adequacy criterion looks like the following. For every abstract state S, there's going to be a concrete state T. And in that state the concrete invariance hold and if we apply the retrieve function to that state we get back to S. What this means is that we have to check for each situation we can get into, with the abstract specification there corresponds a valid concrete one.

24 – Adequacy Quiz

So, for an exercise, if I give you the abstract state that looks like a deposit of three, withdrawal of two, deposits of four and five and then a withdrawal of six, I'd like you to give me the corresponding concrete state.

25 – Adequacy Quiz

Well, as you recall, the concrete state is like the abstract state except we've added in a running total to it. And if we do the addition and, and if I've done my addition correctly, then the value of the running total attribute is going to be 4 at this time. That is the valid concrete state is identical to the abstract one with the running total added in.

26 – Total Representation

That was the first criterion adequacy. The second property three criterion is totality. That is our implementation can't put us in a concrete state that doesn't correspond to an abstract one. No memory fault core dump messages. Memory fault core dump would be a concrete state. Okay? And it doesn't correspond anything in our spec. And we want to prevent such situations from arising. We call this criterion total or totality because we must make sure that the retrieve function is total in the mathematical sense. That it's defined at every point. Here is how this property looks if we express it formally. For every concrete state t, if the invariants are true on t, then it had better be the case that there exists, some kind of abstract state s. That's the result of retrieving out of t. And the invariants of the abstract invariants must hold on, on state s.

27 – Totality Quiz

So let's now turn the exercise around. I'm going to give you a concrete state and ask you to tell me what the corresponding abstract state is. So the concrete state has the transaction deposit 13, withdrawal 12, deposits three and five and then withdrawal six. And the running total of this is the amount of $3. Now I ask you to give me, what is is the abstract state that corresponds to this?

28 – Totality Quiz

Simple right? We just get rid of the running total. The transactions remain the same.

29 – Models

The third and last criterion to examine has to do with concrete operations, modelling abstract ones. That is, each concrete operation must faithfully reflect the intent of the abstract specification. You had to do your implementation right. And this criterion has two parts, having to do with inputs and outputs to operations. That is to refine operations, we must assure ourselves of two things. The implementation must be able to handle all the imports described in the specification. And the outputs produced by the operation, along with any changes made to class attributes or other side effects, must satisfy the operation specification.

30 – Operation Inputs

As far as inputs are concerned, inputs acceptable to an abstract specification must also be acceptable to the concrete implementation. However, the refined operations, that is the implementation, can accept more. This might happen if for example you use a particular library routine that's very general to deal with satisfying some con, some abstract need. Okay, you can accept more inputs, but it has to accept at least as many as the abstract one requires of us. In terms of our specification stated more precisely, refinements can weaken operation preconditions. And in logic where every, concrete state T, if the invariant is true an the, preconditions are true in the abstract state, then it better be the case that the preconditions are true in the concrete state.

31 – Interpretation

To perhaps get you a little more comfortable with looking at first order logic, we're going to parse this a piece at a time. First, we are concerned with valid concrete states T. In which all the invariants, that is invC corresponds to concrete invariants, hold on that state. Now consider the corresponding abstract states and remember the retrieve function. We can go from the concrete states to the abstract states by using the retrieve function. If we are in the abstract state, the corresponding abstract state. And we want to execute the abstract operation with arguments, some argument list, called args here. Then of course, it's preconditioned. Which we've indicated with the prefix ERE, on the retrieve state, and the arguments must hold. If the abstract precondition does hold, and if we have implemented the operation successfully. Then the concrete precondition, which has the, uses the Q version on the concrete state t, must also hold.

32 – Inputs Quiz

For our banking application, which of the three operations in the account class that are affected by this part of the modeling criteria? Check all that apply.

33 – Inputs Quiz

The answer is "Withdrawal."

34 – Outputs

Well that was inputs, now let's look at outputs. In particular the outputs produced by the concrete operations along with any changes made to class attributes and other side effects must satisfy the abstract specification. Among other things this says the answers that your implementation give had better satisfy what is required by the specification. Stated in logic. If we have some concrete state t, and the invariant is true, and the preconditions are, the concrete preconditions are true, and the, after executing the operation the concrete post conditions are true, then it better be the case that the abstract post conditions are true. Parsing this one step by step like we did for the inputs. Valid concrete states as the part that says that for all t element of the set of concrete states, capital T would be invariant. The concrete invariant must hold on that state. And the concrete preconditions are satisfied, and that's the Pre-Q sub i on t and whatever arguments we have. And running the country operation satisfies the concrete post conditions. That's the phrase with post Qi on a concrete state. The arguments, the resulting concrete state T apostrophe, and whatever results are produced. Then the post condition had better map to the stated abstract post condition, the last part of our post logic. Which say's the post condition on the abstract operation Pi, with the retrieve state, the arguments, the retrieved version of the resultant state, and the results produced had better hold. Note that as you might expect, the implement, the implementation can do more than what the spec says as long as it does at least what the spec says.

35 – Outputs Quiz

Let's ask the same question we did a minute ago about the bank account class. Which of the three operations in the Account class are affected by this part of the modeling criterion? That is, the part that has to do with outputs. Check all that apply.

36 – Outputs Quiz

Well this time, all apply. You should have a check in each one of the boxes. All the operations are affected because all of the concrete post conditions refer to running total, the account balance attribute.

37 – Satisfy Property 2

So now that we've, we've got together all the criteria, and the, first order logic expression of things, let's go back to Property 2 for a moment. That requirement that the concrete implementation is consistent. Recall that this means that all operations preserve invariants. The relat, relevant invariant is that running total reflects the sum of the transactions that have taken place. So we have to make sure that each of the operations if that invariant is true before hand, it's also we true, it is also true afterwards. So the balance operation is not a problem because it doesn't effect the value if the variables. For the other two we would need an inductive argument that the invariant is initially true and that deposits and withdrawals keep it true. This intern relies on the fact that the sum operation can be defined in terms of re, repeated additions and subtractions.

38 – Summary

Pulling this all together. The problem is that design is tough. And design mistakes can be costly. Our chief weapons against complexity are abstraction and refinement. So we must make sure that our refinements do what we expect. One way to do this is, is to systematically think about what refinements must guarantee. Here they are, the top level specification, matches the requirements document. Operations at each level preserve invariants. Each refinement is adequate. Each refinement is total. And the concrete operation preconditions and post conditions model their abstract counterparts.

P3L09 Middleware

01 – Architecture of Distributed Systems

So far, in this course, we have considered software architectures. Today, I would like to get into one particular kind of software architecture, the architecture for distributed systems. And all this means is there's multiple computers involved in providing whatever application technology system it's supposed to. Moreover, we will consider heterogeneous distributed systems. Systems in which the different computers are different kinds and provide different capabilities. An example of this is a typical client server system for example, insurance agents go out into the field, they come to your house, they talk to you about various insurance options, and they have their laptops. The laptops are connected to some server machine that, understands the different forms you have to fill out, and the and the rates involved in those in those particular contracts. And ultimately if you sign up for a particular policy then the business logic connects to a data base that stores the information about you and your particular policy arrangements. In that particular situation, there was a a database server machine, there was a web server that was responsible for initiating and computing business logic. And there were various web browsers running on laptop machines, one for each agent. We'd like to look a little bit more in how these particular kinds of systems are architected.

02 – Middleware

Because these types of applications are so common, a group of technologies has evolved to support them. These technologies are called Middleware. That is, they are technologies between the user or client and the ultimate server. They essentially are responsible for dealing with satisfying the non-functional constraints of the overall distributed system. They're similar, in concept, to the architectural connectors that we've talked about. It will be interesting to contrast those connectors with the kind of middlewares described. We'll be using in particular the material taken from the Emmerich paper, which I've asked you to read.

03 – Context

Why is this an important application class to consider? Well, obviously the Internet has just opened up so many different opportunities for applications. That means, not only more applications but more customers, heavier load, increasing demand on the resources. Hence, performance and resource consumptions considerations, nonfunctional constraints. Also, the hardware resources themselves have become more specialized. Think about ATM machines or card readers at gas stations, or mobile phones, or square card readers. Just all kinds of devices that have their own peculiarities. Also we've seen increasingly powerful applications, okay, that involve integrations of existing components, and sometimes even negotiation among them.

04 – Needs

Well, if we're to deal with this challenge of increasing growth in distributed applications, we should consider what kinds of technologies we can use to support them. One category are technologies that provide abstractions, okay? In particular, the abstractions take the form of interfaces realized as application programmer interfaces, or APIs, such as Ajax or Enterprise JavaBeans. Standards, think HTTP or new kinds of architectures such as REST architectures or service oriented architectures. similarly, the development of new notations. Over the last, maybe ten years we've seen the increasing penetration of XML into applications. And similarly, there's been the development of tools and that support the development of these kinds of systems such as WebSphere or Hibernate. You may have heard of those. In the design space, there's Oracle's Fusion and there are even code generators for, for transforming or generating the XML such as XSLT.

05 – Exercise Application

Going through this lesson, there will be some opportunities for quizzes. And what I'd like to do is have you consider a simple application that I'm, then, going to be asking you questions about. The application involves users and a web browser, visiting some kind of website and being asked to vote on certain questions. The votes are then submitted and the system responds back after registering the vote with some count of how other people voted on the same, same question. So the application is web-based, users are polled on a variety of questions, their choices are recorded, and there's a display of how others voted. And, in this particular situation the actual users remain anonymous as far as the database is concerned.

06 – Characteristic Issues

So before, we get into the describing the technologies involved and addressing these problems. Let's look at little deeper into the issues, some of the issues that might involved. We want to look at network communication issues, coordination among the pieces of the application reliability concerns, scalability concerns and then just the implications of the heterogeneity of the various parts of the system.

07 – Network Communication

Let's begin with network communications. Obviously, if you have a distributed application the pieces of the application have to be connected together over some kind of network. Some of the issues that arise in dealing with applications that are split across a network are how are errors handled. And there a variety of different kinds of errors that might arise. There are synchronous errors. That is, when a particular request is made, and part of a system awaits a response. Or asynchronous errors. That is spontaneous issuing of some kind of notification that the rest of the system has to respond to. Primary concern here is reliable delivery of these messages, particularly the error messages. If the network itself is unreliable, various strategies such as delivering more than once can be employed, but then again, you don't want to have more than one actual copy of a message received and, and acted upon. And communicating across the network, another issue relates to how the data is represented. It may well be the case, because we're in a heterogeneous situation, that the different machine and the different machines represent data in different ways. And then third is the whole question of transaction. If there's a database involved, it may well be the case that of the multiple users which might be using the application, one user is updating the database at the same time another user is

reading the database. And there's the potential for the receipt of information that is not up to date with respect to the, the database.

08 – Data Transportability

To drill down for a minute into this question of data transport ability. This is sometimes called serialization in Java, or marshaling, or even pickling. Some of the differences that might arrive have to do with bit order. Yes, some machines order the bits from high order to low order in different orders. Byte orders within a word. Different character sets that are used. Alignment, that is, whether, particular pieces of data are aligned on word boundaries or byte boundaries. Okay? And then the whole question of word length. Now we have, of course, 64-bit, words, but other machines are only 32-bits. And, of course there are, uses, of, data storage in which we wish to use less than a full word. Even after these, differences are resolved there's still the question of how the various pieces of some complex data are, are, are organized. Okay? Must they be kept in a certain order? Can they be, can they be shuffled around in order to compact them? These days, many, approaches to data representation include self-definition. That is, not only are you communicating with data, but you're communicating at a description of the data. That, can then be decoded at the other end. Fortunately, various, standards have, arisen to address these kinds of questions. Internet standard x680 is one, and then on the commercial side Google has developed the idea of protocol buffers. Which are descriptions of data. Which then can be incorporated into your applications for communication among the pieces and interpretation at either end by an API.

09 – Voting Application Quiz 1

So going back to our sample application for a moment, here's the first quiz. Think of how I describe this application, and decide what date it is that needs to be transmitted from the web browser to the server in that direction.

10 – Voting Application Quiz 1 Solution

Of course I expect that you recognize the need to transmit the votes but did you also realize that we have to transmit the question numbers. After all if we're storing these things in the data base we have to index into the database to find the right question information in order to update the statistics and eventually return the reply

11 – Transactions ACID

In the situations where a database is involved and we have to be concerned about transactions, that is making sure that the data that we store and retrieve is consistent with respect to the overall database we have to deal with transaction processing, and in particular with what are called the ACID properties. The primary concern is reliable database access, particularly when there are multiple readers and writers. That is parts of the application reading from the database and other parts writing into the database, particularly on in addressing a particular, a specific record. Imagine that you were in a situation, maybe you were doing airline reservations or something like that, and you need to understand the information in the database, but you may want to update it if you're making that reservation. Okay? So you have to read it, examine it, change it, and then write it back. What happens if another application changes the record after you've read it, but before your change version has been written back. So we'd like to have database transactions that satisfy the ACID properties, and ACID is an acronym. The A stands for Atomic. That means that your transaction, these four steps, reading, examining, changing, and written, are all treated by the system as if they were one

step with no intermittent activities going on. The C stands for consistency preserving, in the sense of database integrity. Okay. That is if the database satisfied all its integrity constraints before your transaction, it will also satisfy them after your transaction. Third, the letter I stands for isolated. Okay, that, what that means is that other transactions can't see into any intermediate states in the processing of your particular transaction. And finally D stands for durable, and that mean that once your transaction commits then it's persisted. That means it, it is a permanent record that the system is aware of.

12 – Voting Application Quiz 2

So second quiz question related to the sample application is whether or not you see a need in this application for transactions.

13 – Voting Application Quiz 2 Solution

In this particular application even though the user is going to see some votes. If the votes were off by one or two it probably wouldn't matter to the application. Providing transactions in a database since can have some extra costs associated with it. And the question is do you want to pay that cost if getting the exact right numbers is not all that important. So I wouldn't answer this particular question's that transactions are not necessarily required for.

14 – Coordination

The second category of issues is coordination. Remember that we have heterogeneous distributed application, multiple things going on at the same time. And these multiple things have to synchronize across certain actions. Okay, and how is that going to be provided? The two main categories of a synchronization are synchronous applications, synch, synchronous communications and asynchronous communications. Synchronous means that when one piece of the application initiates that particular message or interaction that it waits until it gets a reply back, a response back, before continuing. Often synchronous types of interactions are clocked. That means that there's some heartbeat or some other measure of when it can do certain things. On the other hand asynchronous communication means the client can continue to execute after it sent the message and it's notified when the response comes back and it can take appropriate steps at that point. Obviously asynchronous is more general, but writing code for dealing with a, asynchronous coordination is a little trickier, it makes it a little harder to understand. So once again there's a trade off. Another question with respect to coordination is, who's in charge here? Okay, is it the client or the server? You've no doubt seen situations where the server is capable of pushing things out to the client such as web pages which you want to update with current events or it could be that the client requests information as it pulls from the server. Deciding how you're going to deal with that is a key design question in any distributed application. It's always the case with such applications that robustness is important. What this means is that the system can deal with situations where one or more of its components goes down. Think that you send out a message and you don't get any response back. And the reason that you don't get any response back is that the the piece of the application you were dealing with goes down. How does your part of the application deal with that? Think for example, if you were user facing you don't want to just wait there and leave the user in limbo. You want to provide perhaps some time map on a message acknowledgment and be able to let the user know what went on. Similar to robustness is availability. How does the system appear to the user as far as being available? Is it 24/7, type application? Does it have set maintenance times? How does it deal with load situations? That is, does it get so slow that the user gets frustrated? All those kinds of questions. Persistence we've already

mentioned. In general this means how a server stayed maintained. One obvious approach is with a data base. It might be a file system, but nevertheless the choice has to be made. And if you've got multiple clients talking to the system, how is that concurrency handled? Okay, how does the server part of the application deal with all these multiple users? Related to that is the transaction and integrity constraints which we've talked about on the previous issue.

15 – Voting Application Quiz 3

Going back to our sample application situation. We have this voting application. Should it be the client or the server that initiates interactions? Write your answer into the text box.

16 – Voting Application Quiz 3 Solution

Well, there's two choices for you. Either the client could do it or the server could do it. In the client situation, the client sends the votes at the point which the user makes them. If it were the server doing it, the server would have to periodically poll the clients, to know if there were any votes that it needed to get. Seems to me that in this situation, because the user interactions are intermittent, that having the client initiate the interaction probably makes more sense.

17 – Reliability

Their major issue is reliability, which we've alluded to already. We have a complex system that has multiple pieces to it and that means a piece might go down, or might, be overloaded at, at some particular point and how does the rest of the system, respond to that. Overall we can call that a reliability question. That is what percentage of the time is the overall application providing the, the services that it should be providing. Typically you learn about a reliability problem when some message that is sent is not delivered. Or at least you don't get an acknowledgment back from it. When strategy is to try sending the message again. The danger there of course is that the, recipient receives two copies of the message, and does whatever action is doing twice which also may be a problem. This is an example of a classic reliability performance tradeoff. That is. Whenever you use replication that's going to take more time and more resources, as well as, as, as compromising the integrity of the system. You can do it faster but it may not be as reliable. Various policies for dealing with reliabilities use have arisen such as the, the client making its best effort. The client's saying it will do at most once, as far as message sending, at least once, exactly once and so on.

18 – Voting Application Quiz 4

How many different machines might be involved in processing the vote of a single user?

19 – Voting Application Quiz 4 Solution

Well, there is three different things going on. There is the web browser that's spacing the user and processing the immediate user events such as clicking on the buttons. There is the web server that serves the page but also receives back the information from the user and then there is a database server. So at least three different processes are going on. Now these might be on three different physical machines. It can even be on a single machine.

20 – Voting Application Quiz 5

Second part of the quiz, whenever you're involved with different machines and having to send messages back and forth. The question of protocols arises. That is, how is the message formatted, packaged up, in order to be understood at the receiving end? So, the question here is, what protocols might be used in support of the sample applications?

21 – Voting Application Quiz 5 Solution

Well, one obvious answer is HTTP. This is a web based application, and most, web communication uses the HTTP protocol. These days, a second protocol often is part of these kinds of interactions, and that's AJAX, in which the web browser, and the web server cooperate. In a more timely fashion, then going in a kind of round trip HTTP page rewrite situation. Third, is the fact that, ultimately we're updating a database. Typical databases are relational databases and the standard language for dealing with relational databases is SQL. So SQL can be thought of as a protocol. In typical applications like that, the SQL is embedded inside of PHP. You can think of PHP, as a language but any language is in itself a protocol, that is, has certain rules for how it's programs are expressed.

22 – Scalablity

What major issue is Scalability. If you've got an interesting application, that's distributed across the internet, it might grow. That is, it might, have more users over time. Or it might have, a greater load, due to the individual users. In either case, you want to be able to grow the application. And the question then is, how easy is it to grow? This is sometimes called scaling. That is, how easy is it to scale your application? Scaling typically takes the form of adding new hardware. And the question then becomes, to what extent will adding new machines change your system architecture, or its components? Ideally, we'd like to grow transparently. What transparency means is, whether or not a particular aspect of a systems architecture is visible externally, that is, to the, the other machines. If it's invisible, of course, you can scale as much as you want, and the rest of the system would not be concerned.

23 – Kinds of Transparancy

So, the different kinds of Transparency, which you might consider versus Access transparency. That is, it's a particular computational or data resource which the application requires available locally or remotely and in particular does the application need to know whether it's local or remote. If it's Access transparent the application doesn't need to know. Extending this is the idea of Location transparency. Is it necessary to know the physical location of a resource. Third, is Migration transparency, that is, can we move a resource from one. Physical machine to another, in such a way that the rest of the system doesn't have to know that it's moved. This can be tricky of course, if you're in the middle of using a resource when it moves, there's the possibility that, your interaction is going to be inconsistent, or even break. And finally, Replication transparency. One way for dealing with reliability issues is Replication. At one extreme would be Database Replication, where all of your data is in more than one place, and if your, one of your databases go down, at least you have the other one there. Okay, and then as far as Transparency is concerned, is your application aware that there are multiple database servers in that situation.

24 – Heterogeneity

The fifth major category of issues is heterogeneity. We started with the assumption, we have a heterogeneous distributed system, but the dimensions of heterogeneity are many. Hardware, this not just what kind of computers, but also are there embedded devices involved. Are there cellphones involved in this? Are there card readers involved? Are there other, other devices? Different operating systems. Different programming languages. All of the different standards protocols and API's that might be involved in building this application. The pieces of the application that provide access such as a web browsers or these devices. All of these can change. More over even if they don't change. Even if they're uniformly applied. That is every part of the system uses Linux or every part of the system uses Ruby. Are they using the same version? For example, imagine the issue of testing a web based application across browser versions. You have three or four main families of browsers, each coming in three or four different versions. Also the problem of testing becomes that much more challenging.

25 – Implications of Heterogeneity

How are you going to deal with a question of heterogeneity? Some approaches include standard APIs. Those standards come from organizations like W3C, that is the World Wide Web Consortium, OMG the Object Management Group which is the promogater of GML, ANSI American National Standards Institute, international standards organization. All of these standard API's and protocols must at least address the issue of backward and forward compatibility. Backwards compatibility means if you've got a new version, does that version support or break older versions. Tougher is forward compatibility. In your current version, are you promising that regardless of how the particular standard may change in the future, that the current version will be supported. Second is the question of normative architectures. Here OMG is taking a stab with what they call Model-Driven Architecture, MDA, in which the parts of a systems architecture which are machine independent are separated from the machine dependent parts. And then there is a co-generation process that enables the generation of the dependent parts from some kinds of machine descriptions. The major vendors or players in the game also offer there own ways of dealing with heterogeneity. For example, Sony has its what originally was called J2E and is now called JEE. Microsoft has .Net, IBM has Websphere and so on.

26 – LAMP Quiz

For our sample application, there's a call it a normative architecture or a mild form of normative architecture called LAMP. Look up this acronym, and for each of its four letters, each of its four parts, determine the role that, that part might play in the application architecture.

27 – LAMP Quiz Solution

The L stands for Linux, and that's the operating system on which probably the server side of the application is going to run. Linux is a well controlled and uniformed operating system family, and you can take advantage of that as a way of controlling the variation that's going on in the system. The A stands for Apache, and what's meant here is the Apache web server. The Apache web server is the most popular web server, and using it and its configuration file enables you to evolve in a controlled fashion. The M stands for MySQL, which is now part of Oracle. It's the database engine, it supports a standard relational type of database access using the SQL language, so there's another standard. And the P stands for PHP, which is a

means of essentially embedding the database interactions into your HTML program so that on the server side, that can then become SQL transactions to the database.

28 – Other Non Functional Issues

Some other non-functional issues which we've maybe alluded to but that also play a role in dealing with these distributed heterogeneous application. Include fault tolerance. How does the system deal with, let's say, exceptions that arise. Flexibility, how to support, change in any of the components. In general, you want to be able to support reuse. What that means is, can you take some of the pieces, the components, and use them in related applications? Thereby, improving your productivity. In general, these applications can get quite complexed and they are often mission critical. So you need to be able to manage that complexity. And fifth is quality service. Quality of service or QOS is an, is a concept who's goal is to be able to specifically measure the non-functional constraints. And the extent to which the system deals with those. With performance, that's easy. You can, you can measure the things like response time or throughput. But some of the other non-functional considerations are harder to get a number around.

29 – Challenges

Well that's the set of challenges. Variety of different problems that successful distributed systems have to deal with. And the overall question is, what solution approaches have been developed for dealing with it. In particular we're going to look at middleware. Middleware between the client and the server that supports, in particular, addressing some of these non-functional problems.

30 – Kinds of Middleware

The first thing we're going to do is carve the problem up into four kinds of middleware. And the four kinds are going to be based upon the interaction mechanisms between the pieces. In particular we'll look at transactional, middleware because it deals with distributive transactions. We'll look at message oriented middleware in which the interactions are messages, message passing. We'll look at procedural mid, middleware, which has to do with remote procedure calls. And finally we'll look at object or component middleware in which we're making requests on remote objects.

31 – Transactional Middleware

First off, transactional middleware, recall that we have to deal with reliability and those acid requirements. Various approaches including two-phased commit are policies which the application can obey in order to get higher reliability and guaranteed consistency. We'd like to develop a transactional middleware solutions in such a way that we have location transparency. That is, the pieces the server doesn't know where the clients are distributed. And moreover, the clients don't know where the server is other than possibly with some kind of IP address for the web server. Some of the products over the years that have been developed for dealing with transactional middleware include CICS on IBM mainframes. Tuxedo, which is a UNIX-based approach, and Encina, from Hewlett Packard.

32 – Message Oriented Middleware

Second kind of middleware is MOM, okay, Message-Oriented Middleware. Here we are thinking about asynchronous message passing. Each of the pieces acts more or less autonomally, autonomously sending out messages when it needs to. These messages are queued up, so, we have some degree of fault tolerance because if a piece goes down the informa, the messages it has to deal with are still in the queue. And when it comes back up it can peel them off the queue. Message-Oriented Middleware is not particularly transparent because the clients must implement the coordination embedded in these messages. So messages not only transmit data, they also transmit information about about state. Some of the products that are involved here include IBM's MQSeries, SUN's Java Message Queues and and some of Amazon Queuing Solutions.

33 – Procedural Middleware

You've possibly in the past used remote procedure calls. This is an example of procedural middleware. The idea here is that, your piece of a system needs some computation that's available on another piece, and you'd like to make it look, in your code, as if you're just making some kind of function call. When really, the function that's doing the comp, computing is on another machine. Remote procedure calling, technology hasn't been available since the 1980s. It is typically synchronous, okay, that is you block until you get the procedure call comes back. However, it's operating system dependent. And there have been technologies developed by SUN and, NDR, for dealing with the data representations and, and the coordination of the call and return.

34 – Object and Component Middleware

The most ambitious and most recent type of Middleware is sometimes called Object, or Component, Middleware. This is an extension of remote procedure calls to deal with objects. Thinking instead of making a function call, you're sending a message to an object which might happen to be on a remote machine. Some issues arise when we do this. One is what's called object identity. If you're on a single machine, the identity of an object is really its memory address. If you've got multiple machines, those memory addresses you can no longer guarantee to be unique. So you need some kind of a global mechanism for numbering or naming the various objects so that you can send message to the appropriate object. And then inheritance. Could it be that a child instance is on one machine and a parent instance is on another machine and there's delegation of message passing from child to parent. Features that object and, and component middleware might provide include both synchronous and asynchronous message passing. Marshaling of data. Exception handling across machines. Some of the the product approaches for dealing with this include CORBA which was a mainframe approach developed in the 1990s. COM from Microsoft. And then SUN, SUN's now Oracle's Java remote messaging invocation RMI

35 – Middleware Quiz

For our sample application, here's another quiz question for you. Which of these four types of middleware do you think would be the most appropriate one to deal with a voting application?

36 – Middleware Quiz Solution

Well, there's a database involved, so it's likely transactional middleware is the one that we need in this case.

37 – Software Engineering Issues

So far, the issues that we've been talking about have been mostly system issues, things like reliability. I'd also like to spend a moment talking about software engineering side of things, that is building these systems. First category of issue has to do with requirements. Because non-functional requirements dominate this sort of situation, the requirements analyst has to elicit this information from the customers. And of course, it's axiomatic that the customers aren't always sure of what the requirements should be, particularly with respect to quality of service. That is, some kind of measured understanding of how the system is going to deal with these non-functional situations. Second concern of software architecture. Recall, I've been stressing throughout the course that the key element of coming up with a good solution architecturally is how it's going to deal with a non functional requirements, a corollary to that is in coming up with the architecture and choosing the connectors how do those connectors relate to a middleware solutions that we have. Can, for example, define an appropriate middleware technology for dealing with one of the connections we've selected to be included in the architecture. Third software engineering issue has to do with some design questions. Whenever you have a distributed application you have a network. Whenever you have a network you have latency. That is delays in message passing. How is your system going to deal at the software level with this latency. Are there timeouts, are there the implementation of some kind of protocol for resending, and so on. Another key question at the design level as far as distributed applications are concerned is statefulness. You know that the web applications often are stateless. That is, every time you interact with the server, the server has to treat your interaction as a self-contained unit without relying on any variables retaining values. Of course the database sitting there can serve as a persistent but heavyweight way of keeping track of state. The question from the designer then is how they're going to deal with this. One solution is, you've seen probably with respect to cookies, that is the server sends back some of its state to the client, which then returns that information on the next next interaction. And then there's just the general question of concurrency. How is synchronization going to be performed, how can you ensure you don't run into any of the problems like dead like, dead lock or making sure that everyone of the pieces of the system has some kind of interaction on a timely basis that is, that it's live

38 – Research Questions

So more specific questions about building these applications include the following. One kind of question is, how do the clients know what capabilities are available to them? One approach, which is typically used, is naming. That is the client has the IP address or URL of server technology. Okay? And it, it essentially finds that service by providing that name. You can think of this as White Pages in the, in the telephone sense. An alternative to this approach is a client saying, I have a need for service x, and being able to try to find various resources that can find, can provide that service. That's similar to a Yellow Pages lookup. Some Yellow Pages technology have come out, but it hasn't proven quite as successful as maybe we had hoped. Second research question has to do with use of reflection and meta-object protocols. Recall that I mentioned that data can sometimes be self defining, that is the data itself reflects or represents its own structure. The same thing can hold with respect to, to programs, that

is programs knowing what kind of services they provide even knowing how, they deal with non-functional considerations. How many transactions can they provide in a given unit of time and so on. The third category of questions has to do with data representations. For the past 20 years, relational databases have dominated the world, but now there's a recognition that one size does not fit all. The different kinds of applications might require different kinds of organizations for the data. Okay. This movement is sometimes called the NoSQL movement, and there is various commercial, even solutions out there that you can consider in building your applications. Another question is, fat versus thin. Particularly fat versus thin clients. This question has actually been with us for a long time. When we had applets, originally developed by Java, the idea was that the client, that is the web browser, would, download functionality as needed. Be able to try to provide as much functionality close to the user as possible. This is sometimes called a fat client. The other extreme is to say, let's make the client as thin as possible. That is, all it is is really a user interface. Now we've gone back and forth between fat and thin clients. AJAX is a way of being able to reduce the overhead of client to server messaging, in particular client to server when a whole page is being re-written. AJAX is a way of making local changes to web pages without necessarily having full server interaction. Another class of questions has to do with the different kinds of devices that are now parts of distributed systems. Sometimes those devices are relatively constrained by their power consumption and batteries, which might mean that their memories and their chips are going to be smaller or slower. In which case the overall device is going to be somewhat limited that might be a reason to have a thinner client on it. Related to that is the whole question of mobility. If part of the application, if the client side of the application is moving around, what happens if it all suddenly goes in a tunnel, right, and you can't reach it? Your system man has to be more robust with dealing with that kind of uncertainty.

39 – Examples

And now I want to look at a couple of examples of approaches taken to these kind of applications. You can think of these a computing paradigms or, methods for dealing with particular kinds of situations. In particular they both relate to services. One of them is web services and the other is service oriented architecture. For the purposes here we can say that a service is a self contained, self defined unit of computation that's meaningful to the user. We'll go into that a little bit more when we look as these two particular, types of implications.

40 – Service Oriented Quiz

In terms of the definition I just gave, voting application is not service oriented. What change would you have to make to it to convert it into a service oriented type of application.

41 – Service Oriented Quiz Solution

Well the problem is that as described the application provides two services. One is the ability to register votes and the other is to give you some kind of statistics. If we wanted to convert it into a service oriented architectural style then we have to break those into two services. Note that, that might mean we then require the user two interactions when currently there are only one.

42 – Web Services

The first style of application is a web service. You've probably used those. You may or may not realize you've used web services but they're there. And all it really means is that you have some kind of software system, designed to support machine to machine interactions over,

over the web. They often involve particular with APIs or standards that are agreed upon up, by the various parties involved.

43 – Web Services Protocols

Typically, web services really mean that you're conforming to a certain alphabet soup of standards that are out there. On the data side, it means you're using XML and possibly some some other related protocols such as SOAP, RDF, OWL, JSON and so on. On the services side, if you're doing full, full-blown web services. You're probably using WSDL which is the Web Services Description Language. Which gives a way of actually defining the service in such a way that then code generation technology can build some of the, the pieces that you need for managing it. The third element of web services is UDDI which stands for Universal Description Discovery and Integration, and what that really means is "Yellow Pages". Using UDDI you describe your service, it gets posted to some kind of server, then that your potential clients can look up when they need a service, and they form, in fact, provide that service.

44 – J2EE System

Here's a graphic that describes one approach to web services. This one provided by J2EE, which is Oracle's technology for dealing with new situations. Three remaining components on the left is your web browser, which is going to be using HTMLs or applets, or possibly even application. On the right, is some kind of database services, and in between is where the beef is, as far as Oracle is concerned. And that's what holds the business logic. This is the computational segment, that knows about the particular application computations. And the way that J2EE has it, there's two parts. One is a web server, and the other is the EJB containers. EJB is enterprise java beans which is libraries are API within Java. Standard interactions then apply, and presumably you can build your applications using this technology, in a way that can deal with some of the non-functional constraints that these applications have to be constructed.

45 – SOA

Second example approach, Service Oriented Architecture, sometimes abbreviated SOA. Wikipedia calls this a computer system architectural style for creating and using business processes, package to serves throughout their life cycle.

46 – SOA Services

What this means as far as the actual delivery of services is once again you have self contained, self defined modular applications. You can think of these as more or less vertical slices through the system. That each service is capable of self contained that it has all that it needs in order to provide that functionality to the user. There's a definition for it, so that the remainder of the system and the architecture can have a configuration that it understands. And it's modular, okay? It obeys certain rules about interactions. In particular, there's a use protocol use at the level of the software architecture protocol describing, how the pieces work together. The overall intent is to try to directly relate, the business needs of the user, to functionality that's being provided. And typically this means that as a developer, an architect, you're coming up with a suite of sub-services, that together can be composed into. These user facing services. You publish them, you located them in particular places and you can dynamically invoke them.

47 – Characteristics of Services

Some of the characteristics of services are flexibility. That is, you are essentially refactoring, if, if you have an existing application, you want to make that service oriented you factored into, into pieces that then you could combine in interesting ways. Meaningful here on the slide means each of the services is something that's meaningful to the end user. They're stateless, for the reasons described with web services. Stateless means that typically the code is actually simpler than it would be if you have to remember variables. And also it's transparent with respect to middleware. Typically, there's a middleware technology solution that takes your descriptions. Takes your configuration information and generates the the, the middle work pieces that need to be generated.

48 – SOA Rearchitecting

Service Oriented Architecture is, is often of interest at the enterprise level. That is, big corporations or organizations who have a suite of applications that they want to adapt to internet type situations and rather than rewriting the whole suite from scratch, they want to re-architect it into a service oriented configuration. This can be tricky, as you might imagine. Any time you're dealing with large amounts of code and rearchitecting them. You can run into issues of, of reliability. And there's many a story of a lot of wasted money leading to systems that don't work as expected. So, in, in doing such rearchitecting, ultimately you have to take code that maybe was used to running on a mainframe in which a code was controlling it's own fate. It was in charge control-wise for calling the various pieces. Have a service oriented type architecture over the internet, and it's intended to be providing user functions, then it may have to be switched in to a reactive type system. That is, responding to the user interactions. This is a major change to any system to do this. The point I'm making here is that there's a cost in doing this and there's a high degree of risk because of that cost.

49 – Summary

To kind of tie all this together, Middleware is ultimately a collection of technologies for addressing non-functional constraints in these heterogeneous distributed applications. These technologies include application programmer interfaces, protocols tools, and even design patterns. As the Internet grows, as our, our networking and our number of users and the various applications, the need for standard Middleware solutions is going to become all that more important

P3L10 Guest Interview: LayerBlox

01 – Introduction

Yes, so I started at Georgia Tech and ended up getting my PhD there in computer science. Did work in software architecture, software engineering. User interface design and kind of the intersection of those, those three. After I got my PhD I was a faculty member at Michigan State University for about 12 years. Did research in, in formal modeling of software architecture. For the past I guess about six years, I've been working at a local company called Logic Blocks. That specializes in smart databases, and uses a lot of the modeling ideas and formal methods ideas that I've been working on in, in my research, in, in practice. So it's been very interesting.

02 – LogicBlox

What kind of company is LogicBlox, what sorts of things they sell? So LogicBlox's primary product is what we call a smart database. It's an active cloud database that has much of the business logic that one would normally need to write in a traditional imperative language running on separate machines down in the database. So that when data are added to the database, these business rules kick in and update views automatically. It specializes in doing really large-scale analytics and applications that are kind of a hybrid between transactional and analytic, and that's really our sweet spot. And what sort of customers do you have and the kinds of products that they need? Yeah, so most of our customers that are longer-term customers have been customers in the retail domain, that need to do analysis to decide how many products they should order for a given promotion, or forecast demand for a given set of products at their stores, or do financial planning. These tend to be really large, often multi-terabyte applications that have years and years of sales history that we use to try to do a good job of forecasting or predicting how products will move. And we use our smart database to analyze that and roll up and roll down those data very efficiently. So the terabytes is not the size of the source code but it's the amount of data that they have to deal with. That's correct. Yeah, the amount of source code is meant to be very small. The idea is that you want to declare your business rules as concisely and compactly and as close to the way you'd want to just say them in English is you can. And then we install those business rules into our database, which then we load with the terabytes of data. And you mentioned in the cloud, is this what they call softwares and service? It is, yes, so all of our applications are sold as a service. So our forecaster, which I think we're going to talk about in a little while. In particular, our forecast manager, is a service that big retail customers will pay us a monthly fee to access. Part of our model is that they can at any time choose to. They don't have to install a lot of special hardware and have IT expertise in house, they can just subscribe to our service. They access it through standard web browsers. They can turn it off and on as they desire. If they decide that they don't like it anymore, they can just quit. And so it's really nice for customers. Part of our business proposition is that we can quickly get into, give access to a customer and let them see the benefits. They don't have to make a lot of upfront commitment to use our technology

and they can decide to leave anytime. So would you call this a real time application? I'm not sure I'd call it a real time application. It is an interactive application in the sense that there will be users who will be executing queries and analyzing data in what they would consider to be real time so they need fast response time. It's an interactive system. It doesn't have hard real time deadlines, the way we normally think about real time stuff being developed. And how does the customer's data, their sales data or their product data get to you? Yeah, it's actually a little bit complicated. So, customers tend to keep their data in database systems that they've had for 50 years and have their own special format and they're very difficult to integrate with. What we tend to do is get them to export the data that they need to send to us and usually in just big CSV files or some kind of delimited tabular file. And they'll send us these files at various times. We have servers that will receive the files and put them off to Amazon's S3 storage, it's a nice, very scalable, very reliable storage system that we use to keep the incoming data and from that we load them into our databases. But, this data might be periodically updated. That's correct. Okay. Often, on a daily basis, and sometimes, for some customers, even more frequent than that.

03 – Role

Can you tell me a little about your role at LogicBox? So I'm a Chief Application Architect. And what this means is that I oversee all of our app development and pretty central in coming up with the initial design for any kind of new products that we develop. I often do a fair amount of prototyping, data modeling, early in a new product that we're developing for a new client. I'll sometimes, often actually, interact with customers to make sure we're clear on requirements when we start. And I'll often make a lot of the initial programing decisions and then start to farm it out to a group of developers. And I'm guessing you also have a strategic role as far as how a company is going to be developing applications on into the future That's right. So, in addition to Chief Application Architect I'm also Executive Vice President. So, as an architect, what kinds of decisions do you have to deal with in terms of structuring this code base. So typically, there are kind of two scenarios where I get involved. So there are new configurations of existing services that we've implemented before and there most of my work is just in requirements analysis to understand differences between clients. And helping to put together an application development team to go extend that configuration to the new client. That's kind of the normal development role, and I do that for several products in our family that we've done many times before. We've deployed these services many times before for previous clients and it's really just a matter of kind of bringing up a team of developers and just showing them what the architecture is and telling them how to flesh it out. For new products, and as of not too long ago, the forecast manager that we're going to talk about later was a new product. For that, I actually had to do a lot of the initial development of it, and we did things like initial scalability analysis. There's often some pretty complex calculations that need to be performed. Under not under real-time, but under time constraints that a user would need to deem acceptable. So I do a lot of initial scalability analysis, which involves coding up some solution approaches. Making sure that we understand how to do the integration with all the various data that come from the customer and also just doing data modeling. Because, a lot of the times we find when we do a new application, there's a significant data modeling exercise that needs to get done before we can really start making progress. And how about the software process itself, the development activities in particularly, strategically, as far as improving productivity in the long run. Yeah, I also play a role in that as you might imagine. We use, we're an agile shop, and so we tend to like really short iterations where we have demos at the end of those iterations. We use your traditional Jeera and other kinds of technologies

for issue tracking. But yeah, our process mainly is one of when we have a new project, do an initial architectural assessment which may take a month or two, to do the kind of prototyping and scalability analysis that we need. And then we proceed when we start bringing developers on, usually with a small number of developers it scales up over time. Rather than with a big group that starts at the beginning. And we use a very iterative usually one to two week cycle iterations. So this is a grunt process.

04 – Typical Application

Can you give us an idea of a typical application, and in particular, how big it is, how long it takes to develop? Sure. So there are two or three, as I said, different kinds of applications that we tend to develop. I think one, that is a good representative is the Forecast Manager. So a forecast manager, is an application that will forecast customer demand for products at various locations, various stores, of a retailer based on a historical analysis of how those products have sold at those stores over the past two to three years. It takes into account special events like promotions that are enforce and it's actually a pretty complicated set of calculations. It involves some regression analysis and it's pretty involved. So, we generally, when we have a customer, what we'll do is, every week or maybe even every day, we will generate new forecasts for them in a big nightly batch. Nightly or weekly batch. And we'll calculate the next 52 weeks of forecast for all their products at all their stores. After which, most retailers have a group of employees who go through and review those forecasts. And if they find things don't look like they have imagined, because it's forecasting, it's not, you know, it's not an exact science. They'll want to get in and do some tweaking, make some adjustments to them, maybe they think that some estimates of increased demand are too high or too low, and they'll actually want to make some changes. So forecast manager application's responsible for generating all these forecasts allowing forecasters, the employees at a retail organization to go through and drill up and drill down, and try to understand these forecasts, and also to allow them to make changes to the forecast. Tweak them, what if them, et cetera. And they can use they forecasts to predict sales and do inventory? Correct. Sorts of things. It's central to they're planning. It's very similar to their plan, yeah. It's typically hooked up with replenishment systems as well. So once they approve the forecasts that we've suggested for them, they'll then send those forecasts down to a replenishment system which will actually make orders and cause goods to be put on trucks or on ships or whatever. And I imagine with all this data that there are kind of heavyweight calculations, functional requirements that the forecast manager has to satisfy. That's correct, that's correct. And what's interesting about it, and what I found particular in working in the retail domain was surprising, let's put it that way. There are, every different client has slightly different business requirements that make it very difficult to design a completely generic product. So there's always a bit of client specific business rules that you need to take into account and configuration. It's not like a shrink wrap product at all. So, it really is when you want to sell this to a new clien, even if it's a product very similar to one you've released before. You have to do a fair amount of, you have to staff it up with a development team, and you know, do or extend an existing requirements analysis. And do some custom protection. And so when you say product before you were talking about software as a service, the products are really services. That's correct, Yes. Okay,and what I'm hearing is that you don't want to have to do a separate forecaster for every one of your customers, but there are individual pieces. Each customer has their individual needs, and so you need some solution for them specifically, but you can't have a general one for everybody. That's correct, that's correct. That's one of the core parts of the problem that makes it hard. Okay, now how about on the non-functional side? What sort of interesting non functional characteristics

do these applications have? Yeah, you can break them into a couple of categories. One is, you asked earlier about how we get data and we load it, so that implies that there's a batch operation that has to load all this data that come in, into our database. The batches tend to have pretty tight timing requirements. So, for example, when we do forecasting at the end of a week for next week's orders, we typically get the revised sales data that we use in predicting, we'll get that sometime late on a Friday evening. We may get any number of other kind of data from the customer, as well, that we need to use in prediction. We'll have to have the forecast all generated and ready for inspection and modification for the forecasters when they arrive by 8 AM Monday morning. So there's a batch window and that batch window is a very hard deadline. If we can't do all the work within that deadline, we have a service level agreement with customers because we sell this as a service, and if we don't meet our batch window, then we're in violation of that agreement, that has financial implications, et cetera. So batch requirements are big. We also have a nightly batch which has a much tighter window and doesn't consume as much data. So in the weekend situation, is that enough time for you to deal with things? How much of that weekend is consumed with actually computing? So that's a good question. It depends largely by client. We, for our bigger clients, we have one that I think their app is now, it's over 11 terabytes of data that we have to maintain and let them use. And what we do is, we actually partition the database. Over many different nodes, many different nodes on the cloud, so that we can meet this batch window. And what we find when we're scaling these up is, we have to do some experiments to decide how big, how fat can we make one database in order to meet our batch window to load all, not just to do the calculations, just to load the data to begin with, because it takes a while to load 6 terabytes of data. So we have to, we have to scale ourselves and partition it so that we can load all the data, perform all the calculations within the batch window. And there's always a trade off between, the more that you can make, you can decrease your batch time, if you throw more machines at it, but then that increases your cost, our company's cost which we have to eat. So we're always trying to optimize that sweet spot. So what I'm hearing is that you're making use of a large number of fairly heavyweight server nodes on the cloud that are cranking pretty much full time over the weekend- That's correct. To make sure you hit your deadline for Monday morning. That's correct, that's correct. And one of things that's interesting about deploying as a service, has everything to do with that problem. So traditionally, when a retailer would want a forecasting application, that retailer would have to buy and maintain all the machinery he or she might need in order to generate those forecasts even though you're only going to do it every week, every weekend sorry. So, what typically happens with the forecasting application is, it's a of heavyweight work on the weekend, but then during the week, not so much. And so our solution scales, so that we actually scale up over the weekend in order to do all this data loading and calculation. But then we scale back down to a smaller footprint after all those calculations. So you're taking advantage of the fact that you can get elastic computing resources Elasticity, exactly. Off the cloud and only pay for what you need. Precisely. So, I interrupted you before when you were talking about the nonfunctional requirements, what else is there? So batch requirements is the one category. The other category are interactive requirements. So, users are going to, as I mentioned when, after we generate all these forecasts you're going to want to go through and inspect them and analyze them, and drill up, and drill down, and make analyses, and maybe even modifications. So, that's a large set of data that they need to operate over. And anytime you have users doing interactive queries over large data sets, you have the potential that queries and operations could take a very long time. So in addition to our batch window, we also have to design and partition accordingly so that queries will be, user operations will be performed in acceptable times. And I'm assuming that

you've kind of mentioned that reliability is really important. Hm-mm Okay this, you have to get the thing done. Yeah. How about accuracy of these computations, how do you deal with that? Yeah, so usually through a separate process, so Logicblocks has a dedicated data science team that works with big retailers who are very familiar with their own data to do accuracy analysis. And we tend to do this by running our forecasting offline, and comparing on old data, and then comparing it with actual sales data, so there s some metrics we used called mean. Mean percent error which are used to decide whether your forecasts are sufficiently accurate. If they're within a certain percent error of the actual sales. And so, we tend to do that offline, and then if there are any changes to the algorithms, the science algorithms, we roll those into the service so that it'll be available next week when we do the next analysis. So some real significant, non functional requirements, in order to make all this work in a fashion that can satisfy your customers. Right. Okay, how about the architecture itself, let's say of the Forecast Manager, what makes this particular application architecturally interesting? Okay, so I think from an architecture standpoint, what makes this architecture most interesting is the distribution aspect. The services oriented design of the whole thing. And how each of the distributed nodes talk to one another. So I have a diagram we can take a quick look at. This is a multi tier diagram. I'm not showing the top tier, because the top tier is basically a web browser, right? So all of our clients, they're top tier, presentation tier, would be there web browser. They're going to be running HTML 5, JavaScript applications, running in their browsers that are periodically going to make service calls to a uniform location on the cloud. Those requests, those http service requests will all go through this front end node that I'm showing here in this diagram. We consider that the middle tier, all right? So the front end of our application is a middle tier that has a couple of databases running on it, which I'm showing there. One of them is largely the one called UI0. It is hosting services which do things like either directly answer queries or proxy them down to other nodes that actually contain a slice of the real data. And so that data tier level that I'm showing in the diagram, actually has multiple. I'm abstracting it by just showing 2, but there could be up to 50, or 60, or 100 of these things depending on the size of the data. Each one of them is the same in terms of what code they're running, but they have a different partition of the data that they're responsible for, and each one of them is running a web server that's hosting services that answer queries, all right, or do updates. So typically in this architecture, you don't see this much when you're running the app from your browser, but you'll click on a link to go drill down into, you know, the results of some query that will issue a request that goes to this front end node, which may then delegate to one or more of those data tier nodes to gather some data, perform some updates, put it back together and then get your response. And all this has to work very quickly. So, as far as architectural style, would you say this is a three tier architecture? Yes, yes. And, complicated by the fact that you've partitioned your data across many, may networks. Correct, correct.

05 – Motivation for LayerBlox

Well this sounds like it's a very complex collection of technologies that you have. But I understand that you've developed a novel approach to architecting these applications called layer blocks. Could you tell us a little bit about what motivated you to do this? Sure, so thus far, we've talked largely about kind of how the high-level deployment architecture of the system works and what makes it interesting. But I've been pretty abstract with respect to the code itself. And what gets loaded into those databases. You may recall at the beginning I was saying that one thing that makes our company a little different and our technologies stacked a little different is that we actually install code into the databases, and you can think

of those databases as being active rather than passive. Their actively calculating and coming up with results rather than just being dumb date stores. So that code that we write and install on those databases itself has some interesting software engineering challenges because we implement all of the business rules in our applications, in that code, and then install it. And one of the things we've found is that particularly with applications, like forecast managers and most of the other predictive applications that we're developing, is that we are often are not developing just a single product, but in fact, we are often developing something more akin to a product line. That is we find that there are many slightly different variances of use of the same algorithm like forecasting. But they need to be deployed in slightly different ways for slightly different contexts. And so, what motivated Layer Blocks was, after we had been doing some development on the first Forecast Manager application, we noticed that a lot of the work we were doing was kind of developing code that really seemed redundant. That we're developing the same kinds of algorithms multiple times for these different uses. And we wanted to find a way to unify that, so we could do the development once, very efficiently, and manage this growing list of variants of the same component or functionality. So this is getting back to the challenge you mentioned at the start about that each customer has their own specific needs, okay. But that you'd like, to the extent possible, to have a common solution to reduce your costs of dealing with the customers. Exactly. Yeah, you don't want to have to develop a system, a component from scratch for each customer. If you've got 100 customers, you don't want to have to develop 100 components. And what you want to do is you want to develop one component very nicely so that it can be configured and packaged in different ways very easily. And that's what we developed LayerWise for.

06 – LayerBlox

Well, can you give us a 4,000 foot view of what LayerBlox is? So, LayerBlox is a software generator for generating different variants of products in the same product line. So you, you want to be able to generate all of these, these variants. how, how how does your generator actually work? What does it take as input, and, and and, and, and how does it process that? That's a good question. So I should say first off that our generator is based on a pretty well understood idea from software engineering. That goes back to, really goes go back to the, the early 90s on product line generation. And so, each variant that we want to generate is a different program in the same product line. And we, we organize our product lines in terms of re-useable features that we put into a library. We've designed them according to a design idiom that, that, that makes them very composable with one another, which I'll demonstrate here in a little while. And what you do, then, to, to generate a variant is you write something called an assembly specification. Assembly specification explains how you put these features together in some novel combination to generate a particular variant. And it has some particular useful properties, which I think we'll be able to, to dig into by example here in a few minutes. But what's really nice about it is the features you can write once and reuse many times. And you can very easily understand by virtue of comparing these assembly specifications how two different variants in the same product line are common and how they differ and be very precise about that. And I'm curious about the, the title, LayerBlox. Is it, is this related to layered architectures? It is, it is. When you see some examples of assembly specifications, you'll see that the, the components that we're generating, each component, when I use the term variant, I mean program or component. So the when you see how, how a given component is generated, you'll notice that it's, the little program you write to say how to generate it, the recipe looks, is, is a very layered, very hierarchical form. It's related to layered architectures in another way too, in the sense that typically with layer architectures, you tend to think of of software built

in stacks. Where you can understand a layer you can understand one layer, just in terms of the interface that it exports, without any knowledge of how it's implemented or of the layers that are underneath it. the, the assembly specifications that we write using LayerBlox have that same property. And, and just to, to clarify a bit. The layers that you're talking about are, and the generated code you're talking about, they go in that middle tier? No, they actually, in this case they could go in the middle tier. But in this case they go in, down in the data tier. Okay, so the, the tiering is kind of independent of the, of the layering? That's correct. That's correct. And in the diagram, there's also this reusable feature library. Can you say a word about that? So, you know, I mentioned earlier that when we have a number of different clients, their programs and their applications are very similar, but they're not exactly the same. What we found is that if we do a decomposition and design of our software by feature and I know you guys have spoken of feature diagrams and feature modeling in the past. When you, when you when you do a feature-based design, you actually can get reusable pieces of, they're not whole applications. They're little fragments, but the, but they're highly reusable and composable in ways that you can put them together to make different variants of a, of, of the same application very easily. So we did a, in the example we'll see we basically did a feature analysis to understand what are the different features that are put together to do forecasting. And based on that feature analysis, we we designed our reusable features around it and, and got this ability to, to, to compose them in this very, very nice way. So the, the unit of variation is kind of, a customer-visible feature? It may not be customer-visible, I, and, and ideally it, it could be, right? And in other product line work it is the unit of visib, of the unit is customer-visible feature. In our case, it's more implementation centric. But, but still, it's, it's much more on the science side than, then the customer side so in our case, we, we're doing forecasting. And there are some pretty com, complicated algorithmics that go with, with forecasting. So that domain is the domain at which we've we've done the future analysis.

07 – Assembly Spec

Can you tell us a little bit about what one of these forecaster's is? Sure, so at a very high level, forecasting is a pretty simple problem. What it basically is about. We do some analysis to calculate demand, to forecast demand of products in the absence of any kind of promotional activity or any other kinds of events, special events that might cause spikes or troughs in demand. And that generates something called a baseline forecast. So we have some algorithms that are responsible for generating baseline forecasts by doing an historical analysis of sales data. We then have a separate set of algorithms that calculate what we call incremental sales. This is an additional uplift or it could also be negative uplift from the base line when certain special events are in play, like a promotion, a mother's day promotion that's going to cause an uptake in the sale of flowers and gift cards, right? Christmas is going to, yeah, or any other kind of holiday is going to have and uptake in sales. So, we tend to think of a forecaster. The thing that actually calculates forecasts as being decomposed into a baseline forecaster and an incremental forecaster. And each of those things can be parameterized. And then put together to form the actual forecaster that you want to use to generate to calculate demand for products in stores. So that's a forecaster kind of in the abstract. It gives you some idea of what some of the features in it are and how they might compose. And can you give us a, a picture of what this forecaster is like in terms of its assembly spec. So, if you look at this slide, you'll see a very simple example. Probably the simplest example. I call it the Hello World! Of forecasters. And you'll see it just has four lines. Right. So, those four lines explain how to generate four different components that are used with one another. In this case, we've got a really simple one called bForecast, for baseline Forecast. One called mults

for multipliers. These are the multipliers that are used to calculate the incremental uplift when there are different promotions and such in place. We use this two components, bForecast and mults to generate an incremental forecast, where they calculate the actual incremental forecast. And then, we put the incremental forecaster and the baseline forecaster together into a component called batch, which is a batch forecaster. And so what that does, and we can go into in a moment exactly how it does it. But as you can see, we start with some kind of elementary components, the baseline forecaster and the multipliers. And then, we compose those with some reusable features incre for incremental, and fcst for forecaster. We compose those together to make an actual component called batch. And batch is the component that's kind of the output of this thing. That you would then install in your database to calculate your forecast. So is this the description of one variant? This is the description of one variant. That's correct.

08 – Components

Yeah, so let's look at this example and the different parts of it in some detail. So what I'm showing here in green are the four different components that were generating as a virtue of this assembly specification. And as I mentioned a minute ago, the only one that's the, the, the variant, the one that were interested in is batch. But the other three, bFsct, mults and iFsct are all kind of sub components that are used to, to, to create batch. You can think of a component as a little program. It's clear that batch is a little program. It's a little forecaster but each of the other ones is a little program too. iForecast is an incremental forecaster. Mults is a set of multipliers and bFcst is a baseline forecaster.

09 – Interfaces

So, now what we're seeing are what I call interfaces. Each of the components that I just mentioned, exports some interface, which explains to the client of that component, or the outside world, how to interact with it, what capabilities it provides. In our case, those capabilities are what we call predicates. You can think of them as tables in a database. These are predicates, or tables, that are accessible and visible to the outside world that clients could query or update. And as you see here, even though we have four different kinds of components, we only have two different kinds of interfaces forecast and multipliers, and this is by design. What that means is that I can, as you see in this example, I can have multiple components that implement the same interface. Now what's nice about that is components that implement the same interface are plug compatible, and layered assemblies. So I can actually build different assemblies that use each of these components, each of these three forecast components that I've just declared. Put them together in different ways and build a nice, rich library of variance, just by putting those forecasts, just by putting those together in different ways. And can you relate this use of the term interface to its use in other languages like Java? Yes it's almost exactly the same use. So in Java interfaces you declare signatures of methods that many different objects would, obviously many different classes could implement. It's essentially the same thing here.

10 – Refinements

So, the next thing I'm showing on the slide are two of the refinements that are being used in this assembly specification. Refinements are little program generators. They take existing programs of some type that implements some predetermined interfaces. And, they compose those programs to generate a new program, or new variant, that implements some other interface. So, for example, we have here a refinement called Incor for incremental forecaster. And, Incor is parametrized by two different kinds of arguments, one is some component that

implements forecast interface. The other is some component that implements the Multipliers interface. And, when I apply this refinement to components of those suitable types, then I'll actually generate a new component that implements the forecast interface. Likewise, for the FCST refinement, it's parameterized by two components that implement the forecast interface, and it generates a new component that implements forecast interface. So, I'd like to relate this term as well. Earlier In this class, we've talked about using refinement to go from an abstract description of something to a more concrete description. It sounds like your refinements here are doing that with some kind of generator. Exactly, yeah so, but it's meant to appeal to that same sensibility, right? You can think of an interface as representing an abstract program that fills in some details. In particular, in our case, the signatures of the different tables that are going to maintain calculations that we want to make, like forecasts. But,hey may be implemented in many, many, many different ways. And so, each refinement Is a generator that will generate a different way of implementing that interface. It really is meant to appeal the same idea.

11 – Variants

So I can see that you have some mechanism here and in the specification file and the generators for generating variance. Can you give me an idea of kind of variance that might apply in this forecasting situation? Sure. So we already showed an example of the most common case, which is what we call a batch forecaster, one that generates forecast for all the locations and products that pertain to a given retailer. Some other variants on that are we might want to generate forecasts not in batch, right, not down in the database, where they're materialized and kept. We might want to generate them on demand. That is, we may just want to say, all right, well rather than having to wait for the weekly or the nightly reforecast for everything, I might want to quickly regenerate a forecast for some particular product, or some particular set of products. And I don't want to store that result in the database, I just want to get an answer. We call that on demand forecasting. Is this something that somebody, at one of your clients might formulate as a query? Yes, yes. In fact, in several of our clients who use this service, they have existing systems where they need to be able to get an on-demand forecast. Because they had traditionally integrated with other systems that did on-demand forecasting. So we have to package up a service for just that, and that service has to involve all of the same algorithms that we use to do a batch forecast, but it's deployed slightly differently. So that's an example, batch versus on-demand. Another example is, as I may have alluded to previously, users, forecasters, that is, the actual employees at one of the retail clients will often want to make adjustments or do some what-if analysis by tweaking the inputs to our forecasting algorithms to try to see if they can get forecasts that look more in line with what they've seen historically. That's not used a lot, but it is something that is needed, so the ability to do variance that will allow for user adjustments and for tweaking of inputs is another common class of variant. I think in this particular, I should say, in the first application we built, the first forecast manager application, we built at least eight different variants of the forecaster. And I may be missing a few. We may have made even some different ones that we've used in order to test that capability and do the accuracy assessment. So clearly you get a lot of different variance. I mean eight's a lot. Right? You wouldn't want to implement eight of these from scratch. You'd need to do something to manage that variation. And can you show us an assembly spec for one of these variance might look like? So if you look at the slide that is being depicted now, this is the Hello World assembly spec that we were looking at just a moment ago. And this is a new one, a variant that I'm calling Guten Tag Welt. It's a variant of Hello World that allows for forecast adjustment. And if you look at this assembly

spec, you'll notice a couple of things. So first, it's not exactly the same as Hello World. It's got some new components, a new interface, and a new refinement. In particular, there's the mods component that implements a new interface called overrides that represent some of the adjustment overrides that a user of the system might want to make if they're tweaking the inputs to the forecast in order to see the effects. There's also a new refinement called dampener, which allows multipliers to be dampened by these overrides. So typically in these applications, one common adjustment scenario is to go in and allow the users to change, by some percentage, the multipliers that our forecaster calculated to compute uplift so that they can see its impact on the final forecasts. And so what that dampener refinement there is doing is it's applying those overrides by virtue of being parameterized by that mods component to multipliers to give us another implementation, another component that implements the multipliers interface. And we can then use that in the same two refinements that we saw previously in Hello World increment forecast. And the output of this thing is the last line. That's right. You have a batch, but it's an adjusted batch. That's right. Application that you built. Exactly. Exactly. And it's a different component, so we could actually put both of these in the, install both of these in our database if we wanted to, and then we would have a batch forecaster and batch adjusted forecaster if we so desire.

12 – Product Lines

I can see that this might be useful in a situation where you're dealing with a potential customer, and they want to do some some forecasting, but their needs aren't exactly what your existing system is. You want to get some idea of how much work it's going to be to produce a variant for them. Can we use these specifications to determine that? Yeah, indeed you can. And in fact, that exact problem is one of the things that motivated us to invest in this approach. So, if you look at the slide, you'll see Hello World and Guten Tag Welt, those two assembly specifications. And you can notice a couple of things. First off, they have a lot in common. So, Guten Tag Welt reuses two of Hello World's components, the bFcst And mults components and it also uses both of its refinements incr and fcst. Where they differ is also very clear to see. So the Hello World has this iFcst and this batch component where as Guten Tag Welt has the diFcst and batchAdj component. And Guten Tag Welt has this mods and overrides. Component interface respectively. So you can see from this just by comparing exactly what two variants have in common and exactly how they differ. Okay, Kurt can you tell us something about now that you've had some experience with using layer blocks over the last year or so about the benefits you've seen from its use? So, there are several benefits that we can get from it; the most obvious one is that we get a lot of code reviews. I can try to quantify that a little bit. When we had multiple different implementations of these variants and we replaced them, which is how we originally brought layer blocks into being. We replaced them with with a Layer layer blocks based solution, we've cut our code footprint by about five times. So, it was a substantial amount of code reduction that we got in [CROSSTALK]. You mean 80% in code base? Yeah, that's right. Fantastic. 80% reduction in code base. In addition, the virtue of it is that it's designed to do programming by generating variance, rather than writing one off programs. And so that approach, and that way of doing engineering actually helps the scale much better to building a large complex software. It just adds a discipline on it. Now when developers are working on this code base, instead of thinking about how to go write the new method or make this new tweak, they think, could I or should I make this a different refinement of an existing interface or different feature. So you get a lot of simplicity in a design by virtue of that. The scalability of the whole approach really owes to this idea of feature refinements being so reusable. As you saw in the examples, and there are many more,

each of those little generators gets reused quite a bit. So you don't need that many of them, and you can build a pretty interesting and rich library of variance. In addition, features, if you design them by doing a feature analysis are very robust abstractions in a domain, and so they tend to give you the kind of reuse that we get. So if you couple a good feature analysis with applying an approach like this, you're going to have some good idea that you're going to be able to get this kind of reuse in a project. And then finally, I think maybe the last big benefit is, it's really beneficial for understanding and for training. So if a new developer has to come onto a project, and they've never see a forecaster before. But they have an assembly spec which is something that's pretty small, just a few lines. And particularly if they see two or three of them for different variants. They can very quickly get an idea of how to do, a sort of, top down step wise refinement of a variant as a sequence of refinements of a very common easy to understand program. That turns out to have really nice understanding and training benefits. But historically, in software engineering when a code generation technology or a very general solution to things is proposed, there's often a performance hit in the generated code that you have to trade off. Has that been your experience here? So, ironically, it's actually been the opposite. Let me try to explain why. So we are doing a lot of generation, you're correct. But in fact, because we're generating variants that contain really only what's needed. They contain only those features and only those capabilities that are actually needed for a given variant, and they don't have any extra cruft with them. They have been running more efficiently. In fact, when we replaced these hand written forecasters with ones we generated from this library, we saw pretty dramatic performance improvements just by reducing the amount of business logic that we had to install into our database, and which of course then had to always be running and calculating. So in that case, we've actually seen the opposite. So win win. Win win, yeah.

13 – Possible Limitations

Well this sounds great. Are there any down sides, or where are you as far as implementing this in your company? So, we haven't implemented it everywhere yet. We're using it right now almost exclusively for generating forecasters. Although we've started using it in some newer applications for similar variance. We're not generating whole applications with it yet, we're generating mostly components of applications. So there's a lot of opportunity there to expand its use and we're looking into that. Also, I think I mentioned that this whole approach is based on a lot of prior work. In particular, it's based on the work on these layered assemblies, the GenVoca approach, the Head approach, all the work of David Parnas, Don Batory. That work is quite rich, and so we've been able to borrow heavily from it to get the LayerBlox to where it is right now. But that prior work also has gone on beyond what we've done. So there are some pretty nice things you can do in some of Batory's latest work where you can actually think about generating product lines of product families. Which, if you can imagine, would scale much larger to building whole applications. We don't have any support for that yet in LayerBlox, but that's something we're looking into. So let me get a little clarification here. You used the term product line and product families. Can you differentiate those two? Probably the best is to give an example. So what we've looked at so far is how to use LayerBlox to generate different forecasters, different variants of a forecaster. But a forecaster is one kind of product. Often what you find, and what we have to deliver to customers, is a family of related projects. Such as a forecaster that works together with a replenishment system that works together with a promotion planning system. And what you really would like to do to scale this up is to be able to generate product lines of entire product families. And that work is, there is existing research in that area. But we have not incorporated that yet into LayerBlox. So, there's this family of compilers called GCC that are capable of targeting various platforms. But also they

can be used for different, the technology can be used for different programming languages. Is that related to this product lines and product families? Interesting question. Yeah, I guess you could say that. I wouldn't think of a compiler as a product family. But to the extent that it is not for a single language but for multiple languages, like GCC can be used for FORTRAN and for Ada. Right. Yeah, I think that's probably a good way of thinking about it. Anything else you'd like to say about the current status of LayerBlox or the things you'd like to do in the future? We're interested in integrating LayerBlox itself in this method of composition more tightly into the programming language that we use to build our applications. Right now LayerBlox is a separate generator. But historically, this kind of approach has almost always started that way, started with a separate generator. But then led into very tight integration into a programming language. And I think that's probably where this belongs. That will then let us do much more sophisticated analyses of type correctness of these assemblies. It'll help us make some better decisions. You mentioned the generality versus performance problem, which could at some point crop up. I have no doubt that, eventually, it might. By integrating this more tightly into the programming language, you get more opportunities for having more context when you're doing generation. Is this anything like the generic capability in Java? Where you can generate collection classes of various types by parameterizing them? It's very similar, yeah, so in fact that tends to be that kind of method. The use of templates that are supported in the programming language is often ways that this general model gets tightly integrated into a language. So there was some work, Janusz Mardok just did some work on integrating this very approach into Java, in exactly the way you're just suggesting. So yes. And we will put on the class resources page some links to some of the papers that Kurt is referring to in case you want to look into them a little more deeply.

14 – More on LayerBlox

Sure. So once we built the code generator, we built a number of supporting tools that were useful in just understanding these assemblies and communicating them to to others. One of them is a graphical visualization tool that'll let you, let you look at in graph form in a, in a dot form. One of your assembly specs so that you can see exactly what the dependencies are, how the different refinements compose with one another. That's been really useful particularly as we were developing our [UNKNOWN] to begin with, because we've found that over time we could compare these, these different visualizations and we could see them getting simpler and simpler. So that was very useful. We also did we implemented some code metrics, so that we could track how large or small our refinements were. We, we hypothesized and this actually born out to be true that over time and after a lot of use, big refinements will break down into compositions of smaller ones and in fact that's happened. And we use code metrics now to, to help us find candidates that we should, should go kind of proactively dive into. So yeah, there are a number of little tools and supports like that, that are pretty useful.

15 – Implications Advice

So Layer Blocks sounds like a powerful technology. Can you reflect any on what change it's meant to people as they're confronted with developing a new application? Do they think about the architecting of it now any differently than they did in the past? Certainly, when we build new instances of the forecaster. It changes dramatically how we think about configuring that, because now there's a lot more structure in place based on the existence of Layer Blocks itself and these assembly specifications to allow us to plan for a new configuration of this capability. And to divvy up the parts when executing it. In terms of new capabilities, I think it's been a mixed bag in the sense of I think that people often find it very difficult to come up with the

right abstractions, initially. So, even with Layer Blocks and with this approach kind of in the bag of tools, we often don't immediately think, when we have a new problem to solve, that it's going to be a product line. Or that's this is the right way to solve it. And I don't know what to do about that. Perhaps if we got better at, kind of, a priori feature analysis, it would be more obvious. Or perhaps maybe, you have to build one or two products in a product line to recognize that you have a product line there to begin with. I'm not really sure, I think it's still kind of an open question. But that's been our experience to date. The other thing I'd like to ask you is, for the benefit of the students who might want to become software architects. Do you have any reflection on the relationship between, kind of, the academic knowledge that you learn about software design, software architecture and what you have to confront in the real world with doing software architecture? So, let me think of a few things that. It's all over the map. I've found that, in the last seven years, I've used just about every thing that I've learned in a software engineering course that, at the time, seemed very abstract and perhaps I thought I'd never use this. I do. But let me try to be concrete about things I've found that are definitely useful skills. So, maybe the biggest one for me is, well two. One is data modeling, be it data modelling using ER, or ORM, or UML, or whatever. Getting really good at modeling at a conceptual level the structure of data in a system. If you're going to build any kind of system like the ones I'm just mentioning, that is a really critical skill. And even though it's taught pretty well in database courses and a lot of people take database courses, you'd be amazed at just how rare it is to be really good at that. So, getting facility and data modeling and not applying it just to database design, but to any kind of information design problem at all, I've found it extremely useful and to really pay dividends. I've also found it really useful to get really comfortable with many different models of software composition, because you just never know when one might be useful and they often are. So, when I was a student at Georgia Tech, I spent a lot of time learning how to use process algebras and CSP, FSP, LOTOS. There's a whole family of these action languages that were very interesting to me. And we covered them in classes, and we learned how to use them. And they have very nice compositional capabilities that were very elegant and very clean and were very well-studied. But it wasn't really clear how to use them in something. There was not a compiler that you would write programs in this language to use to build a big piece of your system. But over time, I've found that they come back a lot. In fact, we've started using them in LogicBlox as the basis for building a batch automation framework where you've got this problem of designing work flows. And you have to compose them, and you need to compose them using a small number of very clearly defined operators. And doing it cleanly and being able to reason about it. So, things like that have been very useful. Petri nets, which I studied many, many years ago, and also seemed like a nice, elegant way to think about concurrency, and data flows, and work flows. Again, it's not immediately obvious how to just take that and use it, but you'll find that they'll inspire designs that you'll see in the future. And they'll be just the right thing when you run into some architectural problem. What are some others? Any kind of algebraic approach to thinking about software composition, I think that, maybe, is the underlying theme. If you can understand nice, clean, elegant models of software composition. What are their properties? When are they useful? How can you use them to inspire the designs? You'll find that a lot of infrastructure work that you do tends to be better by having been inspired by these ideas that have been worked out by a lot of very smart people over many years, rather than try to come up with it yourself. And, I guess, maybe the last thing I'd say is a segue from that, and that is this is not something that we tend to to very well generally as software developers. But knowing a body of related work, and when you start a new problem, try to relate your problem to a problem that has already been seen, because chances are someone has solved it or solved

some variant of it that you can heavily borrow from. I think that's huge, and I think that that's maybe the biggest piece of advice I would give to a budding software architect. So what I'm hearing is the world is filled with all kinds of complex problems and the more weapons you can bring to bear to solving them, the more likely you are to be able get a handle on that complexity. Indeed, Indeed. Well, Kurt, this has been a wonderful story you have to tell about this, and I'm glad that you were able to apply some of your academic background to solving it. And I just want to thank you very much for conveying that story to the class. You're very welcome, and thank you for the opportunity to tell the story.

P4L1 Components

01 – Bottom Up Design

So far in this course we have been primarily concerned with top-down design. That is, when given a set of requirements come up with a set of pieces, connect them together and refine. In this lesson we will go the other way. Start with self contained pieces called components and put them together to build systems.

02 – Components

Pieces that we put together are called components. However, don't get this use of the term components, confused with the one that we talked about when we were discussing software architectures. Here's the definition that we will use for this lesson. A component is, an executable unit of independent production, acquisition and deployment, that can be composed into a functioning subsystem. The definition and other material for this lesson is taken from the book by Clemens Szyperski, and from the paper by Liwen Wang, both of which I have listed on the class resources page.

03 – Buy vs Build

To place the use of components into context. I want to describe a typical situation that often arises during industrial software development. The decision has to do with acquiring software assets. Should you construct them yourself? Or buy them from somewhere else? This is sometimes called the buy versus build decision.

04 – Buy Quiz

Below are some business factors to take into consideration when deciding whether to buy a software component from a third-party vendor. For each, determine whether the factor is enhanced or diminished if you decide to buy. Put a plus next to factors that are benefitted and a minus next to factors that are diminished. First, your uniqueness with respect to your competitors. Second, the amount of your staff time required to develop the product of which the component is a part. Third, your overall production costs. And fourth, your control of the development process.

05 – Buy Quiz Solution

When you buy a component from somebody else, you're giving up your competitive advantage. You didn't build it yourself, so you can't put into it the bells and whistles that might give you that advantage. On the other hand, the vendor can concentrate all their energy on that particular component and come up with a super-specialized version. So that's an, an advantage of buying. Your development costs may also go down. That is, the vendor can apply economies of scale, thereby reducing the price you have to pay for that, that component.

However, dealing with a vendor is a third party relationship, and the channels of communication that you have with the vendor may not be as effective as the ones you would have if you did in-house development.

06 – Build

Building means in house custom development. The advantages and disadvantages mirror those of buying. The build solution may cost you more, because you don't have the same economies of scale that a vendor might have. However, you can tailor what you build to your own situation. You also retain control of its intellectual property. However, overall delivery risk is increased when you build things yourself.

07 – The Third Way

Buy versus build is a tough decision. Sapersky however, describes a third way. Using third-party components that you can customize during assembly. In general with components, you get the risk and cost reduction benefits of the buy solution while enjoying the flexibility of the build solution

08 – Third Party Quiz

Here's a little quiz to get you thinking about the buy versus build decision. In addition to buying complete applications or components, there are several other ways that third parties can provide computational resources to a client. Match the examples in column one with a category in column two. Column one includes PThreads, the National Institute of Science and Technology's Time Dervice. Tom-Tom GPS, Checkstyle code checker and PHP. The categories include open source software, turn-key equipment, IDE plugins. Cloud-based services and software libraries.

09 – Third Party Quiz Solution

Well, PThreads is an example of a software library. The NIST Time Service however, is a cloud based service. You get the current time off of the internet. Tom-Tom GPS is a piece of turn-key equipment. Checkstyle is an IDE plugin. And PHP is open source software.

10 – Characterizations of Components

Now I'd like to take a minute to characterize components. They are pre-existing and general, making them reusable in a variety of contexts. The question arises, what does it mean to manufacture a component. Well to manufacture a component you just have to copy it. So the manufacturing costs are quite low. You can configure a component with respect to your needs and target environment. That is, there's a great deal of flexibility involved when you have components. The components that property built can be easily composed with each other and with noncomponent code. Components conform to a software component model that prescribes their syntax and semantics, and how they are composed. We will look into example software component models a little later.

11 – Component Life Cycle

First, let's take a minute to look at the component life cycle. Components are produced by external developers. Hence, instead of the normal breakdown between development time and runtime, there are really three phases to consider. Design time, deployment time and runtime. At design time, components are specified and built. During deployment, binaries are configured and deployed into target execution environment. At runtime, components are instat, instantiated and executed. As we shall see, major differences exist between component technologies depending on when composition takes place and whether a repository for the components exists.

12 – Component Models

Inherent in any component technology is that technology's component model, also called the component's framework. A component model is a set of shared assumptions about the component syntax, semantics and composition. Component syntax includes how components are specified, which need not be in the same language as the one in which they are implemented. The semantics prescribes what information is in the component's contract and what is the nature of the environment in which the component runs. Component composition specifies how components work with other components.

13 – Component Models Quiz

Here's a brief quiz on the component models. You might have heard of WordPress, it is a content management solution for blogs. Here are some requirements taken from the WordPress codex, for requirement stocking. For each requirement, determine whether it concerns component syntax, component semantics, or component composition. The first requirement states, any text output by the Action function, will appear in the page sources at the locations where the action was invoked. Secondly, use well-structured, error-free PHP and valid HTML. The third requirement states, actions are triggered by specific events that take place in WordPress, such as publishing a post, changing themes, or displaying an administration screen. The action is a custom PHP function, defined in your plugin, and hooked that is set to respond to some of these events.

14 – Component Models Quiz Solution

Well, requirement one, about any text output by the action function will appear in the page source, that's an example of component semantics. That is, it describes what the component will do. The second requirement concerning well-structured, error-free PHP and HTML, that's all about syntax. And the third requirement stating how the actions integrate with PHP is concerned with component composition.

15 – Examples of Component Models

Here are examples of some popular and representative component models. First off, from the Sun Microsystems, now Oracle Enterprise Solution, there are Enterprise Java Beans also called EJB. Included in this is J2EE and JSP, which is Java Server Pages. Of course, they compete with Microsoft and Microsoft offers COM, which is the component object model, DCOM, the distributed COM, OLE, ActiveX, and COM+ for transactions. Microsoft also offers .NET and in .NET there were technologies such as the common language infrastructure, CLI, and the common language runtime, CLR, and ASP.NET. An older component technology is CORBA and its component model is called CCM. CORBA itself stands for the Common

Object Request Broker. And it comes from the Object Management group. It includes an Object Management Architecture and an Interface Description Language called IDL. Finally and more generically, there's web services. These include web service description language, WSDL, Universal Description, Discover and Innovation markup, also called UDDI. And the Simple Object Access Protocol SOAP.

16 – Issues

To understand components better, we will now look at some of the issues that component vendor has to face when offering a component technology to the marketplace. The issues that we'll look at are configuration, versioning, extension mechanisms, callbacks, contracts, using objects as components, scaling, and domain standards.

17 – Issue 1 Configuration

The first issue is configuration. It may not be apparent from the discussion so far that components are typically configurable. What this means is that the component vendor provides a means, such as a configuration file, by which the client using a component can tailor it to a particular situation. This gives to the designer a powerful means for managing design trade-offs. For example, space versus time. This flexibility that configuration gives comes at a cost, however. Because configuration is a form of late binding, it becomes difficult to unit test the components in the actual usage environment. It is also more expensive to document and to deploy them.

18 – Issue 2 Versioning

The second issue is versioning, which can be tricky. As new versions of the components are released, backward compatibility becomes a problem. If you are a component vendor, you need to keep your customers up to date with changing standards, new programming language releases, and enhanced features. The question then becomes, to what extent should you remain compatible with previous versions? Think of the issue from the customer's point of view. They have a working product. It will cost them time and energy to upgrade to a new version of your component, and there's a risk of breaking their system if they do so. If they don't need the new feature that you are offering, they're going to be reluctant to upgrade. From your point of view, however, this may mean you've got to maintain and support a long history of previous versions at additional expense. Moreover, if there are multiple components involved, each with their own versions, you have an explosion in the number of combinations you have to support. What's a poor vendor to do?

19 – Versioning Strategy

Vendors have come up with a variety of strategies for dealing with component versioning issues. At a minimum, version numbers are used. When deployment is performed the version number is used to perform a compatibility check. Other strategies that vendors have taken include the following. Some vendors have ad hoc compatibility rules. That is the rules pertain to the particular version and change between versions. Some vendors claim that they have immutable interfaces, that is the vendor promises never to change the interface. Some vendors guarantee backward compatibility. Changes can be made but old versions are guaranteed to continue working. Some vendors have sliding windows of supported versions. That is, they will, support the previous five versions or the previous three versions. Some vendors take a middle ground saying they're going to break compatibility only with major releases that include major new features.

20 – Automobile Components Quiz

To think about these third-party component situations, think for a minute about what third-party components the automobile manufacturers rely on. See if you can come up with a few and type them into the text box.

21 – Automobile Components Quiz Solution

Of course tires are manufactured by third parties, batteries, and the fluids in your car. Sometimes the manufacturer will sell them, but also there are third parties that can supply oil and transmission fluid, and windshield washer fluid, and so on. Often, the brakes can come from third parties, typically mufflers do as well

22 – Issue 3 Extensions

The third issue has to do with Extensions. Components are often extended by adding new features. Recall that we met features and features diagrams when we looked at architectural views. When the vendors add new features complicating situations may arise. For example, the particular situation might require that there be exactly one instance of that particular component. This is sometimes called a singleton extension. If there are multiple components involved, we might want to ensure that at most one of the, one has this new feature added to it. Related to that is what happens if there are parallel extensions of, of multiple components in the same dimension. If we do allow the same feature, feature to be configured into multiple components, we may need to be aware of the possibility of resource contention, if each version, version of that component is trying to get access to the same resource. Sometimes there are non-orthogonal extensions, and we have to be careful of possible feature interactions, if a customer configures in more than one new feature at the same time. And then there are recursive extensions. Some component models support adding components that can themselves be extended. This may mean that the component vendor loses control of what components are actually deployed.

23 – Issue 4 Callbacks

The fourth issue is concerned with a technical consideration, the use of callbacks. A callback is an operation provided by the client. When a specified event is detected by component, the client operation is invoked. Callbacks can be a powerful tool that components can be use for interacting with a client, but they come with a price. System integrity may be compromised during the time in which the client is in control. Here is the situation.

24 – Invariants

Typically, the modules in a system are responsible for maintaining invariants. That is, they have to make sure they are in a consistent state before and after executing each service they provide. The same holds true for components, which are just third-party modules. However, the presence of callbacks makes invariant maintenance much more difficult. In particular, during the time when the client is handling the callback request, the component is vulnerable. Because it has given up control to the client, there's a danger that the client may do something that breaks the invariant, like make another call to the component.

25 – Callback Example

Here's an example. If your client code is making use of the GUI toolkit component that allows the n users to type into a text box the client code can register the name of a callback operation that should be invoked when the end user types into a, types into the field. During the period of this call the client callback operation has access not only to the event itself, but the other elements of the component state, such as the type text, the specific text box, and even the pixel position of the cursor on the screen. The client can take advantage of this information to do things like suggest completions or fix spelling mistakes. Imagine further that while com-, computing possible work completions, the client makes a direct call to the GUI component asking it to display a message. When the callback code eventually returns control the component there is no guarantee that the state of the text box is the same as it was before. It might not even be visible anymore.

26 – Callbacks Quiz

To check if you understand this, try the following quiz. Consider this sequence diagram describing a typical callback situation in which a Component captures a user event and invokes a client method via a callback. During the process, the Component is subject to corruption during which time period indicated by consecutive letters in the left margin?

27 – Callbacks Quiz Solution

Well, the answer is between events D and E. That is while the client is actually processing the callback itself.

28 – Callback Summary

To summarize, the advantage of callbacks is that the component can provide a structured regime of calling within which the client executes. The regime can orchestrate the order of operations in a way that the client would have to do by itself using the tradi, using a traditional approach. For example, the regime might provide an event loop. The cost of using callbacks is that the component state is exposed to the client at a time when it might not be internally consistent. That is, using callbacks makes it more difficult for components to guarantee their integrity.

29 – Issue 5 Contracts and Guarantees

The fifth issue has to do with the contracts the components provide and, and the guarantees they provide with them. Because components are provided by third parties, there, there is an increased need for a clear specification of what they are promising. On the class resources page, there's a paper by that lays out the different levels of guarantee that a component provider might promise.

30 – Level 1 Signature Contracts

We'll take the four levels from the simplest to the most powerful. The simplest form of contract that Boniyar calls level one guarantee is a syntactic or signature contract, in which the names and arguments of the component operations are specified. Thus guaranteeing it is possible to link components into an application. Can't really imagine using components in which you don't have this, this level of guarantee.

31 – Level 2 Correctness Contract

More powerful is, a correctness guarantee, in which the pre and post conditions are specified by all, for all callable operations, thus guaranteeing that the component operations successfully execute. We've seen OCL is a good candidate for expressing correctness contracts.

32 – Level 3 Collaboration Contracts

Level three has to do with collaboration contracts in which allowed interactions among components are specified. Addressing issue, issues such as synchronization, liveness and deadlock. The set of allowed interactions for a components complies that component's protocol.

33 – Level 4 Quality of Service Contracts

The highest level of guarantee, also the one that's hardest to provide, is called the quality of service guarantee. In these guarantees, non-functional requirements are addressed, such as availability, mean time between failures, mean time to repair, throughput, latency, data integrity, and capacity. Such an example of such a guarantee would be that the component operates with a throughput of some number of transactions given in a, in a given time period.

34 – Guarantees Quiz

To test your understanding of this, consider the following snippets taken from the Oracle documentation of the format method in the Java PrintStream class. Classify each of these snippets as to the guarantee level. The first snippet states data formats are not synchronized. It is recommended that you create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally. The second snippet is a snippet taken directly from the Java code for PrintStream format method, including its arguments and the name of the method. The third snippet comes from the context of the throw's IllegalFormatException. It states that if a format string contains illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions, then the IllegalFormatException is thrown.

35 – Guarantees Quiz Solution

Well the first snippet, which has to do with synchronization, is a level three guarantee stating what the collaboration conditions are. Snippet B, which is actual program text, has to do with level one guarantee saying, what are the names and types of the arguments, so they can be linked in when you actually run your program. And the third one, even though it states something about the syntax of the format string, is really a level two guarantee talking about the correctness contract between the client and this particular class.

36 – Summary of Contracts

To summarize contracts, when purchasing a third-party component. The customer needs to know what he or she is getting. One way to do this is to see a specification of the component that covers all four levels of guarantees. The alternative is to learn about restrictive limitations later when it may be quite expensive to overcome them

37 – Issue 6 Objects as Components

The sixth issue is also technical in nature. It has to do with using objects as components. It is tempting to identify objects and components, and many component technologies do exactly that, for example, JavaBeans. After all, both objects and components represent encapsulated state that supports operations. However, there are problems that arise when using objects to implement components.

38 – Object as Component Problems

We've already seen the problems with callbacks. And, of course, object-oriented programming language can have lots of callbacks. And an object which has the ability to make calls to self or this, which are self-referencing methods, compounds this problem. In general, with objects, it becomes much more difficult to guarantee contracts in the face of object callbacks and inter-method calls. The problem is even worse in the presence of multi-threading. Other problems like inheritance and fragile base class definitions are discussed in the next few slides.

39 – Inheritance Dangers

We have previously talked about the danger of using inheritance to implement generalization. If objects are used for components and if inheritance is used inappropriately, then subclass objects may violate component contracts.

40 – Fragile Base Class Problem

Another problem with using objects for components is known as The Fragile Base Class problem. This occurs when a new version of a component changes one of, of its base classes. Our existing derived class is broken. Imagine you've been using an object-like component by deriving from one of its base classes. Now your component vendor says that the new release of the component has changed the base class. Can you continue to run? Do you have to recompile? Do you, do you have to rewrite?

41 – Issue 7 Industry Scaling

The seventh issue has to do with scaling and here I mean scaling in the sense of the industry scaling up. As the component entry grows and evolves, a raft of new questions arises. For example, accounting. How should component use be charged, particularly in clients in which there are multiple components coming from different vendors. With respect to deployment and configuration, how are components packaged and how is configuration performed? There's lots of packaging technologies out there such as RPM, DMG, EPKG, Nix, and OSG. What about disputes? How do you deal with an unhappy customer if multiple components with different vendors from different vendors are involved? How about quality of service? How are quality of service guarantees satisfied when multiple inde-, independent components are used? And then there's fault containment and air handling. How do components detect and contain faults when they are not in control, in overall control of execution?

42 – Issue 8 Domain Standards

The last set of issues has to do with domain standards. One area of ongoing concern is the component, in the component marketplaces, the role of domain standards where there's a tension between proprietary solutions and open standards. Dominant vendors try to lock in customers with proprietary technology, while the competition promotes the use of standards

to encourage customer migration. In the longterm the community benefits from standard solutions. However, such standards take a long time for approval and penetration.

43 – Proprietary or Domain Quiz

Here's a short quiz about this issue. For each of the technologies below, enter a P for proprietary technologies and D for domain standard. The technologies include HTML, Direct3D, UNIX, OpenGL, Java and JavaScript.

44 – Proprietary or Domain Quiz Solution

First off, HTML is a domain standard, maintained by the World Wide Web Consortium. In contrast, UNIX is proprietary. It was first developed by AT&T and later sold to various other companies. Direct3D is a propriety graphics API, for rendering 2D and 3D graphics which was designed by Microsoft. Contrary-wise, OpenGL is an open standard graphics API and a competitor to 3D. The other two are interesting. Java, which originally started in an open fashion, being released by Sun Microsystems, is now proprietary. Whereas JavaScript, which originally started from Microsoft, was put into the public domain.

45 – Component Framework

Earlier I mentioned the role of component frameworks. Let's now take a look at some prominent frameworks to get a better feel for how the vendors are addressing the issues we have raised.

46 – Shared Attributes

These frameworks typically all provide late binding, persistence, encapsulation and subtyping. They provide support for communication among components including events, channels and uniform data transfer mechanisms. They also offer some form of component transfer packaging, such as JavaJar files, ComCab files, or CLI assemblies. They all provide a way of describing deployments such as via configuration file. This description is also typically available at runtime via mechanisms such as Reflection or metadata. Any given framework provides a way of serving, of serving components such as an application server model like EJB, COM+ and CCM or web server models like JSP and ASP.Net.

47 – Comparison of Differences

The framework vendors however had made some design decisions that differentiate their approaches. For example, they often differ on memory management. Java and CLR provide garbage collection, COM provides reference counting, and CORBA doesn't provide anything at all. As far as container managed persistence is concerned, Java has Enterprise Java Beans. CORBA has CCM, but CLR and Compost don't provide anything. The vendors also have various approaches for the versioning, some of them freeze, some of them have version numbers, some of them have compatibility rules, and some of them allow for side-by-side execution. Some other differences include the target environment for the component frameworks. J2EE and COM target servers. COM also targets client and desktop machines, whereas CORBA targets legacy applications. They all also differ in terms of their development environments. J(2)EE uses WebSphere, which is the commercial version of Eclipse. And .NET uses Visual Studio .NET. As far as protocols are concerned, Java and CORBA support IIOP and XML. Java supports, also supports RMI. COM and CLR support DCOM. And CLR supports XML and SOAP.

48 – Comparison of Supported Variability

It is also important to look at how the component frameworks differ in as far as their support for variability is concerned. Recall that one of the key advantages of components and their late finding is the fact that there's some flexibility provided to their customer. As far as Java and CLI are concerned, they use a single virtual machine for all platforms. Many languages can generate byte codes that target those particular platforms. COM also supports many languages, but on Microsoft platforms only. CORBA supports, supports multiple languages, but each of the, each of the languages must its own IDL binding. Each platform must also have an object request broker running on that particular platform.

49 – Future Directions

The component marketplace is real and more continue to grow. This is going to bring customers the benefits of well-defined, tested and supported, and documented solutions. Nevertheless, there are some ongoing concerns that need to be addressed. How about liability? When a system that uses a component fails, how is liability apportioned among all the vendors that have provided components to it? Quality guarantees. How can cross cutting quality of service requirements be addressed in a multiple component environment? If the quality of service has to do with performance which of the components is responsible for providing the performance guarantees? Well all of them. And the third question to be looked at. Contract persistent over versions. What is the white balance between new versions and support for incompatible exis, existing features?

50 – Summary

The advent of viable component marketplace has opened up a whole new approach to building software applications. Providing some of the flexibility that comes with building your own solution without incurring all the associated risks. You can therefore expect components and their frameworks to be part of the design thinking that goes into many future development projects.

P4L2 Coffee Maker Exercise

01 – Status

In the course so far, we have looked at analyzing problems, modeling them in UML, and designing architectural solutions. Now we want to look at the actual process of designing objects. To get started, we will do an exercise from Robert Martin that involves designing software to control a coffee maker.

02 – Robert Martins Coffee Maker

Here's how we will proceed. The following slide contains a textual descriptions of the functions of a coffee maker. It is followed by a description of the API to control the coffee maker. Your initial task will be to use the API and the textual analysis technique, we used at the start of the semester. To design a set of OO classes to control the coffee maker. You're going to express your answer in the form of a UML class model diagram describing your design.

03 – The Mark IV Special Coffee Maker

So this is the Mark IV Special Coffee Maker. We're going to assume that it is part of a series of coffee makers. It's the current one but it won't be long before the Mark V is coming out, so we have to be concerned with you know, the, the whole family of coffee makers and how things are going to change in the future. The Mark IV makes up to 12 cups of coffee at a time. There's there's a filter that has to be filled with coffee ground, and it has to be, the filter has to be slid into its receptacle. The user then supplies some some water in, into the water strainer and presses the Brew button. The water is heated until it's boiling. The pressure of the evolving system forces the water to be sprayed over the coffee grounds and coffee drips through the filter, your standard, standard coffee maker. The pot is kept warm for extended periods by a warmer plate, which only turns on if there's coffee in the pot. You don't want that baking in those grounds and being unable to wash them out in the future. If the pot is removed from the warmer plate while water is being sprayed over the grounds, the flow of water is stopped, so that brewed coffee does not spill on the warmer plate. Those are our very high level description of what the Mark IV is supposed to be able to do.

04 – Hardware Quiz

So one way of attacking a problem like this, is from the bottom up. That is, you are given a device that you are going to provide software support for. So determine what its capabilities are. Begin by making a list of all of the hardware devices that are part of the coffee maker.

05 – Hardware Quiz Solution

What did you come up with? So, based on the description we were given, there is a filter, and its associated receptacle that holds the filter in place. So, I guess I should have clarified that by hardware here, I meant electronic hardware. Okay. Okay? The things that the software

for the controlling system have to deal with. Okay, so I do have a couple of those elements as well. So there is a brew button, and all the connected, I guess, circuitry for the brew button. And maybe any other widgets that might be on this. But assume the brew button was the only one described. Okay. And then a plate that will warm the coffee pot, it has to be told to be turned off and on. We have to have that warming plate. We also have to boil the water. There's a heater for the- Right. Okay. It's a water heater as well. Mm-hm. Okay, and then there have to be some sensors. because recall we had to be able to determine whether there was coffee in the pot. Okay. And we had to know whether all the water was gone. That is, had been heated for boiling. So we needed a sensor there. And safety is always a concern, so we better have a pressure relief valve. Which wasn't mentioned in the requirements, but it's another example of coming up with requirements during the course of thinking about the system rather than all in advance. And it's for reducing the pressure in the boiler in case something goes wrong.

06 – Hardware

As is often the case in software development, the written problem description leaves out some other details. So to the above list, let's add three more hardware elements, an indicator light when the brewing cycle is over so you know when you can grab the cup and pour yourself some coffee. A censor for the boiler which determines whether the, water, there is water in it to be boiled. You don't want to turn on that, heater in the boiler, there's no water there. And, we mentioned the pressure relief valve.

07 – Hardware Design

If we were designing the coffee maker as a whole and not just the software controllers for it, we would now spec out each of the devices we just listed. Fortunately, the hardware team has taken care of that task and we have been given a Java API for the hardware. The API is available to you from the class resources page. It comprises methods and constant values. The methods provide a way of determining the status of the hardware device and to change them as requested. The constants give symbolic names to the inner, to integer values that, are needed by the controllers. Note that Robert Martin wrote this example before enums were added to the Java language. So he had to define them using the integer constants himself.

08 – Two Approaches

In the first phase of this course, you learned about object oriented analysis, in which you searched for various categories of words, nouns, verbs, adjectives, in the textural description of the system you were building. Martin contrasts this approach to one in which behavior is central. We will look at both, beginning with OOA.

09 – Traditional Approach Quiz

The traditional OOA approach to doing modeling is to search for nouns, in the problem statement and model them with classes. Do this now, on the textual description of the coffee maker, and produce a list of nouns.

10 – Traditional Approach Quiz Solution

So, there's many, there's many nouns in this in this paragraph description. let's, let's concentrate on the ones of those having to do with the electronics, the software for which we're going to be building in this example. Okay. So, I have the water strainer, but in terms of the

electronics, there's a sensor that we talked about being associated with that, that would tell us whether there's water in it or not. So, there's that, the brew button. And any other kind of control widgets, the on/off light for the coffee's ready. As well as the warmer plate. We're interfaced with that. I wrote down Mark IV special, that's kind of like the facade. Everything we're looking at The overall system? Right, so, there's that element as well. The other heating element, to actually, create the steam. And you know, and then the result with the coffee so those, those are some of the things that are listed. Okay well we could have, we could go through the whole process here of underlining the nouns. We could put them into groups, we could stem them just like we did before. But for now it's enough to get a sense of the the particular elements we're going to have to be primarily concerned with

11 – Class Model Diagram

So when you do this you could then come up with a UML diagram that organizes the important nouns related to the electronics into a, into a diagram. In this case there's main classes having to do with the coffee maker which as you indicated, then has parts corresponding to the graphical user interface. That would be the buttons and the, the lights to the the part having to do with the, the boiler which includes a sensor and a heater and the warming plate which also includes a sensor and a heater. And then you might get excited by the fact you have two heaters and have an abstract class for heater and the same thing for sensor and come up with a diagram that looks something like this.

12 – Limitations Quiz

One of the points of, of Martin's description of this is that this approach is going to get us into trouble. So like you to take a minute and think about any, any difficulties you can foresee using this diagram to go forward

13 – Limitations Quiz Solution

So one of the first, to me at least, it seems like obvious problems with this approach is that if we come back later and have the Mark 5 come out, there doesn't seem to be any real ability to reuse the components that we've already created. For instance, if our warmer plate is described here as just this warmer plate and we flush it out with functions and attributes and then the Mark 5 comes around and we have this completely new warmer plate. There doesn't seem to be any way for us to share some of the things that might be similar between those? Well, we have a play heater class and a boiler heater class. Why can't we use those? Well, I don't know. The heater, I guess, the parts of it that are general about the heating element doesn't seem to be specified here. Like, the boiler heater and the plate heater, they're both heaters, but I don't know what they're sharing. So, am I hearing that you'd like to see a separation between the classes that, a class that describes in general, the property of things, and then the specifics from Mark 4? It seems it should be defined more about how the heating is accomplished instead of just this general classification of it being a heater. If we're heating elements by this particular function, all the heaters or types of heaters that do this particular function, they should extend from that type of heater. So it's more of a functional approach, as opposed to this kind of general put everything in a box because it's a heater. Okay. So there's a couple of things going on here. Let's see what Martin had to say about this. So, Martin came up with four primary problems associated with this approach to doing the model. The first is that, obviously, we haven't paid much, if any, attention to the operations that are going on. And, in fact, many of the classes that we have here have no state at all. So they don't have any attributes either. If you have a class without state, it probably should be an

interface instead of being a class, okay? Second, what Martin labels as imaginary abstractions that even though the heater and sensor are base classes, they are not going to have much to share between them. Third is a term from the objectorian role called the God class. The only substance of class left, when taking account these the other problems, is the coffee maker class. And it contains all the interesting code. When you have that sort of situation, you probably haven't done a good job at your OO design. And so, we want to address that. And then even though we have the button and the light, we don't really have a user interface class. And so we need to represent that as well. So I have a quick question. Is the term here, god class, is there some carry over to that in the invariance maintenance strategies that we talked about earlier, where you have kind of this central node that seems like it's controlling everything instead of distributing that responsibility out amongst the rest of the components in this system? So I hadn't thought about that. One of the guidelines for OO design is to break things up into very small pieces, okay? There's reasons to centralize, okay, both at the architectural level, where you'd like to have knowledge concentrated because you want to control effectively where you might want to do things with respect to safety or security, things like that, but also at the object level. If there are things that might change separately, you probably want to have those in separate places, so you can change just what you need to change.

14 – Use Cases

So Martin suggests using an alternative approach to OOA is to write out use cases. Recall that a use case is a simple story illustrating a single execution or pointing out an important obstacle that the user might confront. For the case of the coffeemaker can you come up with a few use cases? What's brewin'? So if they did press the brew button, but that was, I guess, the first use case there's a couple of things that could happen. Everything could be right, because all of our pre-conditions are there, or there could be no water for the brewing process to start, or just the filter's not in place, or no coffee grounds. Those are some problems that could happen. So on, on that one notice that so far we haven't, we don't have an sensor for the grounds. OK. So if there's no grounds in there you're going to get very, very weak coffee. Okay. And if there's no filter, you're going to get your coffee pretty quickly. Right. I guess I have a question that regards to that use case. Is the problem of them not having coffee grounds something that the system should even worry about? Because it seems like that would be hard to tell, like the filter being in place I can imagine being something a sensor could tell. But how much coffee is in your, your filter could be, maybe something that's not supported, I don't know. So this is an issue with designing software and hardware at the same time. So you could imagine, for example in the Mark 5 having a sensor for the recep, receptacle actually clicking in. Okay. When you, when you push it in there. Trying to figure out how many grounds are in there or how, you know, how deep the grounds are or something our hardware people haven't gotten that one yet. Okay, so we're, we're not going to be able to sense that very well. It sounds a little bit like trying to count jelly beans in a jar or something. But anyways the other use cases I have involve remember moving or placing the pot because we have an indicator for that. Right, right. So it's something that stood out. And then also when you fill the water receptacle. We have an indicator whether the water is there or not. So that seemed to be a, an interaction that had something the user could see. So let's consider four of those. The user press, press pushes the brew button, and that's our primary one. And then some contingent ones containment vessel is not ready. The coffee is all gone or we could have that the, the brewing is actually complete.

15 – Brew Button Quiz

So notice that in this approach we don't have a class model. We just started with the use cases. And we're going to actually derive the class model from looking at the use cases. So let's look at the first use case, user presses the brew button and play through what happens. First off, which of the three classes, which of these three classes receives this event from the hardware interface? Check the, check the one that applies.

16 – Brew Button Quiz Solution

Sure, that's about as easy as you can get on these quizzes. Okay, so that's indicating that we'd better have a class for our user interface that's going to be able to detect this user event.

17 – Brew Button

So then after the user presses the button and the user interface detects it what's the, what's the next thing that you expect would happen? So I think that the next things that are hap, going to happen are going to be all of our checks to start the brew process. So we're- Make, makes sense. Mm-hm. Check if the water's there, check if our receptacles in place. Okay, so there's, there's two primary checks. Do we have, you know, have we loaded up the water, and then, do we have a, a coffee pot to hold the results of, you know, the coffee after we've brewed it. So those are two checks. Is does it matter which one we do first? I don't think so. Not necessarily. Okay. Just as long as they're all held. And so, I, I don't see either. And so, we're not going to be concerned about that. At that point things are ready. We can, we can start the boiler to produce some hot water.

18 – Collaboration Diagram 1

So Martin then does is express these steps using a UML collaboration diagram. We called that a collaboration diagram. Was like an object diagram. But the lines between the objects indicated our operations. And the lines were numbered indicating the order of the operations. So in this case, we had the steps of user interface getting, getting the request from the user with the brew button, and then asking the hot water source for whether it's ready. And then asking if the containment vessel whether it's ready, and then starting the hot water source heating up the water. So there's three steps labeled one, two, and three here. And we can see that there's, there ought to be at least three classes supported to deal with those three possibilities. Is it possible here for this diagram to be more descriptive? In the sense that, it seems like when you ask the ho, hot water source. Is it ready or not? If it says it is ready, then you might make the next check to see if the containment vessel's ready. But if it says no, it seems like a whole another steps set of steps may have to be taken to indicate to the user hey. The water source, hot and ready to go. Okay so remember that a use case, or a representation of a use case in a collaboration diagram or a sequence diagram is not contingent. Okay, it's, it's one step by step walk through the system. And what we're going to have to do is to deal with contingencies which are obviously an important part of this whole process. We're going to have to have multiple diagrams. And one of the things that Martin does is suggest that we can add together those diagrams. So, I think that maybe is where you're going. We want to have them, them all there. But, wha, what we'll do them a step at a time, so to speak. So we would have, I guess a use-case symbol user-presses brew button and a water vessel is not ready. Like, that would be a separate use case and we can model and tie that in later with this situation, which seems like we're pressing brew and everything's ready to go. That's where we're going.

19 – Containment Vessel

So in fact, let's, let's go there, let's, let's go to use case number two, which was the having to do with the containment vessel was not ready. So our first use case asks the question, but the question might come out negative and in that case we have like a contingency we have to, have to deal with. So this is a variant on the, on the first use case. So, in order to get there, the, the steps are that the query is sent to the containment vessel. If it is not ready then a message must be sent to the hot water source, telling it to stop or prevent the flow of water. Now what's, what's going on here is, really a couple things. One is, the startup situation where, before we even start. The, we check whether the, the pot is there, but it could also be the case that somebody pulls that pot out too quickly, okay? And so in that case it's not going to be ready either. In both of those situations we want to make sure that water doesn't start pouring through. We can extrapolate it and indicate that when the containment vessel is returned to the, the heating plate. Another message is sent to the hot water source, enabling it to resume. So we're talking about one to turn, turn off the hot water source and the other one to turn it back on.

20 – Collaboration Diagram 2

So if we're just concerned a part of scenario two having to do with, you know, the check being made and we're talking about maybe the user pulling back a pot after things have started. We have the first step being the start, being sent from the user interface to the hot water source. And then the user pulls off the pot and the containment vessel senses this and sends a message to the hot water source saying, hold it don't pour out any more water. And then later, the user puts the pot back on and the contain, the containment vessel says, okay, we can now proceed. And we can indicate that in a collaboration diagram labeling the steps one, two, three in this case.

21 – Use Case Addition

Now we can even think about adding together these two use cases. The notation we'll use is to say that messages labeled with the letter A correspond to use case number one and messages labeled with the letter B are a part of use case number two. And the steps are still numbered one, two, three, and so on. And so we see that we have now a diagram in which both of the use cases are listed there. And it could also have been the case that the second use case might have, might have used a different class that wasn't used in the first case. In which case, adding it in means that we now have you know, four classes in our, in our diagram.

22 – Brewing

So, let's now add in news case number 3. Brewing is complete. And started it up, its going and maybe the user pulled the pot or stuck it back in or whatever, but we eventual got to the end, brewing is complete. So, in this case, what, what, what steps do you expect to have happened. So at this point we could assume that there should be coffee in the pot. And if there's coffee in the pot, then we have that warmer plate that needs to be activated to keep the coffee warm. But I guess there's also the switch, the situation like you mentioned, the cup, the pot could have been removed and then it's replaced. So we got to do the check to see If, I guess, there's still coffee there in the pot after brewing is complete. But the boiler needs to be turned off, we're no longer boiling. The water source is not really relevant anymore either. I guess we need to somehow maybe prohibit our, well, I guess you could start brewing again but. I don't think that's an issue, I think I, I jumped to that being an issue, but I don't think

it is. And then we need to turn on the light to say hey, there's coffee ready. So we got that light there, so we need to inform the user interface as to where the light lives. So the one other thing that needs to go on is the containment vessel needs to be informed because it keeps track of how much coffee remains in the pot. And at the very start, the pot is full and so it has to be told that the pot is full. So there's a lot of messages and checks that have to go on here. Let's let's add those into the diagram as well.

23 – Collaboration Diagram 3

So now we have a collaboration diagram in which there are three classes and there are a series of messages going back and forth between them.

24 – Collaboration Quiz

And one more to go. What happens when, the party's over and all the coffee, maybe the party's not over, and all the coffee's gone, we have to indicate, the system has to be aware of this, so what, what has to go on as far as the system is concerned in this situation. So, if the coffee is all gone, the party, yeah, like you said, may just be getting started. But I guess the brew, or the, the your copy is ready indicate to on the user interface, would be turned off. Right. A lot of this has to do with keeping the state sane across all the classes

25 – Collaboration Diagram 4

And this, this leaves us with our final collaboration diagram which has all the messages going on. Now it's busy the diagram, the diagram has a lot in it, but it captures at least for this exercise what we expect of the system. In terms of the classes that are involved and the messages passing force path, passing back and forth amongst them.

26 – Alternative to OOA

Note that what has gone on here, we use the use cases to build up our diagram rather than starting with the nouns as we did with OOA. This approach is called role-based-design.

27 – Role Based Design

It goes as follows. First of all, we lay out the use cases. For each use case, we construct or elaborate on a collaboration diagram. Now, let's think about the collaboration diagram instead as a class model diagram with the arrows indicating dependencies. The rectangles instead of reflecting instances are going to be reflecting classes. Each class can be thought of as participating in a variety of roles. One for each use case in which incident in which it is evolved. The overall behavior of the class is the sum of its roles.

28 – Hardware API Quiz

Now we're ready to go to work with our, our actual object design. We have now used all of the use cases to describe the behavior of the three classes that comprise the coffee maker. One approach we could take to going forward would be to implement the described behaviors as methods in each of the classes by having them call to the hardware API. That is that, remember the Java code that we had that gave the API to the hardware pieces of the system? However, Martin says that this would be the wrong approach. Can you think why that might be the case? And as a hint here, think about reuse.

29 – Hardware API Quiz Solution

I can't think of anything right now, but it, I think, it's going to be along, along the lines of it's going to be hard to extend it, functionality later on at this point. You're getting there. And, and, you mentioned something about this this earlier on. One hint to all this, is we have designed three classes that satisfy the requirements of the coffee maker spec, but they, as yet, don't depend on the specifics of the Mark 4. If when we added the methods, we were explicitly include calls to the API, then we couldn't reuse those classes when next years version, the Mark 5, comes out. A better approach is to use a technique called dependency inversion. Let's see how that works.

30 – Dependency Inversion Principle

So far, we have designed an abstract coffee maker with three classes and a set of behaviors. By abstract I mean we, we haven't done anything specific to the Mark IV. This design is of value by itself. That is, it could be used in a variety of different models. And so we want to somehow, be able to capture that and reuse it later. Hence we should not add specifics to it, instead we should sub class it for each specific coffee maker. By sub class I mean have a sub class for each of the classes that is part of the overall system. In particular, the dependency inversion principle says that high level modules, like the ones that we have just designed, should not depend on low level ones. Like those wrapping the API. Note that this is the opposite philosophy from layered architectures, where high level model, modules are built on top of low-level ones.

31 – Example

Here is a image that describes a common place example of the dependency inversion principle. It features, a, situation, describes a situation, which there's a lamp that you want to get electricity to. And the way you do this is by plugging, the lamp into the wall. The lamp depends on the abstract plug interface and not on the underlying electrical wiring. It enables you to plugin a variety of different kinds of lamps and other devices as well. We want to, we want to come up with a similar way of dealing with that in, in our software designs.

32 – Realization

How do we go about realizing the dependency inversion principle for the coffee maker example? First, we express what we have done so far as abstract classes. Then we subclass each of them for the mark four. The subclass versions implement the abstract methods by calling the coffee makers API. Also the subclasses, when communicating with, with each other, use call to, calls to the abstract methods not to the specific versions for the mark four.

33 – Abstract Classes

When we have done this, we get a class model diagram that looks like the following. There's classes for the three classes that we had before, and then there's subclasses for each of them, corresponding to the versions, the specific implementation versions for the mark four

34 – Refinement

So what do we have so far? We have three abstract classes without abstract methods in them. We've defined three classes for the Mark IV. But we still have to fill in the details of those particular operations. We want to do this in a way that we don't compromise our abstraction. What Martin says about this is that none of the three classes we have created

must never know anything about the Mark IV. That is, none of the three abstract classes. This is the depen, Dependency Inversion Principle. We are not going to allow the high-level coffee making policy of this system to depend upon the low-level implementation.

35 – Solution

The solution approach is to add specific interfaces for the Mark IV that inherit from or implement our abstract classes. We can also factor out the specific communication mechanisms by which the sensors are accessed. Have a look at Martin's specific Java implementation which is provided on the class Resources page. So for the part about we could either inherit or implement these abstract classes for our design. I guess the situation for inheriting the abstract classes if we want to find some more policies about maybe a particular class of coffee makers. Like if we have, for instance, our containment vessel, and it has these abstract things that the containment vessels, the concrete classes would need to provide. Then we have maybe a slew of coffee makers that has this style of containment vessel that's a little bit more particular but still we want to have another kind of inherited, abstract class and not like an implemented, concrete class. [CROSSTALK] Let me comment a little bit and then see if you can refine the question here a little bit. So different object-oriented languages provide different capabilities for doing refinement. Java has both interfaces and subclasses but it only has single inheritance. C++ has multiple inheritance but it doesn't have interfaces, although they both have abstract classes. So when I said about inherent or implement, I was really being generic across the programming languages, okay? But in general, implementation of abstract interfaces is nice because in the abstract interface, you've captured something that is purely abstract. In an abstract class you can also have concrete methods. So if it's purely abstract you really captured the essence of something and we could even imagine a situation where each of the roles that is expressed by a class is defined in a separate interface. So we might have a relatively vanilla class that implements three roles, okay? And just provides the details of doing that. And then, if we wanted to change one of the roles or one of the use-cases, we just can go look at that one particular interface. So, given that, you want to re-ask your question? I think that covers kind of my question, I guess I have a better understanding, I think now. I guess the takeaway seems to be that with a very abstractly defined system like this, it's kind of like saying well, I don't care how you provide the isReady ready function for the containment vessel. As long as whatever you implement provides it, it's going to fit with this abstract definition of how our system works. So, you should be able to extend and have some kind of wild and crazy thing that does, isReady or maybe your more typical Mark IV design of how the containment vessel becomes isReady. So, one way of thinking about this is you're really, at this stage, designing policy. And that policy can take the form of protocols that is the names of the abstract methods that have to be called. And the attitude here is to be as abstract as possible, or at least the more abstract you are the more flexible you are with changes in the future. Of course you can go overboard if you're only ever going to have one version of this thing. You might not want to spend a lot of time abstracting on it. And maybe there's a sweet spot in between. But in general, and reflecting back on some of what we've talked about earlier, respect to maintainability of the modules and so on. You want to provide as much abstractness as you can in support of eventual reuse.

36 – Summary

So what have we learned from this exercise? First off, we could, we could use traditional Object Oriented Analysis. But this doesn't necessarily always lead to the best solution. We can think instead about using use cases to determine roles. And then use Dependency Inversion

to reduce the coupling and promote reuse. And the final lesson that you can get from this is, don't try to design any programs before you've had your morning coffee.

P4L3 Object Design

01 – From OOA to OOD

Earlier in the course we saw how object oriented analysis can give you a better understanding of the problem you are trying to solve. The results is a set of diagrams, interface specifications, expressed in OCL, and a list of non-functional requirements and constraints. Then the architectural design process can help you determine appropriate subcomponents. Understand the tradeoffs among the constraints. Select appropriate methods for satisfying the nonfunctional requirements. And choose a suitable architectural style. The eventual target is to produce a program written in some object oriented language like Java. The purpose of this lesson is to introduce you to Object Oriented Design, OOD. The process of getting you from your analysis and intended architecture to the point where you can write the target program.

02 – OOD

The primary concern of OOD relates to the individual classes and relations in the class model in our, analysis diagram. This process is sometimes called object design. Before considering object designs however, several other preliminary topics need to be discussed. In particular the inter model. Consistency among your diagrams. The actual steps in going from analysis to design. Some system design considerations. Abstraction mechanisms you might use. And, I want to take a moment to talk about as, a somewhat, different approach called collaboration-based design.

03 – 1 Intermodel Consistency

First of is Intermodel Consistency. The analysis process may have produce multiple diagrams, each of which we can think of as a model. These must be checked for inter-model consistency. For example, if you produce use cases, in, in the form of a use case diagram, you want to make sure that you have some representation for the individual use cases. Possibly in terms of a communication. Or sequence diagram, and the names would have to match up. Similarly, if you have classes that have complex behavior, you want to make sure that you have a StateChart for those classes, and in the StateChart, any events, actions and activities correspond to methods in the class model. And, if you, have data conditions on your StateChart. Those data conditions are going to refer to attributes. It had better be, the case that those attributes are listed in your class model diagram.

04 – OOA to OOD Quiz

For a start off quiz, lets consider the situation when you're about to begin object oriented design. You come in with a set of UML diagrams, possibly including OCL constraints, which you produced during the analysis phase. When you've completed the OO design, you also have a set of diagrams and these diagrams serve as a direct expression of what the target program will look like. What I'd like you to consider for a moment, is what is the difference between these

two sets of diagrams. In particular, what UML elements used in the OOA models don't have direct equivalents in object-oriented programming languages, thereby requiring some change to be made to the object-oriented design diagrams.

05 – OOA to OOD Quiz Solution

Are here are four. First off is associations. Associations are everywhere inside the OOA diagrams, but object-oriented programming languages don't have associations. It's going to be up to you to decide during object-oriented design how you're going to implement those associations. Related is the question of aggregation. We know in our diagrams that various, the diamond is used to represent aggregations. You will have some choices at programming time as to how you are going to represent what programming language futures you are going to use to represent those aggregations, and during O-design you have to make those decisions. Third off, is invariants, so in your OCL you may have to express certain invariants. And it's up to you in your program to enforce those invariants. That is, make sure that it's always the case that the invariant holds. During OOD design, you have to come up with some strategy whereby those invariants are going to be maintained. And finally with StateCharts, OO programming languages don't have state machines. And as we seen, the state charts that are produced during OOA can be complex, so you will need some kind of approach in your OOD for how you're going to deal with that complexity.

06 – 2 From Analysis to Design

As far as going from analysis to design and process there, here's three things that you should consider. First off, I would suggest treating the entire system itself as an object. And that any of the objects which you are actually going to have in your program would be attributes of that overall system object. Second is we just men, mentioned. It may be the case that we're going to have to decide how to deal with associations and invariants. And third, as you'll see a little bit later in this lesson. Although you're going to have in your OO design, classes that were originally specified during your object oriented analysis, you're also going to have some addition, additional classes. And we want to look at what are the circumstances that would lead you to having some additional classes.

07 – 3 System Design

The third preliminary topic is system design, and that is considering the system as a whole, possibly with respect to the environment in which it's going to run. First item is architecture. Okay? We've already talked about nonfunctional requirements and their effect on architectural style. Second is how you're going to deal with concurrency and there's a spectrum of possibilities here between having everything in a, in a single process to having one thread per object. Physical design has to do with how you're going to allocate tasks to processors and deal with any peripheral advi-, devices that are part of the system. If your application has a, a significant data storage requirement you have to make some decision about whether you're going to use a database to do that. Whether you're going to do files. And how you're going to deal with issues like lock, locking. And protocols talking back and forth to the data repository and the system. What is going to be the overall control regime of your application? Okay. Is your application going to be reactive, such, such as would be the case with, when you have a GUI and the user is controlling things with mouse clicks and, and key presses? Or is your application going to be proactive, that is, it has its own it has its own control in mind Whereby it calls the appropriate subcomponents in order to accomplish whatever its goal is. And finally, an important thing to consider is how you're going to handle

errors and failures of various sorts. Okay? Is there a recovery technique you have in mind? Are there error reporting approaches you're going to take with the user? How are you going to deal with that overall problem?

08 – 4 Abstraction Mechanisms

The fourth topic I'd like you to be aware of is the Abstraction Mechanisms which you can bring to bear in solving problems. As I indicated at the start of the course, the key element in success in doing design, is having experience in solving similar problems. The way that the human mind stores this experience is in terms, of chunks or vocabulary, such as words like client server, or visitor pattern. This Vocabulary can be thought of at different levels. So at one end are, programming idioms that is snippets of program text which, it's almost as if you've memorized in order to accomplish some specific task. Classes of course are an Abstraction Mechanism, and being aware of whatever Classes are available to you in system libraries can ease your task of, of constructing the application. A little while later we're going to be talking about design patterns and I mentioned one a second ago the visitor pattern. Being aware of those already existing solutions can help you avoid problems. And we'll be talking in this course about aspects but aspect orientated programming is a way of specifically dealing with cross cutting concerns. We'll also won't be talking much about object orientated frameworks. Well, they too are existing patterns of solutions which, if you are aware of them, can significantly increase your productivity. And, as we've already talked about, architectural styles, [UNKNOWN] approaches to solutions at the architectural level, are a powerful mechanism you want to be familiar with.

09 – 5 Collaboration Based Design

Finally, I want to say a word about collaboration-based design. When we did object-oriented analysis, we were essentially picking out the important objects in the description of the real world. And we're using those as the basis of deciding on classes and and the associations among them. And eventually we got to the stage where we were mentioning methods that the classes might provide as services. Collaboration-based design takes a somewhat contrary approach to things. In particular it's concerned, it starts with use cases. Individual use cases sometimes called user stories or scenarios. And with these use cases, each use case has various actors playing a specific role for that use case. Collaboration design, by the way, is sometimes called role-based design. So we can say that for a particular actor, participating in a particular use case, that actor's fulfilling some kind of role. So let's specify that role. That role takes the form of the actions that, that user takes in that user's story. Once we have catalogued those actions, okay, we can then examine other use cases that involve that same actor. And from the set of activities which that actor under, undergoes in the various use cases, we have built up a set of roles and the combination of that set of roles is in fact going to be what the overall capabilities that that class must provide. So we can synthesize the classes from the set of roles we've defined during this, this process. I'm not going to go into role-based design any further, but you may come across it in the literature or you may want to look into it more yourself and, and try it out. Because it can be an alternative approach that might be helpful under some circumstance

10 – Object Design

With those preliminaries out of the way, let's jump into Object Design. In particular, we're going to look at some of the specific elements that are currently analysis model and how we're going to deal with them during the design process. Those elements include methods, new

classes you might have to devise, how we're going to deal with generalization, associations, and dependencies. How we're going to implement control, and then how we're going to deal with abstraction abstraction, such as abstract classes, interfaces and types.

11 – 1 Sources for Methods

First, as far as methods are concerned, where do the methods come from? One obvious source is the operations in our analysis model. In addition to the specific services provided, there are signals, okay, and there maybe actions, activities, and events in any behavior model such as state charts. We also have to make sure that we're signing those operations in the form of methods to particular classes. In addition to those that come directly from the analysis model, there's others we should always be aware might need to be included in our classes. You should include constructors and destructors, getters and setters,copy constructors, printers or, or methods that construct string versions of the data inside of a class. Selectors, if you have, complex data, how you're going to take out the pieces of it. And, any kind of, iterators if you have, if your class has more than one, you know, is a collection class of some sort. How are you going to provide the elements of that collection. It's not necessarily the case that every class has to have all of these methods, but you should be aware of the possibility of their existence so you can determine early on whether or not that you need to define them

12 – 2 New Classes

I mentioned earlier that your analysis model has classes in it, and those classes are likely to show up in the design model. However, some additional classes may also need to be there. What's going on is the following. Object-oriented development methods have an advantage called traceability. What this means is that you can see, starting with the real world that's being modeled, classes that directly represent those real world objects. Those classes show up in the object-oriented design as well and in the ultimate code. That is, you can trace a line in either direction between the real world problem and the code constructs. This can be a real benefit in dealing with maintenance of the system. However, along the way, you may have to invent some other classes and that's which we want to examine. First and foremost, you may need to implement relationships. Your programming language doesn't have associations in it, and it's going to be up to you to come up with a way for each of the associations in the analysis model, how you're going to deal with it at design time and in your program. You're also going to have to deal with intermediate results. For example, if you have some complex computation that's going to be used in several places, you want to store it in some intermediate inter, intermediate variable and then reuse it in the two places so you don't have to recompute it. Constructing those intermediate results may mean you are inventing new classes. And third, you may want to invent new classes for abstraction purposes. Object-oriented languages provide you the ability to have abstract classes, which capture common features of lower-level classes, and inventing those abstract classes and including them into your design can improve long-term maintainability.

13 – 3 Generalization

I now like to begin looking at how you're going to deal with, you, and you'll know, relationships that show up in your analysis model. In your ultimate program, there aren't any, direct representations of those, particular relationships, so during design you need to come up with a strategy for how you're going to deal with them. First one we like to look at is generalization. Now, object oriented programming languages have a feature called inheritance, and I want to take a minute to describe to you the differences between inheritance and generalization,

because it can get you into trouble if you just routinely treat one as the same as the other. First off, Generalization is an abstraction between two classes that mean that all instances of the child class are also instances of the parent class. Inheritance, on the other hand, is an implementation technique, whereby messages sent to a child may be delegated to a parent. You can use inheritance to, implement Generalization, but you have to be careful how you do it.

14 – Generalization Example

Here's an example of how not to do it. Say you have a nice address book application with an address book class in it, and that class has a sort method in it. And now you're writing some financial application that or, or a financial part of an overall application, in which you'd like to do sorting. And the piece you are working on has to do with ledgers, for keeping track of credits and debits, and so on. So you'd like to, to use the sort that's part of the address book in your ledger class. So one strategy for doing that might merely be to have ledger be a subclass that is inherent from address book. But this would not be an example of generalization. An address book is not a general version of a ledger. This would be an abuse of power. So what can you do instead?

15 – Generalization Advice

So how should you go about thinking about implementing generalization. You certainly want to be able to take advantage of whatever inheritance feature is in your programming language. You just need to stick to certain rules. Typically, this means that in children classes, you can add features, but you don't want to take away features. And you want to really restrict how you do any kind of overriding, in the child class. When you do want to override a method in the child class, make sure that you obey the two following rules. First of all, when calling that particular method, you want to make sure that the child method can accept any arguments that the parents method could accept. That is, the child has to be as open to inputs as the parent does. Secondly, the output produced by the child method, when given the same arguments that were given to the parent method should produce the same result. What we're saying with both these rules is that for the same situation in the parent, the child needs to do the same thing. It needs to be, a special case. Now the child can do more. If the child is handling a special case of the parent it, it can deal with that particular special case. It just has to always, act as if it is, it is also obeying whatever rules it specified for the parent.

16 – Generalization Quiz

Here's a little quiz for you. Imagine that you're concerned with two classes, Squares and Rectangles. Do you make Square a subclass of Rectangle or do you make Rectangle a subclass of Square?

17 – Generalization Quiz Solution

I would say that it depends on what you actually mean by square and rectangle. So, if you define a square as a rectangle with both sides equal, then square is, in fact, a specialization of rectangle. That is, rectangle is the parent class, and square is the child class. If you defined square as a class having one attribute, which is the length of a side, and one method, for example, to compute the area. And, if you define rectangle as having two sides, that is an additional attribute, one for the height and one for the width. Then every rectangle is, in fact, a square, in the sense that it has all of the attributes the square does, and has all the methods a square does. So, this case, we would say that the square is the general class, and

the rectangle is the child class. The point being, you have to be a little careful what you mean by these classes, in order to decide who's the parent and who's the child.

18 – Implementing Generalization

There are a variety of different approaches to implementing generalization in object oriented languages. One that we just alluded to, is inheritance that follows specific rules. Second, you could just simply use a single class that has some kind of flag that indicates whether a particular instance is of a certain type, and you might have multiple types and the fly could have multiple different, different values. If you did that you would then be hiding the child data that particular type inside that that that class as indicated by the flag. In Java you could use interfaces or enums which are two mechanisms that allow you that enforce the rules that I've that I've laid out, you can use the state pattern when we get to design patterns you'll see. That there might be situations where you want to have some flexibility that's not provided by using strict sub-classing. For example imagine that you're implementing a, application having to do with a library and the library might have different categories of books. It might have one-week books, two-weeks books, four-week books, and it would, might be natural to say well, I'll have three subclasses of the, of the class book. But what happens when a one-week book becomes a two-week book? And auditory language is you can't change the class of something. Once you've established as as being of a class, it's, it's there for ever. So the state patterns allow, allows you to have a way to have dynamically be able to adjust the class of an object. And then for languages like C++ that have multiple inheritance, this gives you the, the ability to specify the properties that you wish to inherit in more than one class and be able to inherit from those classes just what you need.

19 – 4 Implementing Assocations

Well that was generalization. Even trickier than generalization is figuring out how to implement associations. OO programming languages do not directly support associations so the OO design process must choose the best means of implementing these associations. Some of the factors you have to take into account, first off is directionality. What this means is, if your program is going to need to interact with several classes, is the direction of that interaction always in one particular way? First A and then B or might you go in either way. Second is cardinality. That is for particular instance of one class are there multiple instances in another class? And third is the kind of access you will make into classes. Sometimes these are called the CRUD properties. Where C stands for create, that is how you going to create instances, R is for read, that means are you just going to query or access the, the instances, U is for update, that is, could you change the instances and D is for delete. Depending on, which of these particular kinds of accesses and how frequently they occur, it might particularly in the performance area effect how you choose to implement them. And finally as far as invariant maintenance is concerned, okay associations often have invariance associated with them. Like referential integrity constraints and it's up to you as the designer and programmer, to build your program in such a way that these invariants are maintained

20 – One Way Assocations

Let's start with the simplest case. That is, we have a one-to-one association and we're always going to traverse it in a single direction. We can do that with a simple pointer. Instances of class a are always going to refer to instances of class, exactly one instance of class b. And we're always going to go from the direction from a to b. You just have a simple pointer, which means in an object-oriented language, we have an attribute of the target type in our, in our

class. Okay, this is quite simple to do. And it even extends in the case where there might be multiple instances in the target class associated with a particular instance in the source class. In that case, instead of just using a simple pointer, we use a vector of pointers.

21 – Pointers

Here's an illustration of that process if we have an object of class Foo and it has various attributes in it. And if we wanted to associate with each class Foo, Foo, some number of instances of class Bar, we could just have a vector in each instance. And each of the vector elements would refer to one of the instances of the, the target class.

22 – Two Way Associations

Two-way associations are trickier than one-way association. One approach is to just duplicate the approach we had with one-way associations. That is have your attribute in class A pointing to an instance in class B and have in class B, an attribute which points to an instance in class A. Or you could do these with, with vectors if there's many, too many relationships. You can use this approach if if you don't run into referential integrity constraints. If you have referential integrity constraints, you have to make sure, for example, when you're deleting an element from a vector in instance of class A, that you also go over and find the instance in class B. And look in its vector to find the back pointer over to the instance in class A and make sure that's deleted. Puts an extra burden and complexity on your program. An alternative to having a symmetric solution is to just have pointers in one direction. If you want to traverse in the other direction, you have to do a search. So, let's say we'd like to go from an instance in class B to an instance in class A. First you have to find that instance in class B and from it go back and find the instance in class A. It's extra work. And a third approach I want to mention is to use associations themselves as objects.

23 – Associations as Objects

Recall our situation where we want to loop through all of the elements in all the instances of class B. And they're pointed to by instances of class A. What this means is we have to loop through all the A's in order to get to all of the B's and do whatever we're going to do with them. This approach if which is involves pointers in one direction only can be cumbersome and costly. Instead, you could implement the association between classes foo and bar by intros, by introducing a new class an association class that has two attributes. One is a name of or pointer to foo instance and the other is a pointer to the bar instance. That is, the instances are essentially a set of pairs. We have reified or made our association itself into an object. It's then an easy matter to link through all the possibilities, to loop through all the possibilities, because they're all instances of this association class. There is a cost associated with, with doing this, it, the cost takes the form of an extra step. If we want to go from a particular foo to a particular bar, instead of having a, a reference or a pointer directly to it, we have to go to the association object, look up that particular instance of the foo class, find the associated instance of the bar class and traverse in an extra step.

24 – Tables for Assocations

Here's what it might look like. That is, we use some kind of collection class, a hash table or a set or an array to hold all the pairs of associated objects. Each of the columns is going to correspond to an attribute in an instance and we have one instance for each of the links in the original association.

25 – Associations Quiz

Here's a short quiz for you. Imagine that you've had an association between classes for students and classes for courses. The association was called Take and the implication was that a student is taking a particular course. I have four options for you in how to implement this association, and I'd like you to give me a reason for each one which might be thought of as a disadvantage of that particular approach. The first approach would be a reference in each Student object to a Course the student is taking. Second option is a vector of references to students in each course. Third is to do a, an association class containing two attributes, one for a Student and the other for a Course. And the fourth possibility is symmetric vectors in Student and, and Courses pointing back to the other class.

26 – Associations Quiz Solution

For approach one, where we have a single reference in a student course. The problem is that using this scheme a student can only take one course. So for the second approach a vector of references to students in each course, this solves the first problem. But it makes it hard to find the courses taken by a student. That is you have pointers from courses back to students but you don't have pointers from students to courses. And the third option was to have an association class. And the disadvantage here, as I indicated a moment ago, was that there's an extra step involved in doing any kind of traversal, which might have a performance hit. And finally, the symmetric Vector approach in which there are vectors in each instance of the student class and each instance of the course class is perfectly general, but it may lead to referential integrity problems. That is when a student drops a course, we have to make sure that you remove that particular reference in the student class but also in the course class.

27 – 5 Implementing Dependencies

The third kind of UML relationship that we have to deal with during design is how are you going to implement dependencies. Dependencies are at once both the most common and the most varied kind of relationship. Having a dependency between two classes merely means that one class somehow uses the other. This relationship can be implemented in a variety of ways. Most simply you can have an attribute or a global object having the type of the target class. You could receive arguments of the target class in one of your methods. You could make a method or construct your call to the target. Or you could import the target as either directly or as part of a, a, a package into your class. All of those are examples of how you're using the target in your particular implementation.

28 – 6 Implementing Control

The sixth issue you have to deal with in doing object design, is how you're going to implement control. Recall then, in your analysis model, you may have some state charts. Those state charts describe the allowable behavior of objects of a particular class. How are you going to implement that behavior? Regardless of how you do it, recall that the state that you're dealing with, is essentially, the possible values of the class's attributes. The ad hoc approach merely says, write the code, okay? Treat it as each particular situation, that is each state is detected and you take appropriate events. So you are essentially implementing by hand that, that state chart. Fortunately, there are more productive ways to spend your time. There are libraries that already exist that support finite state modeling, finite state implementations and you can make use of one of those or as an intermediate step you could write your own table driven interpreter. That is the table rows have particular situations that you're in and

corresponding events or inputs that might arise and the table then tells you what to do or what to call when that particular, that particular situation obtains

29 – 7 Abstract Classes

The final topic in object design, that I want to mention, has to do with how you're going to deal with abstraction. During the process of development, you may wish to increase, maintainability by enforcing abstract interfaces. And there are various mechanisms in object oriented languages, available for you to do this. For example, there are abstract methods. An abstract method is essentially a method signature, in a parent class that says the types of the arguments and the type of the return value possibly also any exceptions that might arise. Child classes then, have to provide the implementation of the abstract class, obeying that particular signature. Any class, with an abstract method becomes an abstract class. And abstract classes can have no direct instances. It's only the child classes, that have implemented the particular method that can have instances. Essentially then, abstract class is providing a kind of contract. That all subclasses must obey. Java provides, goes one step further and provides the concept of interfaces, and interface is an abstract class in which all the methods are abstract. For example, the serializable class is an abs, is an interface. I should say that serializable interface is an interface. In addition there are no non-final attributes in interfaces.

30 – Modeling to Implementation Quiz

Here's a quiz matching OOA modeling concepts with the corresponding Java concepts that might be used for their implementation. The four OO modeling concepts are generalization, aggregation, invariants, and states. The Java concepts that might be used for the implementation include collection classes, methods and constructors, enumerations and subclassing.

31 – Modeling to Implementation Quiz Solution

First, the answer for generalization is D, subclassing. Subclassing is one way of expressing generalization relationships in code. The answer for aggregation is A, Collection Classes. Collection classes are designed to support aggregation. Examples in Java include ArrayList, List, Set, and so on. The answer for Invariants is B, Methods and Constructors. Constructors are responsible for establishing invariants in the first place. Subsequently, each method is also responsible for maintaining them. And the answer for States is C, Enumerations. Simple states can be modeled as enumerations, such as moods like happy, sleepy, sad. Java supports adding behaviors to enumeration instances, enabling custom actions.

32 – Summary

The object oriented analysis you perform goes a long way to determining what your final solution will look like. There however some issues that arise that you have deal with before you get there. Okay many of these resolve around how best to deal with UML elements at the associations and state machines. They don't exist directly in the OO programming language you're going to use. While there are many tools like design patterns, architectural styles, and design guidelines available to you. In most cases, you're going to have to think through the tradeoffs involve before choosing an appropriate solution.

P4L4 Design Patterns

01 – Design Experience

The single most important predictor of a successful software design effort is the extent to which the development team, staff, have experience on similar problems. Because we all want to participate in successful projects, access to that experience is crucial. One key source of such experience is familiarity with applicable design solutions. A design pattern is just that. A description of a solution to a problem in context. What this means is that for a given problem, a design pattern provides a way of solving it, including a description of the issues and tradeoffs involved. Situations where the solution applies, options that you as a designer have, consequences of using a solution, and any implementation issues. Thus, design patterns are reusable design experience.

02 – Architectural Patterns

A concept of design patterns actually arose in the field of the design of buildings, architectural design. In the 1970s Christopher Alexander wrote an influential book called A Pattern Language, in which he described various patterns which arise in the course of architecting buildings. For example, think about most of the public buildings that you've entered. You enter into an area called an atrium. Aside from enabling the entrance to orient themselves spatially, an atrium provide psychological and aesthetic means for adjusting to the building's purpose. Alexander's book describes many such patterns.

03 – The Gang of Four

In 1995, four master software designers documented their knowledge in a now classical book called Design Patterns. They were Gamma, Helm, Johnson, and Vlissides. Because of the number of the authors, this book is sometimes called the Gang of Four book, or simply GOF. After the publication of GOF, the idea of Design Patterns caught fire in the software design community. And books and websites have been publishing a variety of related topics, including analysis and refinement patterns. As well as programming language specific pattern catalogs. And even a book on anti-patterns, which are common problems to avoid. The resources page, of the class includes a citation of the Gang of Four book as well as Alexander's book. And a pointer to a website which has a catalog of such software design patterns

04 – Definition

What is a software design pattern? A design pattern is a solution to a problem in context. According to GOF, this amounts to a description of communicating objects and classes that are customized to solve a general design problem in a particular context. Thus, design patterns are a means of capturing and reusing design knowledge. We'll look first at an example of a particular simple pattern called the composite pattern.

05 – The Composite Pattern

The problem that the composite pattern addresses is how to organize information about whole parts relationships. That is, situations where you have to keep track of data about things and their parts. This is a common problem in software development. Think, for example, about user interfaces which are built from widgets such as windows, menus and forms. That are made by, made up of other widgets, text boxes, dialogues, color choosers, and so on. Other examples of composites include web pages made up of frames, pictures, texts, and links. Documents made up of chapters, tables, diagrams, and paragraphs. And UML diagrams made up of boxes and lines.

06 – Composite Classes

GoF is concerned with object-oriented design patterns, those that support building systems using object-oriented techniques. Patterns in the GoF book are presented in a stylized fashion, including one or more UML diagrams, primarily class model diagrams, objects, diagrams, and sequence diagrams. The class model diagrams for the composite pattern comprises four classes. Client class, component class, leaf class, and composite class.

07 – Composite Client Class

Let's start with the simplest class which is the client class. That is the class representing all of the possible uses of the composite pattern in some systems.

08 – Composite Component Class

The client interacts with the rest of this pattern through an abstract interface. For example, if your application provides the end-user a way of drawing diagrams, then the abstract interface might have the name Graphic. If you were modeling an organization and its employees, you might use the term Unit. To indicate the general nature of the pattern, GoF uses the term component. In the diagram you can see that the client class talks to the component class, and that the component class is abstract because its name is in italics. The component class has a variety of operations as well as the ability to add and remove children and access any of the children of the component that exists.

09 – Leafs and Composites

To the client, the value of the composite pattern is that it can treat all elements if the data one way using the component abstract class. That is where the component can contain other components, which is called a composite, or it's treating components that don't have other components, which are called leaves. Need not be dealt with as far as the client is concerned. For example, a graphic may be a line or rectangle. Which are Leaves, or it might be a picture containing other graphics which would be a Composite. Note that in the diagram Leaf and Composite are subclasses of Component.

10 – Aggregation

There is one further element to add to the class model for the composite pattern. It is an aggregation line from composite back to component. That is a composite can be made up of further components, there by allowing for hierarchies of any depth.

11 – Textual Content

The class model diagram gives you the overall essence of the composite pattern, but it isn't by no means the complete expression of it. The pattern's textural doc, documentation consists of several other valuable pieces of information. Note that each of the patterns in the Gang of Four book is formatted in a similar fashion, including diagrams, such as what we just saw, and a structured, textural description.

12 – Intent and Motivation

We'll now go through each of the structured paragraphs in the Gang of Four description of the composite pattern. First off is intent, which is a summary of the value provided by the pattern. The intent of the composite pattern is to describe a way of representing whole-parts hierarchies in such a way that, that the client treats individual parts and composites uniformly. The next section of the description is called the motivation section, and this typically takes the form of a scenario, or story demonstrating that having a problem, having such a solution would be valuable. Earlier, we used Graphic as an example. Clients of Graphic shouldn't have to test elements to see whether they are leads or composites, if all they want to do is copy them, for example.

13 – Applicability and Structure

The next section is called applicability, which includes important design considerations that the pattern addresses. Stated another way, this is the context in which the sol, solution can apply. The next section is the structure, which consists of the diagram we've already seen.

14 – Participants

After the structure description in the diagram is a section called participants, in which each of the classes in the descri, in the diagram is described as far as what its role is, what role it plays in the overall operation of the pattern. In our case, we had a component, we had a leaf, we had a composite, and we had the client itself. Each of those plays a particular role with respect to the overall operation of the composite pattern.

15 – Collaborations

After the participant sections come the collaborations sections. Collaboration is how the participants work together to accomplish the pattern's goals. The composite pattern is an example of a structural pattern. One in which the organization of the information provides the primarily, the primary value added. For structural patterns, collaboration plays a less important role than structure in providing information to the designer. Nevertheless, understanding inter-element behavior is important. For example, with the composite pattern, a typical behavior is to have the composites iterate through their children, performing some operation on each.

16 – Consequences

The next section is called Consequences. Which are the advantages and disadvantages of using the pattern. One of the most important elements of the pattern description is an understanding of what tradeoffs using the pattern entails. For example, the composite pattern makes the client interface simple at a possible cost of safety. That is, if we were to refactor the add operation into component. To make the interface more uniform to clients, this might

mean that leaves can have children. Which wouldn't make any sense unless we put in some kind of ugly check to prevent it.

17 – Implementation Alternatives

The next section in each pattern description talks about implementation. The design pattern that we've, understood so far has to do with the design of a solution. Not necessarily It's implementation. Implementation means translating that design into some code. Okay? Doing that often means that there are choices arise. And it's important to understand the implication of the those choices. The implementation section of a design pattern description lays out those implementation issues, and alternative ways of addressing them. For the Composite pattern, here are some of the issues that arise. We know that in the pattern so far, we have references from parents to children. An issue that you might wish to include, or a feature you might wish to include is, do you have pointers from children back to parents. Once you do this, of course, referential integrity problems might arise. Another issue is whether we would allow multiple parents to refer to the same children. Imagine that you have separate hierarchies in which the leaf elements are shared. This, of course, can be powerful if you wish to do it because it reduces the overall number of objects that you have. But it might also increase your code complexity if you were to do that. Similar to the situation where we just described in which moving the add operation up in the hierarchy has the benefit of making the uni-, the interface more uniform. However, it might lead to having unnatural operations at too high a level in the hierarchy. Similarly placing the list of children up one level, also would mean that somehow now leafs at children. Now, the issue is what data structure should you use to keep the list of children, their hash tables, their arrays, link, lesson, so on. Finally is the question of whether, when you delete a composite, do you also delete its children?

18 – Patterns and UML Quiz

Which leads to the following question. We all know class model diagrams distinguish associations in which the leading collection deletes its elements from those that don't. The question for you is, what is the visual indication of the former, that is, the situation in which deleting a collection deletes its elements? And I give you four choices. A triangle on the end of an association line, an asterisk on the end of an association line, a filled diamond on the collection end of the line, and a delete operation in the class itself.

19 – Patterns and UML Quiz Solution

Well, if you recall the answer is a filled diamond on the collection end of the line.

20 – Code, Uses, and Related Patterns

Patterns in GoF have three other sections. One is sample code and this may be the largest section, in which in a variety of languages including C++ and Smalltalk examples of coded uses of the pattern are included. As a side note, this particular book was written before Java become popular, so there are no java examples. However there are other books which include java solutions to similar pattern problems. Next section is known uses. As I said the authors of GoF were master designers and they had themselves had written [or were familiar with many important object-orientated systems. And they indicate which system use which patterns in this section of, of each pattern description. And finally, in the final section is related patterns. That is, how the given pattern relates to other patterns. Turns out that elegantly written applications often use multiple co-operating patterns. This is sometimes called pattern

density. This section of the book lists other patterns which might be used together with this pattern.

21 – Composite Pattern Quiz

Here's another quiz for you. Imagine that you are writing an application to manage parts inventories. That is, inventory management application. Match the class name from the composite pattern given in column one with the corresponding application data described in column two. The classes are Client, Component, Leaf, and Composite. The particular pieces of data are, a description of a StainlessSteelHexBolt, three eights inches. Some OutOfStockDetector, indicating you might have to reorder. InventoryItem class, and the BlueBirdBoxKit.

22 – Composite Pattern Quiz Solution

An example of a client class here, might be an out of stock detector. That is, you have an application that's going through your inventory, trying to find places where you might have to reorder. The component class here, is inventory item. That is, it's an abstract class of which all of the parts subscribe. An example of the leaf class is this stainless steel hex bolt. An example of a composite class, might be the BlueBirdBoxKit. Which is itself made up of other parts.

23 – Categories

Well, that was an example of a gang of four structural pattern, in fact the, the book has three categories of patterns. In structural patterns, the main value added is the description of the various classes and how they're connected to each other. The book also has a category called creational patterns, that describe ways in which objects can be constructed. The largest and most interesting part of the book has to do with behavioral patterns, which describes interesting interactions, interesting ways in which classes interact to accomplish some particular goal. We will now take a minute to look at each of these three categories beginning with the creational category.

24 – Creational Patterns

The book describes five creational patterns. Their names are singleton, prototype, builder, factory method and abstract factory. In a minute we will look at the singleton pattern. The prototype pattern is a way for designers to make use of a different kind of inheritance. Most object oriented languages provide class based inheritance. But some languages like LISP, provide a different way to inherit. Instead of inheriting from classes, you inherit from other objects. The prototype pattern tells you how you might get that same facility within a class based language. The builder patterns gives you a way of separating the actual construction of the object from how it's pieces are built. Factory method is a way that lets the sub classes decide which class to instantiate. The framework as a whole merely ask for creation. Specific creation is done by a concrete factory. And if you want to apply this method to a set of related classes, you can use the abstract factory pattern. For example, user interface tool kits may allow you to specify the look and feel of a set of widgets, and the abstract factory has a way of accomplishing that.

25 – Example Creational Pattern Singleton

We're going to go a little, into a little bit more depth with respect to the singleton pattern and provide you an example of it. Singleton's provide you a way to ensure that a class has only one instance and to provide a global point of reference to that particular instance. As far as motivation is concerned, consider the top level of your architecture where there may appear components that should only have one instance. For example, a database or a log-in manager. How do you guarantee that only one such instance exists?

26 – Applicability and Structure

For the single [UNKNOWN] the applicability is fairly obvious. There must be only one such instance of the class, and it's must be accessible to clients from a well known place. As far as structure is concerned, there is a single class called here, singleton class, but it can be whatever name you want to supply. The singleton class has two particular attributes. One of which is static or sometimes called a class attribute. That is, it's an attribute of the class and not an instance of the class. Here it's called unique instance. There may also be whatever data you'd like to have within that singleton as other attributes of the class. The singleton class also has some operations, and one of those operations is a class method. That class method is responsible for retrieving for the client the particular instance which is the only instance of the singleton. There of course may also be operations within the singleton like there could be in any other class for providing access, for example, to the other data that's there or doing whatever operations you'd like that singleton to do.

27 – Participants and Collaborations

As far as participants are concerned there's only one participant it's the singleton, it's responsible for it's own con, construction and it defines a class level instance operation, lets clients access it's unique instance. Collaborations are also minimal. The clients access the single instance through that class method

28 – Consequences

There are several consequences of using a singleton pattern. One of the benefits is, you provide controlled access. The only way to get access to the singleton instance is through that, that class operation. This has the potential of reducing problems with the names, the program name space. In particular, the alternative would be to have one or more global variables that refer to the instance. Once you've got global variables, they can be copied, and, and, and referenced. Thereby leading to potential problems. Because singleton is a class, it can be subclassed, which gives you additional flexibility. And if you were so inclined, you wanted to have a, a class in which there could be exactly two instances or three instances or four instances, whatever. Okay, you could take the basic idea of the singleton and adjust it accordingly.

29 – Implementation

In order to implement the singleton pattern, the first thing you do is define a class variable holding the instance. Then, you can define a class operation that creates the instance and saves a reference to it in the class variable. The operation checks whether the instance already exists and if not creates it. In order to protect yourself from creating other instances in implementing the Singleton pattern you make the constructor private or protected.

30 – Implementation Issues

Because access to the singleton is through a class, and class names are normally known globally, singletons somehow, sometimes act like global state instead of the traditional owned instances that we see in other uses of, of classes in object-oriented languages. We can also run into trouble in situations where the clients are multithreaded. That is, several threads may be trying to create that single instance at one time. Leading to the production of multiple singletons. Question arises as to when you create the single instance. One strategy is to do it at startup. Which you could think of as eager construction. Or do you wait until the first use to, to create it, which could be called lazy construction. Then some issues with respect to what it actually means to be a singleton. Does singleton a word mean at most once or exactly once? Similarly, does singleton mean only one ever or only one at a time? In languages with destructors, like C++, you could get rid of the instance and then later create another instance of that same singleton without violating the rule that there's at most one such instance.

31 – Singleton Quiz

Although they sound simple singletons are actually somewhat controversial because you can run into problems. Here's a little quiz that might get you into understanding what that problem is. Say you were in the process of writing a battery of unit tests for an application that you intend to run frequently during development. And that implementation might have use of some singletons. The question is what difficulties do singletons impose on such testing approach.

32 – Singleton Quiz Solution

If the tests are being run by a testing framework such as JUnit, in which a single process is involved in running a batch of tests. You have difficulty keeping the tests independent. That is, each of the tests might like to have its own unique copy of that particular Singleton to test against. This violates the principle of what a Singleton is.

33 – Structural Patterns

The next category of patterns, in the Gang of Four book, are the structural patterns of which the composite pattern is example we've already seen. Some other ones that are provided include the adapter pattern which you would use to convert an existing interface to look like another interface. The bridge pattern, in which you decouple an abstraction and implementation. The decorator pattern, in which you would add a single feature to an existing class. The facade pattern, which provides a higher-level interface for a subsystem. This might typically be used in a situation where you have some non-object oriented legacy code, which you'd want to access from within an object oriented application. And you need to make it look like an object oriented interface. The flyweight pattern allows you to use sharing to support large number of fine-grained objects. So imagine a situation, for example, when you're doing text processing, and each of the characters you'd like to treat as an object. Well, this can be quite expensive, because, because, there, because be tens of thousands of such objects. Instead flyweight, allows provides you a way of doing this without creating all those objects. Finally the proxy pattern allows you to control access to an object.

34 – Behavioral Patterns

The third category of patterns described in the book, comprised the behavioral patterns. As you might guess, usually these are the most complex patterns and hence the most powerful ones. Behavioral patterns, describe interesting ways that objects can interact.

35 – Catalog of Behavioral Patterns

The gang of four book, includes descriptions of 11 behavioral patterns which I briefly survey here, and then we have an example of one. First of is the chain of responsibility pattern which allows you to separate a request from the mechanism by which the request is handler, and also allows for you to have multiple handlers for a given request. Second is the command pattern, which takes what sounds like a verb, and converts it into a noun. That is, you can have objects that represent commands. Third is the interpreter pattern, which is quite complex because it provides a mechanism, essentially, to have an interpreter for a language. You can represent the grammar and interpret its instances based upon whatever operations are expressed in the language. Next is the iterator over enumeration pattern of course, now languages like Java and C++ have iterators and enumerations. But at the time the book was written, they weren't part of the language and hence they were described there, you might think of, the occurrence of that pattern in the book as a motivation by which the eventual feature was added to the other programming languages. A Mediator pattern is a powerful way of encapsulating object interactions, into an object. The Memento pattern captures an object's internal state for later restore. Think here about undo and redo, you want to capture the state so you can go back to it if you do an undo. Next is the Observer pattern, sometimes called the Listener pattern, which is a way by which classes can notify dependent classes when an object changes. The state pattern is an interesting one, in which we've alluded to before, that you could use in situations where you might like to change the class of a particular object. Example I think we gave before had to do with library books, in which they went from being one week books to two week books to four week books. Of course, in most object oriented languages you can't change the class of a object, State Pattern is a way of doing that. The next pair of classes are often useful for representing algorithms. The Strategy Pattern is a family of algorithms with the same purpose and interface. An instance of the pattern is a specific algorithm. Related to that is the Template Method Pattern, which is a skeleton of an algorithm with hooks for the specific step. Finally, the Visitor pattern is a way of applying a method to elements in the structure and we're going to use this as an example of a behavioral pattern.

36 – Visitor Pattern

The Visitor Pattern is a popular way of navigating a complex data structure applying item-specific operations. Moreover, Visitor is a natural complement to Composite, which we saw earlier. That is, the data structure being navigated by the visitor can often be represented using a, a composite class.

37 – Vistor Pattern Description

The intent of the visitor pattern, is to vary the operation to be performed on the elements of a complex structure, without changing the classes of the elements of the structure itself. As an example use, imagine that you have an abstract syntax tree, it might be in a compiler, useful for representing a program. You might wish to walk the tree for various reasons. For example, to generate code, to pretty print the, the program or to do type checking. This would result in three different visitors, all walking the same composite data structure. The

motivation that the pattern addresses is to be able to decouple the structural elements that is, the data structures. From the operations applies to them. You think about this. This means that there are two factors that control how you are applying the operations. One is the data structure itself which may have, may, may have many different kinds of nodes. And the other is the class of operations such as, the code generation, pre-printing, and type checking

38 – Visitor Applicability

You would want to use the Visitor pattern. If you need to perform several different categories of operations on the elements of a complex structure. And you want to simplify the element code by factoring out these operations. For the Visitor to be a value, the data structure would be relatively stable. You wouldn't want to change it very much because that would, that would break the overall structure. However, the operations can change, you can add new ones without, without breaking the overall structure of the system.

39 – Structure

Here's a picture of what the visitor pattern looks like, there of course is a client class, which is going to lead the operations to be applied. And then two categories of other classes. One is a category having to do with the data structure itself and the other is a category of classes having to do with the visitors. As far as the data structure is concerned, there will be some kind of abstract element and then concrete elements corresponding to the different parts of the data structure. The abstract element provides an abstract method called accept with an argument visitor. That is, as you are navigating through the data structure and you want to apply the operations you send the visitor, as an object, to each of the elements you come to. And it must accept that visitor, and essentially call back to the visitor to perform the operations.

40 – Comments on Structure

If we were talking about using the viter, visitor pattern inside a compiler, then the concrete visitors might one might do type checking, one might do pre-printing, and so on. And, the concrete elements might correspond to things like assignment statements, or declarations, or other parts of the Code. The object structure class, itself, represents the parse tree as a whole, and is your starting point for doing the navigation through the structure.

41 – Visitor Participants

There are five sorts of classes involved in the visitor structure. The participants, one is of course, the visitor itself, which is an abstract class declaring a visit operation, that is then applied by each of the concrete elements. ConcreteVisitors are specialization of the visitor class, implementing an operation on each of the concrete elements. And, in addition, they may store local state, that is, if your navigation wants to accumulate statistics, there's a place to do that accumulation inside the ConcreteVisitor. The Element class is an abstract class declaring the accept operation that takes a Visitor as an argument. It is sub-classed by Concrete Elements, representing the various different kinds of nodes in the complex data structure, and each of those elements takes an except operation with a Visitor as an argument. Finally, the fifth of the classes is the Object Structure class, itself, which usually provides a way of enumerating the various elements, serving as the root of the data structure itself.

42 – Visitor Behavior

The Visitor pattern is an example of a behavioral pattern. And in order to describe the behavior, we use a sequence diagram in this case. Recall that in the sequence diagram, each of the columns corresponds to a different object. The horizontal lines correspond to messages being sent among the objects. And that time marches down the page. The first column in the diagram corresponds to the data structure itself. And it is responsible for sending messages to each of the concrete classes, and those messages are accept messages passing in whatever visitor we currently want to implement. The concrete classes are in the second column. Their responsibility is for doing the callback. That is, they are given a visitor as an argument and they need to pass themselves to the particular visitor operation responsible for whatever visitor they're currently implementing. Those are the messages at the top which go from the second column over to the fourth column and in the middle of the screen from the third column over to the fourth column. Finally, in the visitor operation itself in the fourth column, can make calls back into the elements, taking advantage of whatever operations those elements provide.

43 – Visitor Collaborations

As far as collaborations are concerned, the client is responsible for creating instances of a concrete visitor object. And traversing the object structure. The visited concrete elements, called the visit operation. With self as an argument.

44 – Visitor Consequences

The visitor pattern is quite powerful and popular however there are some issues with it. First off the implementation of the operations are placed in a different place from the Elements being operated on. It means that the operations are kept together. The elements are kept together but in a sense encapsulation is compromised because those two are separated. Second consequence is that adding new operations is straightforward, you just have new classes on the visitor side of things. In a sense you are actually extending the operations on a class without changing the class itself on the other hand adding new element types is hard. This would break the data structure and cause a lot of reprogramming. Final consequences, if you need to, visitors can accumulate state as I indicated before an example would be collecting statistics.

45 – Visitor Implementation

Couple of issues arise with respect to visitor pattern. First off, if you think about it, the actual operation called at any time is dependent on two things. An element such as assignment statement and a particular visitor such as type checking. This dependency is sometimes called double dispatch. It most of object oriented languages you're familiar with, there's single dispatch. That is, you send a message to a particular object. That, whatever method responds to that message, depends on what object you're sending it to. In languages like Ada, the determination of who's going to handle a particular message is determined not just by one argument, but by all the arguments. Here we're looking at a situation where we're going to make that determination on what operation we're going to apply based upon two arguments. Second issue is, who is responsible for performing the actual traversal, if we're talking about a compiler what we want to do is a tree walk and there are variant, various variants of tree walks. We can place the code to perform that tree walk in several places. We can place it in the ObjectStructure class, in the Visitor class, or we can have some kind of Iterator object.

46 – Pattern Quiz 1

Which design pattern does the following object model represent? Enter your answer in the text box. The design pattern listed here includes three classes, one labeled Application, one labeled Wrapper, and one labeled LegacyComponent.

47 – Pattern Quiz 1 Solution

The answer is the adapter pattern, which is responsible for altering the interface that an object provides to conform to the needs of its clients. Often these clients comprise legacy code that cannot be readily altered.

48 – Pattern Quiz 2

Which design pattern does the following object model represent? Enter your answer in the text box. The design pattern features five classes, a reader class, an abstract converter class, and three concrete converter classes.

49 – Pattern Quiz 2 Solution

The answer is the builder pattern which isolates the steps involved in constructing a complex object from the representation of that object.

50 – Pattern Quiz 3

Which design pattern does the following object model represent? Enter your answer in the text box. The pattern comprises seven classes. There's a Client class and an abstract class called Collection, which has two subclasses, ListCollection and MapCollection. There is also an abstract Traverser class with four methods, and two subclasses, a MapTraverser and a ListTraverser.

51 – Pattern Quiz 3 Solution

The answer is the iterator pattern which is responsible for traversing a collection, applying some action to each element. This enables clients to visit each element without necessarily knowing how the collection is implemented. Of course since the time of publication of the Gang of 4 book iterators have been added to the java and C++ languages

52 – Problems with Patterns

As I said earlier the Gang of Four book was immensely popular and has spawned a whole, a whole movement within the software development community. However, patterns are not without problems and I want to look a few of them here. These come from Czenecki and Eisenhecker book which is listed on my class resources page. First off, patterns are primarily implemented using, using two standard object-oriented compo, compo, composition mechanisms, inherit inheritance and object composition. But using patterns can add complexity. Often, there's the introduction of extra objects and extra levels of indirection. So your code becomes more complicated even though you're using standard techniques.

53 – Problem Area Object Schizophrenia

Second problem area is sometimes called object schizophrenia. And this is, an example of this was the visitor pattern, where we split off the functionality having to do with the elements into a separate place where all the operations were contained. In a sense, this beaks delegation. It has also been called the self problem. When you delegate responsibility to perform part of a computation to methods in your attributes, as opposed to handling them yourself. This can lead to additional complexity, particularly when you're trying to debug and find out where something went wrong.

54 – Problem Area Preplanning Problem

Another problem has to do with the fact that in planning your applications, patterns are only useful if you know enough in advance to include them in your plans. What do you do if you're part way through the development of your system and you realize that you should be using a particular pattern. Well, you can use refactoring to get you there but the only real way to deal with it is to become quite familiar. With the catalog of patterns. So that you can a, think about them in advance in order to apply them when you need them.

55 – Problem Area Traceability Problem

A final problem I want to mention can be called the Traceability Problem. The code of the program that uses a design pattern does not necessarily make explicit that you're actually using the pattern. You don't have to use the names of the classes taken from the book. And, as you saw, there were lots of options with respect to implementation. Consequently, somebody reading the code may not realize you're using a pattern. Therefore, there's an extra obligation on you as a developer for documenting and using certain naming conventions, so it's obvious what's going on. If you're not careful, this can lead to increased the failure to do this documentation can lead to increased fragmentation of the code, having to do with the additional classes and methods which patterns introduce.

56 – Summary

To conclude, patterns are an essential part of developers' vocabularies. We simply must be aware of what's there in order to take advantage of it. However, patterns are difficult to learn passively. You can read all the catalogs you want. But unless you actually get some experience with catalogs, they're not going to be, come to mind readily when you're in a development situation. However, despite these costs and potential problems that might arise from using patterns, they are so powerful that you want to take advantage of them when you can.

P4L5 Design Principles

01 – Design Guidelines

At the start of this course, we mentioned that design can't really be taught, but has to be learned through experience. That said, we have offered several ways in which we can learn from the experience of others. Including catalogs of architectural styles and design patterns. In this lesson, we'll go over another catalog, that of design principles. Which are informal guidelines to be judiciously applied in appropriate circumstance

02 – Design Quality

Design guidelines are one of several ways of gauging the quality of your designs. Of course, the ultimate validation of a design is to build a program and have its users report their satisfaction. Short of that, you can build a prototype or conduct design reviews. There are also various metrics based on the structural properties of your design that compute actual numbers assessing the design's quality. Least expensive, but conceivably most valuable, is adherence to the principles we will go over in this lesson.

03 – Design Guidelines

A Design Guideline, also called a design principle or design heuristic, is an informal piece of advice about the structure of a design. With respect to objected-oriented designs, we are concerned with Design Guidelines that take the form of do's and don'ts. We will survey some of the most well known design principles. But first, let's review several important foundational concepts

04 – Coupling

Recall earlier on when we talked about coupling. Which is the extent to which the module is independent, from other modules. You would like the coupling of your modules to be low in order to make it easier to maintain them. Coupling, as a software design principle was invented by Larry Constantine.

05 – Cohesion

Another foundational concept is cohesion, which is the extent to which a module has a single purpose. You would like the cohesion of your module to be high in order to enhance its understandability and promote its reuse. Cohesion as a software design principle was also invented by Larry Constantine.

06 – Orthogonality

The third foundational concept to review is orthogonality. Which is the extent to which the features of a system can be varied independently. You would like to enhance the orthogonality of your system in order to make more options available to its users. Orthogonality also clarifies system descriptions and documentation, and supports automatic generation of system components. The principle of orthogonality was developed by David McGovern and Christopher Date.

07 – Information Hiding Principle

The fourth foundational concept is information hiding, which is also called encapsulation. This is the extent to which the implementation details of a system are hidden behind abstract interfaces, thereby protecting other parts of the program from changes to those details. Information hiding is one way to reduce the coupling of a system. On the other hand, the use of inheritance can violate information hiding by making parent classes implementation details, details visible to child classes. This principle was developed by David Parnas.

08 – Foundational Concepts Quiz

Here's a quiz for you to try out what, your understanding of those four foundational concepts. In column one are the four concepts, and in column two is an effect or benefit of concepts. And see if you can match concept to its effect, by putting the letter for the concept into the box on the right.

09 – Foundational Concepts Quiz Solution

In terms of the first benefit in proving reusability. Well, cohesion does that by making each of the modules that you have, have a single purpose which you can then identify and reuse that module if you have that particular need. The second benefit is enabling maximum variability, and that's what orthogonality is intended to support. A third one, in terms of raising the level of abstraction, is what information hiding does by preventing access to details and implementations and finally, the fourth factor is requiring more code reading and that one is coupling if you have highly coupled systems and then you change one of them you may have recode in many other modules

10 – Design Principles Catalog

We now begin the catalog and for each of the particular principles we're going to give the principle's name, its author, a definition and the implications of its, of its use. But, before we start, please remember these are guidelines and not hard-and-fast rules and in any given situation you have to decide which principle may or may not be appropriate and whether just how you're going to apply it. The guidelines presented in this catalog are taken from the writings of many different authors and more details on the sources are found on the class resources page.

11 – Liskov Substitution Principle

The first principle is the Liskov Substitution Principle named after Barbara Liskov and she proposed the principle that subclass instances should, should satisfy parent class constraints or contracts. That is, if the client module accepts and works correctly on a parent class instance it should also work when a child class instance is substituted. This implies that child class

instances should obey parent class invariants and method contracts including their pre and post conditions.

12 – Law of Demeter

Karl Lieberherr has developed the Law of Demeter, which suggest limits on the classes that can be refered to by a given method. Imagine that you are writing code for a method m of an object o. Your code can refer to features of other objects or features, either in attribute or a method. The question is, what other objects is it reasonable for you to refer to? Answering everything can lead to typely, tightly coupled systems. Instead of everything, Lieberherr proposed some limits to the objects that can be referred to. You can refer to features in O itself. You can refer to features. In classes that are the, the classes for the parameters that go to the MethodM. You can refer to any objects created or instantiated within M, and you can refer to the objects O's direct component objects, that is, its attributes. Obeying the Law of Demeter reduces coupling, but sometimes requires introduction of extra wrapper classes.

13 – Hollywood Principle

Donald Wallace introduced the Hollywood Principle for object oriented frameworks. These frameworks consist of a set of abstract classes together with rules for the ways in which, their concrete subclasses may interact. These rules suggest that calls should be made from the framework to client classes, rather than the other way around. The pattern of frameworks calling clients is the opposite of the situation where normally a client would call the resources in a library. Hence the principle is also called inversion of control. Wallace dubbed the principle the Hollywood principle, after the supposed response by a Hollywood producer, to yet another unsolicited screenplay. Don't call us, we'll call you.

14 – Dependency Inversion Principle

The next principle is Robert Martin's dependency inversion principle, which we have seen in an earlier lesson. It says that high level modules should not depend upon low level modules. Both should instead depend upon abstractions. This is related to inversion of control, in which normally abstraction framework classes would make use of concrete client classes. It is the opposite to the way that modules are structured in traditional layered architectures. Stated another way, Martin is saying that our layering should be one of abstraction, rather than one of control or data access. Doing so will lead to designs in which the controlling principles are enforced at the highest levels of our architecture

15 – Open Closed Principle

Bertrand Mayer, author of the Eiffel programming language has proposed the open-closed principle: that a class should be open for extension but closed for modification. The implication is that after you have released the class, any enhancement to it should be made only in subclasses. This policy, this policy will help deal with the fragile base class problem that we mentioned in an earlier lesson.

16 – Design Principle Quiz

Here's a short quiz that looks at the principles we just covered. Imagine that you have designed the following classes, one for a Motor, one for a FancyMotor, and one for a Robot. The Robot class contains within it an attribute of type motor. That is, it calls upon that motor to perform some computation, here called motor.run. Assume that the Robot class is very

complex, and we now want to change it to instead make use of a new FancyMotor. Making this change will difficult with the current design because it violates which design principle? There are five choices for you here. There's the Substitution Principle, the Law of Demeter, the Hollywood principle, the Dependency Inversion Principle, and the Open-Closed Principle. Which of these is the one that's violated?

17 – Design Principle Quiz Solution

Well, the answer is the dependency inversion principle. We've got things just backwards from way that they should be. If we want it to be easy to make this change. In particular, if instead of the given design, we have an iMmotor interface that motor implements and robot uses. Then, when we add FancyMotor, we merely have to make sure it implements iMotor. Robot doesn't have to change at all. The solution is inverted because instead Robot depending downward on Motor and FancyMotor, all three classes depend upward on iMotor.

18 – Interface Segregation Principle

Here's another principle by Robert Martin, called the Interface Segregation Principle. Robert Martin developed this, principle, which suggests that clients depend on an interface to a part of a large class's features rather than directly on the large class. Note the relationship of this principle to role-based design that we discussed earlier in the course. In role based design, classes are synthesised from interfaces each of which reflect a role that objects of that class might play. The implication of this principle, like that of role based design, is that class interfaces should be broken into small pieces, each corresponding to a single use case.

19 – Reuse Equivalency Principles

Several other principals proposed by Martin concern Reuse. One of these is the release Reuse Equivalency Principle that says that the granules of reuse should be the granules of release. That is you should release your your software in such a way that the pieces can be individually reused. Of course, reuse can take place at all levels so, what Martin is really suggesting is that we re, Release highly cohesive code units. In particular, Martin suggested Java packages as a good unit of release. The converse of this principle is the Common Reuse Principle, also by Martin. Classes that aren't reused together should not be grouped together.

20 – Common Closure Principle

Sometimes we can't hide the implementation of a design decision inside a single method or class. It's just too big. Martin's Common Closure Principle says that regardless of the level of granularity that we are forced to use, we should group the related elements into a common release unit. In Java, this would typically be a package. Stated succinctly, this principle says that classes that change together, should be released together.

21 – Dependency Structure Matrix

The next few principles make use of a device called a dependency structure matrix or DSM. This device was devised by Baldwin and Clark, to deal in general with management of design changes. A DSM is a Boolean matrix in which the rows nad columns correspond to components with a one in a cell indicating that the component in a given row depends on the component in the given column. Note that the order of rows and columns doesn't matter to the information conveyed. So we can feel free to permute them, in order to produce more meaningful views.

22 – Lattix Image

Here is a screen capture from a tool called Lattix, that can construct these dependency matrices from code and point out violations of various principles. The Lattix tool provides several interesting features. On the left hand side of the image, actually is conveying the hierarchical structure of the system's components. This hierarchical structuring can be dynamically specified by the user. So within the left hand column there are some sub columns. The ones on the extreme left, contains the ones that are just to the right of that, which contains the ones just to the right of that and so on. In the Lattix version of the DSM, here shown, the numbers in the cells are not just zero and ones, they are integer values which indicate the number of dependencies. More over, the user of the tool can specify the kinds of dependencies and get numbers for each of the different kinds. The red triangles on some of the cells indicate violations of user specified design principles. And, the internal brown squares that subdivide the overall DSM indicate candidate modules having no violations. The user can construct such modules, by suitable column and row permutations. And once you've done that and you can then focus on the remaining cells that have violations and try to get the whole matrix to be violation free.

23 – Acyclic Dependency Principle

Martin's Acyclic Dependency Principle states that the dependencies between packages must not form cycles. Expressed in terms of D.S.M.'s this says that you should be able to permute the rows and columns of the D.S.M. in such a way that the transitive closer of the matrix is lower triangular. What this means is that there's a strict ordering among the Components, such that a Component only depends on other components beneath it, and never on one above it. A violation of this property is seen in a system where component A depends upon component B, and component B depends directly or indirectly on component A. Not only is it difficult to maintain such systems it is even hard to understand them in other words, you can't understand A without understanding B, and you can't understand B without understanding A. There are several ways to deal with violations of the Acyclic Dependency Principle. If you have a situation where A depends on B and B depends on A, you can invent a module C, take the part of A that B depends on and place it in C, and have both A and B depend upon C. Another way to break the cycle where the packages are siblings is to add an interface class into b and have a implement it.

24 – Stability

Martin uses the term stable to mean hard to change. Or, if you try to change it, it's going to have many implications. Typically, a module's hard to change if a lot of other modules depend on it. Martin's stable dependency principle suggests that you should depend in the direction of stability. In other words, no package should be dependent on packages that are more likely to change than it is. This principle is similar to the previous one, that is, you should depend downward and not introduce loops into the dependency hierarchy. Note that we usually think of the term stable as a positive term but Martin is treating it as undesirable. A corollary to this stable dependency principle is Martin's Stable Abstraction Principle in which stable packages should be abstract packages. The idea is that they're hard to change but easy to extend.

25 – Bad Smells

Kent Beck and Martin Fowler popularized the notion of refactoring as part of extreme programming in the 1990s. The idea was to move some design activities that were previously done before implementation was started into the actual implementation phase of development. The intent was to reduce rework in situations with rapidly changing requirements. The first step in refactoring is the recognition of bad smells, which are code situations that are suggestive of design problems, such as duplicate code, too many comments, or long classes. So you can think, bad smells as being, things to avoid in other words, design principles describing situations which you, you don't want to be in. The Fowler's book recognizes dozens of bad smells and the avoidance of each should be, could be thought of as a design principle. For example, the duplicate code bad smell should be thought of as the avoid duplicate code by factoring principle. You're encouraged to explore Fowler's book as a way of becoming familiar with these situations, it is referenced in the class resources page.

26 – Design Heuristics Riel

Another participant in the development of design principles is Arthur Riel, whose book is titled Design Heuristics, referenced on the class resources page. Here are some examples of Riel's heuristics. You will notice the overlap with some of the principles, principles that we have already talked about. These particular heuristics don't have names that are catchy like the but they do exhibit. Advice to you about situations that you want to avoid or ways of structuring your code that you want to try to promote. First one is, most of the methods defined on a class should be using most of the data members of the class most of the time. Otherwise there's an opportunity to split the class into pieces that indi-, individually obey this principle. Another Riel Heuristic, check constraints in constructors, rather than in method preconditions where possible. Following this principle will reduce the overall amount of checking that needs to be done by the class. Another Heuristic, factor the commonality of data, behavior, and interfaces as high as possible in the inheritance hierarchy, thereby facilitating reuse. This of course is standard O-O dogma. Here's another Riel heuristic. Inheritance should be used only to model a generalization hierarchy, hierarchy and not to facilitate the sharing of implementation code. We've heard this one many times in this course. Another prefer composition which we can also think of as aggregation or delegation. Over inheritance. Particularly with respect to implementation inheritance. Another. It should be illegal for a derived class to override a base class method with a no-op method. That is, essentially, a method that does nothing instead of the behavior that the base class prescribes. Doing so, by the way, would violate the substitution principle that we saw earlier. Riel also suggests that we not change the state of an object without going through its public interface. Doing so would violate information hiding. If we, kind of extend this idea to deal with, subclassing, and we strictly obeyed it, this heuristic, it would mean that a method in a class cannot change an instance variable without calling the setter. Method in that class. That is, you couldn't make a direct assignment to an attribute. You'd have to call the setter which did it. And another Riel principle, users of a class must be dependent on it's public interface. But a class should not be dependent on it's users. Finally here are two Riel heuristics indicating how you should distribute key design knowledge among the components of a system. Distribute system intelligence horizontally as uniformly as possible. That is, don't artificially concentrate knowledge in one place. This heuristic is sometimes expressed as, do not create God classes, or God objects in your system. A corollary heuristic is to distribute system intelligence vertically down narrow and deep containment hierarchies. You're, I encourage you to have a look at Riel's book, where there's many more such pieces of advice.

27 – Single Choice Principle

In procedural code, systematic variation is often dealt with via case statements or else-if cascades. In object-oriented code, this approach is considered a bad smell. Instead, you should use parallel, factored subclasses with the choice specific code embodied in a subclass method. This approach is called the single choice principle.

28 – Transparency and Intentionality

I want to end the catalog by mentioning two other principles that I have gleaned from my personal work. Transparency and intentionality.

29 – Transparency

Of course, we came across transparency when we talked about middleware and we listed various kinds of transparency that was appropriate to middleware situations. In general, transparency suggests providing interfaces that enable client code to be written without having to be concerned with specific details. Of course, this generality comes with a cost of extra design and testing work.

30 – Intentionality

The last principle that I would like to mention to you is also the most abstract one. It is called the Principle of Intentionality. That is, design your software in such a way that your intent is manifest and localized in the code. What this means is that the conceptual distance between the problem that you are trying to solve, and the code with which you are solving it is as small as possible. Intentionality supports traceability, validation and maintainability. You can improve intentionality by appropriate use of cohesion, and naming conventions.

31 – Principles and Heuristics Quiz

Here's a quiz that covers the various principles that we've seen in the catalog. James Gosling was the author of the Java programming language. At a Java users group meeting he was asked, if you could do Java all over again, what would you change? He replied, I'd leave out classes. This, of course, got a lot of laughs, and after the laughter died down he went on to explain what he meant. I'll turn it around and ask you, which of the principles or heuristics that have been mentioned in this lesson, support Gosling's idea about leaving out classes.

32 – Principles and Heuristics Quiz Solution

There're actually several relevant principles, such as the Liskov Substitution Principle, the Interface Segregation Principle, the Stable Abstraction Principle. Riel's Inheritance, should be used only to model a generalization hierarchy. All of these express Gosling's belief that inheritance should not be used share implementation, but instead use implementation of interfaces to share abstractions.

33 – Summary

I recognize that a catalog of design principles is too abstract to be immediately useful to you. I hope, however, that by being made aware of these principles, you'll be sensitized to problematic situations when they arise. You can then look up the relevant principle and it's suggested solutions to help you resolve the issue that you you've seen.

P4L6 Design Reviews

01 – Introduction

High quality designs become more important as the size of the application being designed grows. But as the size grows, so too does the likelihood of some kind of design flaw occur. It therefore becomes essential to validate the designs and the most common way of doing so is with design reviews. This lesson looks at design reviews, the participant's roles. The process of performing them and general guidelines for producing effective reviews.

02 – Exercise Intro

Let's begin with an exercise. We'll use a code example instead of a design example, but the principles of reviewing are the same. The following code is written in Java. It computes the sin function for argument x to accuracy e using a Maclaurin series expansion. In the example, line numbers appear in parentheses at the beginning of each line. They are not part of the program but are there so we can refer to the lines.

03 – Defects Quiz 1

Here is the example program. As an exercise, write down any defects you detect alongside the relevant line number. There are no wrong or right answers for this quiz.

04 – Defects Quiz 2

For the same program, indicate the types of errors found at each marked line. Mark b for bugs, d for documentation issues, v for violations of coding standards, or i's for inefficiencies. Note for multiple errors, use comma separated entries.

05 – Defects Quiz 2 Solution

It turns out there're at least 13 different defects in this short piece of code, starting with line number 1. First off, you can't meaningfully get a double precision result from a single precision argument. This is just a bug. In line 2, there's a documentation problem. In the comment, the word declaration should be instead, method. Also in line 2, in the comment, there should be a space after the right parenthesis in sine of x, and this is an example, another example of a documentation problem. Also on line 2, the comment should mention that the algorithms use the Maclaurin series expansion. Finally on line two, the comments should actually include what the series looks like in terms of x minus x cubed over 3 factorial and so on. These last two are both other examples of documentation problems. Lines 5 and 6 are an example of another bug. In this case, some analysis will indicate that the given solution won't work for negative values of x. On line 6, to conform with the standards of the rest of the program, there should be spaces around the equal sign. This is a coding standard violation. Line 6 also illustrates an inefficiency. The variable term gets initialized twice. Actually, the initialization on line 4 is the superfluous one. On line 8, there's another inefficiency. It makes use of exponentiation,

which in these sample, simple circumstances, could be replaced by a more efficient x times x. Also in lines 8, because the value of x doesn't change inside the loop, the computation of x times x could be moved outside of the loop, thereby improving efficiency. On lines 9 and 10, there's another bug. The returned value produced by the algorithm is wrong because the test comes before the accumulation. Line 10 illustrates another inefficiency in the computation of sum. The exponentiation is merely doing the job of alternating signs. Surely, there's a simpler way to do that. Also on line 10, another inefficiency. Multiplication is used to flip the values of the sign and this is inefficient also.

06 – Observations

I expect that no one of you found every one of these defects. But that, the class as a whole noticed most or all of these problems. I also expect that some of you have noticed problems that are not on the list. The point is that groups do better than individuals. This is sometimes called the many eyes phenomenon. There's a cost, however. If 200 people each look at 13 lines of code for 15 minutes, this amounts to more than one week of staff time. At current loaded salary rates, this might cost a company thousands of dollars. Is it worth it to a company to spend that much money to find these problems? Would a smaller group size have worked just as well? Would more time per person have worked better? Notice also that there were all kinds of defects. I intentionally didn't tell you in advance what the word defect meant, but here there were bugs. There were documentation issues. There were inefficiencies, and there were even violations of coding standards. Software engineers have studied defect detection and concluded that team review efforts can be cost effective way to find defects. However, the reviews must be done in a systematic fashion.

07 – Reviews

A review, which can also be called an inspection or a walkthrough, is a systematic reading of a software development artifact. Reviews can be a cost-effective way of finding defects in the artifact, and reviews complement other verification techniques such as testing and proofs. That is, they find, tend to find problems that the other techniques don't find. The purpose of a review is to detect defects, which, depending on the artifact, might be called bugs or faults. Reviews may also be used to check adherence to corporate or governmental standards. Reviews should not be used to educate staff members, report status or fix the detected problems. Reviews can be applied to different kinds of artifacts produced during the software development. These include requirements documents, specifications, architectural designs, detail designs, new code, fixes, test plans, and documentation itself. Effective reviews are systematic. It is not sufficent, sufficient to just have a meeting and talk about an artifact. People's time is expensive and group meetings are especially so. Here are the recommended steps to take for an effective review.

08 – Step 1 Planning

First off is planning. During the planning phase, participants are selected, a meeting is scheduled, roles are assigned, the specific artifact, or part of an artifact is specified. And the materials, that is, the artifact, the review form, and any background materials, are distributed to the, the participants. This planning should be complete about five days before the scheduled meeting, to give time for the participants to prepare.

09 – Step 2 Preperation

The second step is the preparation itself. During this preparation period, the participants should individually study the material, noting any potential defects. The idea here, is to save time in the meeting by having the participants detect particularly superficial type problems that can be reported to the, reported before the meeting. And not have to take time during the meeting to go over them. The expected rate of individual review should be about ten pages of text or 100 lines of code per hour.

10 – Step 3 Review

The third stage is the review itself. The actual meeting takes place. In general, it should last no more than two hours, lest fasi fatigue set in and effectiveness, overall effectiveness be reduced. The rate of review at the meeting should be approximately the same, as that used for individual preparation. During the review meeting, the individually noticed defects should be collected. In most cases they should not be further discussed at the meeting. The detailed meeting process will be described after we discuss the roles of the participants.

11 – Step 4 Rework

After the meeting is over, there's a rework period. The artifact's author should investigate the issues that are raised. And, if, in fact, they are defects, they should be, corrected or at least saved in an issue tracking system for later correction on a subsequent release.

12 – Step 5 Follow Up

And finally, there's a follow up process. The author of the artifact should report to the moderator the results of the reworked process. The moderator should confirm that the fixes have been properly implemented. Also, the moderator should collect data on the review itself. Such as the number and types of defects detected, the number of participants, and the total time spent reviewing. This data should be recorded and saved, so that the process of reviewing itself is being reviewed and, and possibly improved if it, if it can be. Finally, the moderator should suggest these improvements to the review process. And, take them up with, the organizational, quality people, so that o, over time, the effectiveness of the review process itself can improved.

13 – Roles

In formal reviews, the participants play specific roles, including that of a moderator, a recorder, a reader, and 3-6 reviewers.

14 – Moderator Responsibilities

The job of the moderator, in addition to those already mentioned, concerning preparation and follow-up, include the following. The moderator should determine whether the participants have done the necessary preparatory work. If necessary, the moderator should abort the review if the team is not prepared. Why, after all, go through an expensive meeting if it's not going to be effective? The moderator should also evaluate whether the work to be reviewed is actually ready for review. Let's say it's a code review. And the organization has guidelines that say, before code review the code must be completed, it must be successfully compiled and it must go through unit tests, the moderator can then check whether those events have happened and if not, send it back and reschedule the meeting. A moderator is responsible for running the meeting. Keeping the participants on track, arbitrating any differences, and managing time. Moderation is a skilled activity and moderators typically have under, undergone some kind of

training at the task. Finally, moderators should be technically competent, but not necessarily expert on the specific artifact of technology being reviewed.

15 – Recorder

Another important role during a review meeting is that of the recorder. The recorder is responsible for making a record of the issues raised during a review. Note that it may not be possible to determine in a review meeting itself whether an issue represents a defect. This determination may require some offline research. It is the job of the recorder to proactively clarify the issues raised. During the course of a heated, heated discussion, this may be difficult. Several different issues may intertwine more of a two seemingly different concerns may actually reflect a common question. The recorder will often ask for clarification until he is satisfied that he understands the essence of the problem being discussed.

16 – Recording Form

The recorder typically makes use of a form for recording issues. The for includes for each issue raised, it's location within the artifact, it's description, it's type and it's severity. Issue types are artifact dependent for code review for example there may be logic issues, library issues, standards conformance issues and so on. Severity levels are also typically pre-defined, and some suggested ones are listed on the next slide.

17 – Severity Classification

Each organization should determine it's own severity classification, based on its release and artifact peculiarities and whatever source issue tracking system or source control system that they have. Here's one possible schema. It includes three different levels. The least severe is that minor re, rework is required. That this rework can be verified by the author. This would be the case if there were questions about the comments or standards conformance, something like that. A somewhat more severe level would be where there's conditional rework that it would be verified by the moderator. And then, for major rework situations, reinspection is required. And here a guideline would be that if greater than 20% of the document or 20 hours of work or 100 lines of code have been affected then a rereview might be required.

18 – Reader

Another review role is that of the reader. Reviews should be systematic and thorough. One way to deal with this need is to explicitly address each part of the artifact being reviewed. For example, in the code review, each line should be individually looked at. The reader is the person responsible for enforcing this thoroughness by leading the participants through the artifact and for each part paraphrasing what, what the artifact is expressing for that part. The paraphrasing should be its, should be descriptive, and not try to say why that particular part is there. Some organizations run their reviews with the artifact's authors being the readers. Other organizations make sure that someone else does the reading. I think in the case of the first type of organization, the thought is that an author might buy us the discussion by emphasizing certain things, the author felt as important, even at the expense of perhaps hiding some details which need to be looked at more carefully. In either case, the reader should use impersonal pronouns such as it, referring to the artifact, rather than referring directly to the author with I, or he, or she, or something like that. Personalizing a review by using I and, and you, and, and so on, can raise the defensiveness level of the participants thereby reducing the review's effectiveness.

19 – Reviewers

The other participants in the meeting are the reviewers. These ae the people responsible for raising the issues. Not to say that the, the moderator or the recorder or the reader can't list some issues, but their, their primary focus is going to be on their other, on, on their individual roles, whereas the reviewers, the job here is to do exactly that. To review the artifact and to point out raise, raise issues. Typically a meeting will have three to six reviewers, fewer run the risk of not having enough eyes on the target and more can be overkill. Right not being effective as far as the, the time invested in, and the number of defects found. Reviewers should raise issues by asking questions, as opposed to saying that's a problem. They should not explicitly suggest improvements, but rather ask if the author thought about doing things in an alternative fashion. The viewer should not blatantly assert defects, but ask what would happen under different circumstances. By taking a questioning attitude, the team can productively raise issues without getting diverted by emotionally driven debates.

20 – Review Meeting

So those are the participants. Now, let's talk for a minute about the review meeting itself. For a effective reviews, the review meeting should be itself, be structured. This includes the following steps. First off, introduce the participants to each other. They might not previously know each other. In particular, one effective strategy would be to bring in a member of a different team. who's, you know, has a similar level of expertise, but to, to put a different set of eyes that may not be biased by the you know, common understanding. Second, is a statement of objectives by the moderator. This is a reminder that the purpose of the meeting is to raise concerns over specific artifacts. And not get diverted into problem-solving or, or other other issues. Third is an evaluation of the preparedness to determine whether the meeting can go forward. This involves checking with the reviewers as to whether or not they had done their preparation, and to collect their issues that they've already found. And also to check whether the artifact itself is ready for review. Next is the systematic review itself using some means of ensuring thoroughness, and we'll look at some of those in a minute. This is where the term walk-through is very important. The actual going through the artifact in a very systematic step by step fashion to make sure the whole thing is covered. During this process it could be recording of results in the form of the issues raised on the review form. After the systematic review, but still inside the meeting, there should be some kind of summarization, often led by the recorder of the issues raised, including a determination of severities and priorities. Finally a determination of who is responsible for looking into these issues. In many cases it will be the author. But it may be that in certain cases somebody else gets assigned to do that. There should also be an agreement about how resolution will be verified. With respect to is it the moderator's responsibility? The author's? Or is there going to be a view review

21 – Thoroughness

A key determinant of a successful review is how thoroughly the participants examine the artifact. There are variety of means that have been devised to encourage this thoroughness. Line by line coverage of, of the code or the documents involved. similarly, if, if it's a, a diagram is being, reviewed or going through systematically on the visual elements of a diagram. If we're talking about the early stages, the requirements document, it may be the use cases, and making sure all the use cases are going through. Another technique for ensuring or promoting thoroughness comes at things from a little bit different point of view. And this is a check list based reviews. The checklist is based on common def, types of defects, either derived from

common industrial practices, or company specific empirical data. So for example, if it's a code review there and the company has a history of problems with correctly interfacing with libraries then it might be that you add a check list for making sure that library interfaces are looked at a little bit more deeply. And one, one other one is coverage of verification conditions. This is kind of a specialized, checklist. Verification conditions were invented as part of a clean room software engineering methodology by IBM. A verification condition is a rule that obtains in a particular situation. For example, when a loop is encountered during a code review, a specific verification condition is to examine whether the loop is guaranteed to terminate under all circumstances.

22 – Metrics

Reviews, like design, testing, and coding, are software development activities. As such, they can be measured to see how effective they are. The key statistics to compute are the following. First off, the review rate in lines of artifact reviewed per staff hour spent reviewing. Second is defect rate in the number of defects detected per staff hour spent reviewing. Then from those numbers the defect density is the defects per line of artifact. This can be used to indicate whether the process of producing that artifact is leaving too many defects in it. And then, process yield is computed by comparing the review detected defects to total defects, or how do you know what the total defects are? One way of getting a total defect includes those defects detected by other means, such as testing and those that are eventually reported by users of the delivered product.

23 – Process Data

Reviews are an early step in an organization's effort to improve the quality of it's products it produces. A more sophisticated step is to review the review process itself. That is, to collect data on the effectiveness of the reviews and use it to improve the review process. Among the data that might be collected are the following. What was the artifact being reviewed and at which stage of the development process does the review take place? What was the date and time of the review and how long did it last? Who were the participants and how much preparation time did they spend? How many issues were raised, how many of them turned out to be defects, what were their types, and what were their severities? Organizations can also collect subjective data by distributing post review effectiveness questionnaires to the participants. Finally, for large organizations, it makes sense to store this data in a database for aggregate analysis over time.

24 – Alternative Review Styles

There are many different styles of review. Here's a brief list of some notable ones. The class resources page points you to further information about them. What we've been talking about are sometimes called Fagan reviews. Fagan was an IBM researcher, her, who first studied them and proposed the structuring that we're, we're talking about. But a a more recent and alternative approach is called pair programming, which is part of extreme programming, an agile development method and this was studied by Laura, Laurie Williams. It suggests that review is part is performed synchronously with, with coding or developing of an artifact by having a partner looking over the coder's shoulder pointing out problems at the time that that coder is, is, is doing his or her job or that the designer is doing their design. Another alternative is what's called pass-around reviews. These are typically conducted by email. This is often applied in open source projects where the people aren't collocated and the source code management system alerts participants to take a look when a new change or new file has been

checked into the source control system. And in any of these approaches there's a possibility of applying tools. Tools might be differencers to compare versions the source control management system itself as I've, as I've indicated, and there're also program analyzer tools that can provide enforcement and feedback and examples here are Lint. You may have heard of the Lint program or CodeCheck, which is a plugin into Eclipse.

25 – Guidelines Participants

Reviews have become a common part of everyday software development practice. As such, much has been learned about how to perform them. What follows in this lesson are some dos and don'ts of gleaned from actual experience. We begin with some about the participants. First off, reviews should not be used for personnel eva, evaluations. If participants feel that what they say about someone else's code will affect their performance rating or salary, they may be reluctant to speak. For this reason managers should in general not attend review meetings unless they have participated technically in the production of the artifact being reviewed. Or they're outside of the participant's reporting hierarchy. Similarly, in general it is not a good idea to allow non-participant observers in review meetings as they can provide distractions. Don't treat a review as an opportunity for training new staff members. This will reduce the productivity of the review session. Instead, hold a separate meeting with the trainee. The author of the artifact being reviewed should not be the moderator or recording. These are specialized tasks that require full concentration. Some organizations also require that the author not be the reader.

26 – Guidelines Content

Here are some guidelines about which topic should be avoided during your review. If possible, avoid discussions of, of style. These can be a lot of fun, and there's a lot of flaming going on, but people have strong feelings and consistency is more important, that is, consistency of, of style, is more important that what particular style is emp, is employed. Avoid problem solving during review meetings. The goal of the meeting is to raise as many issues as, as possible. Solve the problems offline or at another meeting. Avoid use of the word you and any phrasing that might raise defensiveness.

27 – Guidelines Process

Here are some guidelines about the review process itself. First off, spread out the reviews over time. Performance degrades if reviews are too concentrated. A practical limit for the duration of a review is the smaller of 250 lines of code, or 2 hours of, of artifact reviewing. Each type of review should have its own criterion for thoroughness, which should be determined in advance of the meeting. You can, treat the review as a go, no go decision activity. That is, the review meeting should end with a decision about whether or not the artifact is ready for the next stage of the development process. In fact, some organizations require the reviewers to sign off on the acceptability of the artifact, thereby improving accountability.

28 – Effectiveness

Code reviews are an effective technique for detecting defects in artifacts. Collected data indicates that for typical formal review, as described above, between 70 and 90% of defects are found. That is, if we're talking about a code review of all the defects in that code, the review meetings are going to find between 70 and 90% if you have effective reviews. Of course, there is a cost which amounts, in typical situations, to between 10 and 20% of the total cost of development. This cost is largely due to the staff time involved in the preparation work and the

review meting itself. The above data notwithstanding, defect detection rates vary dramatically depending on the specific rule goals, artifact complexity and how effective the meanings are themselves.

29 – Other Costs and Benefits

In addition to the numeric data, there are several other benefits and costs to be aware of. Individual skills can be enhanced by looking at other people's artifacts. There's some evidence to suggests that lightweight reviews, those involving the author and a couple of other people doing desk checking, are more effective in finding defects per staff hour. Of course, this does not mean that more total defects are detected and when using them, it is more difficult to collect overall organizational data. Your organization is going to have to learn from trying out various approaches as to what's the most effective approach within that organization. You should also be aware of the so-called ego effect. That is, the knowledge that an author's artifact will be reviewed tends to improve the quality of the delivered artifact. That is if you're going to show what you're doing to somebody else, you're going to make it better. However, there's a danger of damaged egos when an author is confronted with his or her defects. Collectively, there is also what is called the big-brother effect, in which stress levels are raised because people know they are being watched and measured.

30 – Summary

The later that a software development problem is detected, the more expensive it is to fix it. Hence, we want to find problems as early as possible in the process. Reviews are a cost effective way to find problems in all kinds of artifacts, including design documents. Going further, we want to detect deep problems if we can. And reviews run the risk of revealing shallow problems with a document self or with superficial aspects of the design. To reveal deep problems requires exposure of the design to experienced designers who are among the most highly paid and time stressed people in the development process. Therefore it makes sense to have the design review be as focused and well-run as possible. To do this reviews should be institutionalized. By that I mean, they should be a formally defined process element and part of the corporate culture. By doing so, the short-term costs involved in conducting reviews will be more than offset by the eventual saving on reduced maintenance and increased customer satisfaction.

P5L1 Geeks in Black: The Code Review

01 – Introduction

Ladies and gentleman, I am Agent B.of MIB with an important announcement. Thousands of alien bugs have been invading our code. The invasion threatens our flight navigation systems. Our financial infrastructure and our ability to play Candy Crush. [MUSIC] To fight back, MIB has invented a secret weapon. The code review and assembled a crack team of reviewers gather in our headquarters in New York City. [NOISE] Gentlemen, would you now introduce yourself to the public, first our moderator. Hello, I am Moderator, sometimes also called the controller. My name is 3.8. I'm a professor in computer science, and they call me 3.8 because it took me 3.8 billion years to evolve. Next, our Reader. Hi, I'm the Reader, Dr. Bug. They call me Dr. Bug because I smash bugs. Here's our recorder. Hi, I'm Crazy Bob. They call me Crazy Bob because I'm crazy about Ada. I'm the tech lead on our avionics project. We delivered 1.4 million lines of safety-critical Ada code. Next we have Inspector Fra Elbertus. Hi, I'm Fra Elbertus. I'm the friar of doom. I was the originator of the sticky net virus when I consulted for one of those three letter government agencies. You haven't heard of sticky net because it was so evil, they had to redo it and call it Stuxnet. And our other inspector, Byte.Me. Hi everyone my name's Byte.Me. I used to have a back story, but I pseudo RN'd it years ago. I like coffee, code and Reddit. That's about it. Finally, I will also be an inspector. Now ladies and gentlemen, let me show you the first invasion site. A key piece of software infrastructure called BlankCount.java. [MUSIC]

02 – Part 1

Now I will turn things over to 3.8 to lead the attack against these bugs. Is everyone ready, Crazy Bob? I am ready, sir. Bite Me? Ready to go. Ready to roll. Doctor Bug? I was born ready. Oh, excellent. We are all ready here. So, let us start. If I could interject something before we begin [LAUGH] I just would like to say, before we started this project, I I advised management that we wouldn't be having a lot of these core problems that we've got now if we just use data. We're on job aid it would really, you know, it would solve a lot of our integration problems and all. And I just wanted to make that clear, that I told them before we began this project that we should do it, and here we are looking at jobs. So, I just wanted to make that clear before we got started. Well, I mean, with that, just trying to read this Baloney Code that this guy wrote is unbearable. Well I think we should just, I am just sick to my stomach. We should just proceed with the review probably. Oh, well, okay. Gentlemen, let me remind you that I'm the controller here. [LAUGH] Thank you very much for your advice. Dear reader, Doctor Bug, please talk. We start with two imports in the code. The first one is an import to the java.io package, all the classes in the package, and there's the right semicolon right after that. And there's another importer, which is to java.lang.System, which is used for the io, and another semicolon, so this looks fine. So line one, line one is actually not good coding style, because you should never import you're polluting the namespace. So we should never

import asterisk, we should only import the classes that we're actually using. Well, I think it's okay, all right, if we need to use multiple ones. Do you want to spell them all out. But then you're polluting, you're bringing in all these names that we really don't need. So, I think it should be, what? Java. I think the only IO class, let's see InputStream.read, so we should have java.io.InputStream.reader. I think you're, you're wasting space by doing this, I mean this is, why is it there? And thanks for telling us about the semicolons, that's a, a big help. Yeah I figured. Can I continue? Yes, please continue. So should I record that as defect or, or we don't believe that's a defect? I think you should I, personally, I gotta, I gotta echo Crazy Bob here because if we move to the next line here, purpose, we have the slash star here for comment style and then further on down we're going to use the slash slash, some consistency would be nice. So, I think we should record that as an error. But let's move on. That's bad style. Sure, sure. Okay, Crazy Bob will record it. Dr. Bug? Okay, fine. okay, then the main class starts, which is Before we do that, let me complain about line 2. I don't believe you have to import anything from java.lang, right? I believe java.lang comes in automatically. I believe that's correct. So that's a useless, that's a useless import. Okay, fine, that's what my mentor told me that I was supposed to do but that's okay. Okay, so then we start with the main classes. Maybe we should reconcile this common issue, before we move on. As brought up by me. I think so. I mean. That, that's just. Don't you have coding standards and which would include how to write comments in your, in your. I do and it's usually, just stick with one and then that's the way it goes for the rest of the, the comments. The slash-star has so much history with it. You know, we want, we want to reflect the fact that this is a historical artifact. I don't know if I agree I, I think not only that, I'm not sure what constitutes getting its own white space like an extra line here. We move down further maybe we come back to this but I have a couple places that I marked that I just don't quite understand. Maybe Bite me is saying, let's go one down and come back to this. Doctor, Bug please continue. Thank you.

03 – Part 2

The next line, line 6. The main class starts, which is called BlankCount. And [COUGH] after that there's two constants that are being defined. And one is the blank the other one is a sentinel. And then we move to the main, which is actually the main the main body of the class and also the main method for the program. Everything is included into a try catch block, and the first instruction is actually to initialize the InputStreamReader, ISR, with a new InputStreamReader and takes the system in stream as a parameter. So I believe on that line, there's also a problem because that violates code the interface rather than implementation, so they actually have the implementation class on the left, left-hand side. It should be declared as the ab-,. . as the abstract interface and then the concrete implementation on the right-hand side with the new. So, now, other people agree? Yeah, I agree. Yeah. Bob's right. Okay then there's the code declares two integers next and count, and next is going to be use the as the next character in sentence, and count is going to be used to count the number of blank counters, which is the main goal of the, of this class. I'm glad you stopped telling us that there's semicolons at the end of each line. It's that helps. Well I think it's important, correct? So before the semicolons though, shouldn't we initialize these values like we have the ISR variable or both? [CROSSTALK] Yeah we should initialize. All right, go. Should we decide whether to character or an int? The comment says character. That's true. [CROSSTALK] Operation says int. Yeah, that, that's excellent to pick that up. Yeah. That's that comment problem again. If you'd done slash star, maybe you would had written it right the first time. Should, should we go for comments on the right side of the? Or should we put them above? If they get

their own line, maybe it'd help us read it better. I don't know. I, this is, this is going back to, to line four up above. I, I don't, I'm not quite sure when we, when we want them beside the code, when we want them above the code? Well, there is nothing in the standard that we use in the company, but that's fine. I mean, if you want to pick a, you know, a way to do it We could make a standard. [CROSSTALK] That's fine. We can make a standard. We can make a standard right now. I'd like a standard, yeah, mm-hm. So what's, what's the standard? We going to go slash, slash all the way or slash star? Slash star seems a little more robust. We can, you know? You don't have to use as many slash slashes. Javadocs uses slash star kind of, that kind of structure when you do. I thought it was kind of old fashioned but that's okay. We can use slash star. What's wrong with old fashioned? [LAUGH]. [LAUGH]. Okay, Dr. Bud continue please.

04 – Part 3

Okay let me get back to where I was. Okay, so we declared the two variables, and then there's two print lines the first one Excuse me did, did the recorder correct the comments? To make, make note of the errors in the comments that, that Slash asterisk should our new standard coding as well. Yeah these easier being pointed out, that we should be, it's said that they're characters but they were declared as integers. For int next, it says next character is that it? Yeah. Put a line through. The implication is that you're confused about characters versus integers. Mm-hm, even though the representation was also signed. Okay so as I was saying there is two print statement, the first one prints to the user, enter a sentence ending with a period, and the second one says, follow each character buy a return. We have a typo. Yeah, I guess, buy should not really be buy, And also probably, follow, should be capitalized, right? I mean it's a separate sentence so you're trying to [COUGH] write it as a sentence. Or it should be all on one line, because you're, you're breaking at interest and it's ending with a period oh, okay, I see. But it also, it's interesting, Interest in it is ending with a period doesn't have a delimiter at the end where it's follow each character by a return does have a delimiter. Should that be, have a colon there? It seems like we're really getting stuck on minor details. I think so too, let's move on, let's move on back to Dr. Bug, please continue. So a character is, misspelled. It's charcter. Just making, I think it's making for bad user experience, we don't, we don't, you know Yeah, actually I think I was just copy and paste in here from someone elses code, but that's fine. Yeah, I think it's. [LAUGH] Is that what we do now? Is we copy, copy from other people's [CROSSTALK]. Well, it's just this, you know? This was [CROSSTALK]. Whose was it that we copied from? It was part of a log, I mean, I think it was actually Crazy Bob's code, but [LAUGH] I could be wrong. It wasn't my code because it's not in native. [LAUGH] Yeah. It was a native piece of code. In fact, if we could just back go back and revisit, 13 and 14, you know, this initialization problem wouldn't even be occurring because you can't have uninitialized data in aid. So if we'd actually done what we said, what I said originally, we wouldn't be having these problems. You, you're absolutely right, I forgot about that. Crazy Bob bring, brings up some excellent points, about this lousy code we're looking at. I have to say, actually, I'm pretty sure that in Java, when you declare a lock of variable like this, it gets initialized automatically to zero. But I might be wrong. That's the instance variables that are initialized, and the local variables are not initialized. Okay, maybe I'm wrong then Even if there are automatically initialized, good code behavior would be that you actually initialize them so everybody doesn't have to remember whether it works or not. Well, just, you know, I was used to developing data, so that's reality. Yeah [LAUGH]. Okay, let us continue Dr. Bug.

05 – Part 4

Okay, so after this there's a next gets the first character in the string using the read method. Also in this case, suitably followed by a semicolon. I hope that makes you happy. Oh, I'm very happy with that. That's great. And and then we print line in which we print this character that we just read. Okay, so, for, for 19 should we? This extra white space around next. This kind of goes back to, to line ten where we have some extra white space between the two brackets for, for the string. Is there a, what do you want to do there? We, we always want to have pad space around the functions, or the arguments into the function? Yeah, I guess that's something the editor data, that's not much fine with, you know, doing it with all the spaces. It's not consistent. [CROSSTALK]. Again, we need to have a standard of some kind. I think so. It's obvious you don't have any kind of coding standards. Yes. [CROSSTALK]. Most of this is copied code. [CROSSTALK]. Yeah, it was copy and paste. So, so a hodgepodge of, of different styles. So, this old one is kind of no, no empty spaces around those variables. And the no copying of code from, from random forums online. That might be a good idea. Is that okay Dr. Bug from now onwards? Isn't that, doesn't that mean we're going to have to re-code stuff that we might save energy by copying? Well maybe if as long we're copying internal code that has the, the standard, the standard applies. Maybe we need to write something that's a standard checker for our internal use on it. Dr. Bug? Okay, so after this we enter into a while loop with the given condition. And the, at that point once we get into the while looper we check if the next character that we read is a blank. Actually, we have a problem here, we should be stopping when the next doesn't equal the sentinel, otherwise we're just stopping maybe immediately. Do we ever enter this loop? Unless the first character is the . Our count, our count will be zero. We're not going to count the blanks if we, because we want to stopiat a period, correct? We're going to read the number of blanks in a sentence and stop. At least that's how I'm reading it. Oh, oh yeah sorry about that one. Yeah I was coding in a hurry. I'm sorry and it's yeah, you, you're right. This should be the, the while next is not a sentinel. And I, yeah and I have to apologize, because I kind of came through that when I was going through. I should have looked at that more carefully while doing the reading. Or maybe you just copied somebody's while loop and didn't even bother to check what it was about. Well, in this case I think it's actually my fault so, sorry about that. Oh. Cannot blame everything. We're so happy you apologized. Cannot blame everything on Bob. If we could just for a second revisit every other thing we've talked about previously. I just so, I'm supposed to be recording the severity, so I assume all the severity of everything up til just now is minor. Is everybody good? And this is our first really major severity. That sounds reasonable. Defect. Mm-hm. What are our severity levels? Well, according to our little log here, we only have major and minors. Okay. [LAUGH] So then, yeah, I would agree with that. I, some, some of the things that were tried to put in between but we don't have an in between, so I think you've got it right. It's the best we can do with what we've got.

06 – Part 5

So then, I guess you know, this, there is a bug there definitely. And then if next is equal to blank which is the character we want to count the count is incremented using the plus, plus operator. [COUGH] Then we read the next character again using the read. function, the read method on on the stream. And then it would print the character that we just read. Should we really be printing all these characters on a new line every time? Or do we want the sentence to actually look like the characters are one after another in a sentence? You have to say that was not specified. So I made that decision but. It didn't sound like a very good decision. I mean, you think you're going to have this, [LAUGH] this line of characters, vertical line of

characters coming out that you, then it's difficult for you to check if the sentence that I put in, that was the sentence, try to figure out you know, if it actually caught them all and, and, and- [CROSSTALK]. Kind of drives- Right now it's a vertical line of integers, it's not even characters. Yeah, that's true. So, it's- Even worse. Right. Yeah, you're right, okay. Okay, okay, that's fine. I didn't realize that, that was going to be printed as an integer. You're probably right. And so it should be really printed as a character. [CROSSTALK]. You didn't realize. I mean, you never even ran your code once on your test. I did run it I just, the input wasn't just that. It's like students who appear in science class. It just compile it doesn't work. I mean, this is one of the advantages of- [CROSSTALK]. [LAUGH]. This is actually, you know, one of the advantages of inspection. I mean, this code, is could just, part of a larger piece of code and so we're just reading it right now and didn't really. [COUGH]. Go through a serious testing phase. Yeah, I compiled. [COUGH]. To make sure there were no syntactic errors. We can continue from here. We can continue. Thank you. [COUGH] Okay, so at this point- We're going to get bite me to go see if he can compile it and see if it even compiles. I'm not even sure if it compiled yet. Well, it may have of compiled. I can't remember. The last time I checked I know it didn't work. It didn't give us what we wanted. It does compile though. I mean I'm positive about that. Okay, okay.

07 – Part 6

Okay. So now at this point that we exit from the loop, which means that we encounter the sentinel which is the, marks the end of the, of the sentence. [CROSSTALK]. Are we guaranteed to exit? Well- You know, we'll, w'll have to fix the condition that we identified. But assuming that we, we fixed it, what happens if last character is not a sentinel? Well, we tell the user, right, that he should follow, that he should enter sentence, end it with the period. That's right, the user are always right. With their responsibility, right? You have to take care of all the possibilities here. Well, as you might recall on a keyboard and a computer, the period and the comma are right next to each other, and frequently people punch the wrong one. It's called a slip in, in the cognitive world, and they put a comma in and then this thing loops forever. I was thinking, in fact, if there was an enda file without a, without a sentinel at the end. Even better. Yeah, I guess then what we'll get, an exception exit, right, and catching the exceptions, so. Without printing anything out? Well, you know, the, the user will figure out that they did wrong. We gave them instructions. I mean, I don't know. If we want to account for all the possible behaviors, sure, I mean, just you know, let's agree to do that. Hey, listen son, this is Aviaonic software we're building here and that means planes crash if we can't- Sure. Characters in a sentence. That's fine. You've kind of been on my back the whole time, but that's fine. I mean if, if you think we should account for the, for this thing. We'll, we'll account for that. Do you want to mark it down as as a problem? We might send- Crazy Bob, if you don't mind. Maybe we need to send this guy back to the IT department. I think we have read the point. I think we have read the point. Doctor Bug gets the point, right Doctor Bug? Yeah. I, I get the point. Yeah. There's no need to kind of reiterate every time like three or four times but, anyways. Okay so where were, was I

08 – Part 7

Okay, we exit the loop and, well, I'll take into account, I guess, if there is a point or if there's not a period at the end we might have an exception, we'll take that into account. Then the, the code brings a new line, and then- But for, for, for the 31, 31, the commenter, are we assuming that we were going to search something here with the, the use of the word assert or is that just, is that a,- Well I know it was more like a note for myself that the count is a a

blanks because that is the conditions that should be verified here, I mean I might, may make that into an assert later on. Weren't you for my own edification an amazing, these one, two, three, four, Four print lines in a row, tell me what you're expecting to see coming out of each of those. I'm just expecting to see well, some white space. That's about it. So the first one will be a blank plane. Yeah. Just, you know, to kind of put some distance between the list of. Okay, then the second one will. Let's just say the number of blank. Is. Is, blanks is and then print the number of blanks. Should be printing to a count. Yeah, no, you're right, it should. As I said, I mean, I was kind of kind of writing this in a hurry, and yeah, this should be, this should be count. And, it would be on the next line. Mm-hm. And, it would be on the next- It won't be width contiguous with the text. Yeah. Well hold up that, that's [CROSSTALK]. No, actually no. Oh, I'm sorry, print. [CROSSTALK] Couldn't we just combine all these into one. [CROSSTALK]. Yeah, I should probably, yeah. [CROSSTALK]. Couldn't we just combine all four of these into one? . Sure, we could use, you know, slash n and then, is slash n is Slash n. [CROSSTALK]. Sure, yeah, we could use slash n. Is that allowed in the kind of code standard that we use? You just told me there weren't any standards and so, yeah. This is just a little amusing. [CROSSTALK] much of, yeah. Mm-hm. Okay, so this can be connected, concatenated in one line, maybe, I thought that this was going to improve readability, but that's We don't have to use that. Maybe that's not, that's not the case. The other side of this, this paper. We have an extra log sheet. You going to run out of room? I got it. Okay, because,- I anticipated ahead that we would have many defects. Well, obviously, that's been the case. Maybe you're being too picky, but fine. It looks like dogs barks, repetition precedes it. [LAUGH]. Well let's continue for now.

09 – Part 8

'Kay. And then they know I catch any exception because we know that every time I, well, I'll just do it, okay? Do we mentioned that line 34 prints the wrong? Yeah. It prints that. [CROSSTALK] Prints next to the . Yeah. Yeah. Sorry. You need to make a note of that, though. Can we put? Who's checking the the recorder? [CROSSTALK] Crazy Bob. here? Major, major, our majors are what we had the not equal to sentinel. That needs to change. Right. We're not checking for a period or have the exception. We're printing every character on new line as an int. Okay, that's pretty good. There's questions about whether the loop even terminates in the network between the wrong variable. Okay. I shouldn't catch, can, are we going to, you know, catch beyond its own line or catch comes after, you know, on, on line 36 there. Should, should we, is there, do we want to talk about a standard for that? I personally like this style. I mean, the, I think this is the, the new style or whatever. I like this line. This where the, the opening curly brace is on the same line and then the close curly brace on . Actually, Crazy Bob and I talked about that a lot and we thought that, that was a, a nice way to put the code. And again, we wanted to define the standards as different we can do it in . Guess that mix-up coding were discussed between the two of you. Just . That's interesting. Well, we named- We were talking about how ADA, you know, ADA doesn't even use these curly braces. I mean, you, we could begins and ends and then it's really clear exactly where these blocks are beginning and ending. We wouldn't having all these crazy Java problems that we've got. Which you know Crazy Bob. It's from pre-Civil War era. So. Yeah. We, you, you need, need to be careful when you listen to him. You know, one thing that's bothering me. I keep looking at this thing, is that here's a piece of code, you're claiming that you read it. But there's no header in this code that says, you know, this is a Doctor Bug's code, the revision of the code. Any of that kind of information, which is typical in, in software engineering, and we know who's it is, how it's revised. If other people been making new versions of it, we know who

they are, what the dates of, of, of each revision, original development art, et cetera. None of that is in here. So that ten years from now the maintainers can call you up in the middle of the night and, and ask you what you were thinking? Could the number of bugs in this code is clear- [CROSSTALK]. Right, no Java down here. In this case it might be appropriate if you also put the sources of the different places you visited copy and pasted code. That might be. Okay, that was just a minor. I mean I really copy a coupled of things, not too much. [CROSSTALK] Oh. Okay. But that's sure. I mean I, I- Like the while statement that's incorrect. Mm-hm. Yeah. Yeah. Well, I think I told you before that the while statement is actually my mistake. [CROSSTALK]. The point has been made, very good to talk about continuity. Okay. So, at this point yeah. That the cases. We're going back to line 36, I believe it's improper to catch just a generic conception. You're supposed to catch the most specific catch you're actually expecting. Generic catching, catching generic conceptions is just catching generic conceptions. That's just to cover your behind kind of thing. Yeah. [CROSSTALK] Actually trying to catch IO exceptions. I believe there should be IO exception to the rule. [COUGH]. Yeah I was just trying to be comprehensive, because you know, this way an exception that happens will be caught. Sure. We need an exit-able system without printing anything is also really bad. I mean, we should be writing to the logger. In the case of where the exception was raised. Yeah, in the case where the exception was raised. I'd call that, definitely, a major. Both of them. I don't know if my colleagues agree with that or not. So, we need to catch specific, and then have a generic catch that follows? Is that what we're saying? We just catch the ones we were actually expecting. What about the ones that we don't expect? We're not, we won't get any information about those will we? Well you know if we were using eta they have a catch all at the end that we could always. [CROSSTALK]. There. Well enough with this eta stuff. [LAUGH]. I think we are using Java and so we'll have to deal with that. Again I think we should make that also part of the quoting standards.

10 – Part 9

Are we debugging code here or are we building coding standards? Are we building coding standards or debugging code? I dunno. It seems to me that I'm being blamed on a lot of things, that just you know, depending on the fact of having a standard or not but it's . Research says building standards significantly reduces the number of errors in software. And so, we should. Make a note that there are no standards existing in this group, and that they ought to be done. Here's some suggestions, but that's a bigger problem than what we're addressing here. [COUGH] While you're starting it, let's continue it Doctor Vaughn. At this point we just you know, there's just brackets to c lose catch statement And then the method, and then the classes, and then that's it, the codings. What is the exit status if everything runs correctly? Oh, it just says it's fine. Shouldn't we be setting an exit status of zero. I think that's probably by default or something. Is it? I don't know, I'm not so sure. I don't know I thought it was by default. That's what told me. Yeah, but- [CROSSTALK]. That was for in our job. So, you have to be. [LAUGH]. Well, while we're [COUGH] mention printing to a log. Shouldn't we, for the, when we're printing the characters in the while loop. Shouldn't that, isn't that debug information. Does the user even need to see that? They just really need to see the count, right? Shall we be worried about what's being print to kind of the logger and then what's printed to standard out? Well, I see that you're kind of changing the specs on me but yeah. If there has to be a logger then yeah the model will have to change. Well, it says right here, the purpose. Count the number of blanks in a sentence. Right, I know. It doesn't say a purpose. Write out all of the things. Hm. You know, as you're going along. So is the comment wrong or the spec wrong? Well, it's a, that's an excellent question. I would say that the I believed what I read

there. But I could be incorrect. Yeah, that's the spec I got. Since this is a program without a specification. I think he just invented it, you know, just. [CROSSTALK] So he said I need to do this. He said, okay I can do that. [CROSSTALK] I can copy code from various places and hack this thing together. This way it was nice for the users to see what but it doesn't have to be the case. I don't know what kind of users you've been around. Oh, you know, users that can put a period at the end of the sentence. [CROSSTALK]. We did and Dr. Bug, anything else for this? That's sound perfectly good.

11 – Summary

Crazy Bob will you please summarize all the bugs have you found? All right. So, I'll read. I'll read basically the type, the severity, the location and then the description. So, this was wrong. Minor, line one, don't import asterisk. Wrong, minor, line two, useless import or useless import of .java.lang. Stylistic minor, line 12. Program to interfaces not implementations. Missing minor. Lines 13 and 14, no initialization of variables. Stylistic minor 13 and 14 integer versus character. Style minor lines four, seven and eight. We discussed all about our standard coding style and comments should be slash, slash asterisk. No, shouldn't. I'm sorry. I kind of agree with said that we should go with, maybe, a minor plus plus or some kind of other severity. If, if styling is, is important as we kind of talked about maybe that should be somewhere between major and minor. Okay. Okay. Really need to crack down on that. Make sure that one just doesn't get washed away. Yeah. And also before you go to the next one Crazy Bob, please read the meaning of the redispositions again. So the first one is what? The first one is the line number. The second sources. Yeah. So our line numbers are four, seven and eight. That's where the comments were. No, I wasn't being clear. On the form? They're on the forum. Oh first, oh this one. This is just the number. Okay. Like consecutive number. All right. Of the defect that we found. Then the type, so there's missing, wrong, extra usability, performance, style, clarity or question. And then there's the severity, either major minor, and then we can add plus plusses, or minus minuses if you'd like. And then it'll locate the actual line number of the code. So then we have style minor line 16. Follow should be in capital letters. And character was misspelled and buy was misspelled. And then stylistic minor line 19 inconsistent space in white space around, and then we talked again about needing a some kind of coding standard for white space. then, wrong major plus. On line 20 the equal equal should be not equal per our loop condition. Then wrong again, major lines 19 and 28, we're printing every character on a new line as an integer. Then stylistic major line 20. We had a question about whether this loop ever even terminates, and we should be checking for some kind of enda file. Stylistic minor line 31. The assert comment is confusing. Stylistic minor lines 32 through 35. We should catonate all that system out into one line. [COUGH] Then wrong implementation major line 34. We're printing the wrong variable. Style again, minor everywhere is our brace placement standard. We need to come up with something, some kind of standard. Mine are plus plus maybe I don't know. Then another plus plus. And then style minor everywhere, there's no java.doc author tags or any other information on the code. Stylistic major line 36, we should catch specific exceptions rather than generic exception. Stylistic major, line 37, we should log exceptions instead of just exiting immediately. Stylistic, again major, line 39, we need an exit status on our success. Stylistic minor, lines 19 and 28, we should not be printing debug information to the console. And then I added another important one that I think is important, major everywhere our implementation should've been in Ada. Oh, that's a minority report. [LAUGH] Very good. So, I think the code passes, right? Thank you team. I can certainly breathe easier now that we have squashed these bugs. Ladies and

gentlemen, I'm afraid I'm going to have to neuralize you to protect the identities of our team members. [MUSIC]

12 – Credits

[MUSIC]