OO++ DESIGN PATTERNS: GOF REVISITED

by

Seth James Nielson

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Brigham Young University
December 2004
GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Seth James Nielson

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Dr. Charles D. Knutson, Chair

Date

Dr. Bryan S. Morse

Date

Dr. Dan Ventura
As chair of the candidate’s graduate committee, I have read the thesis of Seth James Nielson in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Dr. Charles D. Knutson
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences
ABSTRACT

OO++ DESIGN PATTERNS: GOF REVISITED

Seth James Nielson
Department of Computer Science
Master of Science

Programming languages and the programming paradigms they embody co-evolve over time. In many circles, for example, object-oriented programming has evolved from and effectively replaced imperative programming. More recently, many object-oriented languages have assimilated features from other programming paradigms, evolving into multiparadigm languages we refer to as “object-oriented plus-plus” or OO++. However, language evolution, like that seen in OO++, weakens the design-implementation interface introducing what we call “design dysphasia”—a partial disability in the use of programming language because of incongruous design methods. Design dysphasia persists until design methods are extended to match evolved language features.

One popular contemporary design method is the use of software design patterns. These patterns capture elements of design that can be reused within a specific context. When the programming languages that are part of pattern context evolve, patterns must adapt to the language changes. Otherwise they may reinforce design dyspha-
sia in the practitioner. Because of this, the current pattern maintenance model of “capture/recapture” is suboptimal.

This thesis presents an investigation of the shift in contemporary object-oriented languages to OO++ and analyzes the characteristics of the OO++ paradigm. The nature of design dysphasia is defined and discussed generally and in the context of software design patterns. A “capture/modify/recapture” maintenance model is presented as an effective replacement to the “capture/recapture” cycle. A concrete “modify” phase is defined for the adaptation of existing object-oriented patterns to OO++ languages illustrated through the adaptation of the 23 patterns presented in Design Patterns by Gamma et al. to OO++ variants.
Words are not sufficient to express the gratitude I feel for my wife Amy who has supported me in this academic “adventure” (as we call it).

I am also very grateful for my advisor, Dr. Knutson, who has spent countless hours teaching me about research, technical writing, and esoteric multiparadigm theory.

Most importantly, I am grateful to my Heavenly Father on whom I am dependent my intellectual talents, my sweet wife, and for guiding me in choosing Dr. Knutson as my advisor.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Background and Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 Design Patterns and Design Dysphasia</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Thesis Organization</td>
<td>6</td>
</tr>
<tr>
<td>2 Introducing the Object-Oriented Plus Plus (OO++) Paradigm</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>8</td>
</tr>
<tr>
<td>2.2 A Survey of OO++ Languages</td>
<td>9</td>
</tr>
<tr>
<td>2.3 OO++ Python</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Key Aspects of the OO++ Paradigm</td>
<td>15</td>
</tr>
<tr>
<td>2.4.1 Assumptions</td>
<td>17</td>
</tr>
<tr>
<td>2.4.2 Concepts</td>
<td>17</td>
</tr>
<tr>
<td>2.4.3 Values</td>
<td>19</td>
</tr>
<tr>
<td>2.4.4 Practices</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Conclusions</td>
<td>21</td>
</tr>
<tr>
<td>2.6 Appendix: Functional Extensions in Python</td>
<td>25</td>
</tr>
<tr>
<td>2.6.1 Python Functional Background</td>
<td>25</td>
</tr>
<tr>
<td>2.6.2 Higher-Order Function Extensions</td>
<td>26</td>
</tr>
<tr>
<td>2.6.3 Self-Referential Lambda Calls</td>
<td>27</td>
</tr>
<tr>
<td>2.7 Appendix: Logic Programming Extensions in Python</td>
<td>28</td>
</tr>
<tr>
<td>2.7.1 Python Logic Background</td>
<td>28</td>
</tr>
<tr>
<td>2.7.2 Defining Logic Programming Facts</td>
<td>28</td>
</tr>
<tr>
<td>2.7.3 Defining Logic Programming Inferences</td>
<td>29</td>
</tr>
<tr>
<td>2.7.4 Logic Programming Queries</td>
<td>30</td>
</tr>
<tr>
<td>2.7.5 Special Operations</td>
<td>31</td>
</tr>
<tr>
<td>2.7.6 Performance</td>
<td>32</td>
</tr>
</tbody>
</table>
# Section

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>**5  **</td>
<td>OO++ Builder Extensions 85</td>
</tr>
<tr>
<td>5.1 Base Pattern - Extension Name</td>
<td>85</td>
</tr>
<tr>
<td>5.2 Motivation</td>
<td>85</td>
</tr>
<tr>
<td>5.3 Forces</td>
<td>85</td>
</tr>
<tr>
<td>5.4 Applicability</td>
<td>86</td>
</tr>
<tr>
<td>5.5 OO++ Interactions</td>
<td>86</td>
</tr>
<tr>
<td>5.6 Structure</td>
<td>86</td>
</tr>
<tr>
<td>5.7 Extended Participants</td>
<td>86</td>
</tr>
<tr>
<td>5.8 Collaborations</td>
<td>87</td>
</tr>
<tr>
<td>5.9 Consequences</td>
<td>87</td>
</tr>
<tr>
<td>5.10 Implementation</td>
<td>88</td>
</tr>
<tr>
<td>5.11 Sample Code</td>
<td>89</td>
</tr>
<tr>
<td>5.12 Related Patterns</td>
<td>91</td>
</tr>
<tr>
<td>**6  **</td>
<td>OO++ Factory Method Extensions 93</td>
</tr>
<tr>
<td>6.1 Base Pattern - Extension Name</td>
<td>93</td>
</tr>
<tr>
<td>6.2 Motivation</td>
<td>93</td>
</tr>
<tr>
<td>6.3 Forces</td>
<td>94</td>
</tr>
<tr>
<td>6.4 Applicability</td>
<td>94</td>
</tr>
<tr>
<td>6.5 OO++ Interactions</td>
<td>94</td>
</tr>
<tr>
<td>6.6 Structure</td>
<td>95</td>
</tr>
<tr>
<td>6.7 Extended Participants</td>
<td>95</td>
</tr>
<tr>
<td>6.8 Collaborations</td>
<td>95</td>
</tr>
<tr>
<td>6.9 Consequences</td>
<td>95</td>
</tr>
<tr>
<td>6.10 Implementation</td>
<td>96</td>
</tr>
<tr>
<td>6.11 Sample Code</td>
<td>96</td>
</tr>
<tr>
<td>6.12 Related Patterns</td>
<td>98</td>
</tr>
<tr>
<td>**7  **</td>
<td>OO++ Prototype Extensions 99</td>
</tr>
<tr>
<td>7.1 OO++ Prototype Discussion</td>
<td>99</td>
</tr>
<tr>
<td>**8  **</td>
<td>OO++ Singleton Extensions 101</td>
</tr>
<tr>
<td>8.1 OO++ Singleton Discussion</td>
<td>101</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9  <strong>OO++ Adapter Extensions</strong></td>
<td>103</td>
</tr>
<tr>
<td>9.1 Base Pattern - Extension Name</td>
<td>103</td>
</tr>
<tr>
<td>9.2 Motivation</td>
<td>103</td>
</tr>
<tr>
<td>9.3 Forces</td>
<td>103</td>
</tr>
<tr>
<td>9.4 Applicability</td>
<td>104</td>
</tr>
<tr>
<td>9.5 OO++ Interactions</td>
<td>104</td>
</tr>
<tr>
<td>9.6 Structure</td>
<td>104</td>
</tr>
<tr>
<td>9.7 Extended Participants</td>
<td>104</td>
</tr>
<tr>
<td>9.8 Collaborations</td>
<td>105</td>
</tr>
<tr>
<td>9.9 Consequences</td>
<td>105</td>
</tr>
<tr>
<td>9.10 Implementation</td>
<td>105</td>
</tr>
<tr>
<td>9.11 Sample Code</td>
<td>106</td>
</tr>
<tr>
<td>9.12 Related Patterns</td>
<td>107</td>
</tr>
</tbody>
</table>

| 10 **OO++ Bridge Extensions** | 109  |
| 10.1 Base Pattern - Extension Name | 109  |
| 10.2 Motivation                | 109  |
| 10.3 Forces                    | 110  |
| 10.4 Applicability             | 110  |
| 10.5 OO++ Interactions         | 110  |
| 10.6 Structure                 | 111  |
| 10.7 Extended Participants     | 111  |
| 10.8 Collaborations            | 112  |
| 10.9 Consequences              | 112  |
| 10.10 Implementation           | 113  |
| 10.11 Sample Code              | 113  |
| 10.12 Related Patterns         | 116  |

<p>| 11 <strong>OO++ Composite Extensions</strong> | 117  |
| 11.1 Base Pattern - Extension Name | 117  |
| 11.2 Motivation                  | 117  |
| 11.3 Forces                      | 117  |
| 11.4 Applicability               | 117  |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.11 Sample Code</td>
<td>133</td>
</tr>
<tr>
<td>13.12 Related Patterns</td>
<td>135</td>
</tr>
<tr>
<td><strong>14 OO++ Flyweight Extensions</strong></td>
<td>137</td>
</tr>
<tr>
<td>14.1 Base Pattern - Extension Name</td>
<td>137</td>
</tr>
<tr>
<td>14.2 Motivation</td>
<td>137</td>
</tr>
<tr>
<td>14.3 Forces</td>
<td>138</td>
</tr>
<tr>
<td>14.4 Applicability</td>
<td>139</td>
</tr>
<tr>
<td>14.5 OO++ Interactions</td>
<td>139</td>
</tr>
<tr>
<td>14.6 Structure</td>
<td>139</td>
</tr>
<tr>
<td>14.7 Extended Participants</td>
<td>139</td>
</tr>
<tr>
<td>14.8 Collaborations</td>
<td>140</td>
</tr>
<tr>
<td>14.9 Consequences</td>
<td>140</td>
</tr>
<tr>
<td>14.10 Implementation</td>
<td>140</td>
</tr>
<tr>
<td>14.11 Sample Code</td>
<td>141</td>
</tr>
<tr>
<td>14.12 Related Patterns</td>
<td>141</td>
</tr>
<tr>
<td><strong>15 OO++ Proxy Extensions</strong></td>
<td>143</td>
</tr>
<tr>
<td>15.1 Base Pattern - Extension Name</td>
<td>143</td>
</tr>
<tr>
<td>15.2 Motivation</td>
<td>143</td>
</tr>
<tr>
<td>15.3 Forces</td>
<td>143</td>
</tr>
<tr>
<td>15.4 Applicability</td>
<td>143</td>
</tr>
<tr>
<td>15.5 OO++ Interactions</td>
<td>143</td>
</tr>
<tr>
<td>15.6 Structure</td>
<td>143</td>
</tr>
<tr>
<td>15.7 Extended Participants</td>
<td>144</td>
</tr>
<tr>
<td>15.8 Collaborations</td>
<td>144</td>
</tr>
<tr>
<td>15.9 Consequences</td>
<td>144</td>
</tr>
<tr>
<td>15.10 Implementation</td>
<td>144</td>
</tr>
<tr>
<td>15.11 Sample Code</td>
<td>145</td>
</tr>
<tr>
<td>15.12 Related Patterns</td>
<td>145</td>
</tr>
<tr>
<td><strong>16 OO++ Chain of Responsibility Extensions</strong></td>
<td>147</td>
</tr>
<tr>
<td>16.1 Base Pattern - Extension Name</td>
<td>147</td>
</tr>
<tr>
<td>16.2 Motivation</td>
<td>147</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>16.3 Forces</td>
<td>147</td>
</tr>
<tr>
<td>16.4 Applicability</td>
<td>147</td>
</tr>
<tr>
<td>16.5 OO++ Interactions</td>
<td>148</td>
</tr>
<tr>
<td>16.6 Structure</td>
<td>148</td>
</tr>
<tr>
<td>16.7 Extended Participants</td>
<td>149</td>
</tr>
<tr>
<td>16.8 Collaborations</td>
<td>149</td>
</tr>
<tr>
<td>16.9 Consequences</td>
<td>149</td>
</tr>
<tr>
<td>16.10 Implementation</td>
<td>149</td>
</tr>
<tr>
<td>16.11 Sample Code</td>
<td>150</td>
</tr>
<tr>
<td>16.12 Related Patterns</td>
<td>151</td>
</tr>
<tr>
<td>17 OO++ Command Extensions</td>
<td>153</td>
</tr>
<tr>
<td>17.1 Base Pattern - Extension Name</td>
<td>153</td>
</tr>
<tr>
<td>17.2 Motivation</td>
<td>153</td>
</tr>
<tr>
<td>17.3 Forces</td>
<td>153</td>
</tr>
<tr>
<td>17.4 Applicability</td>
<td>153</td>
</tr>
<tr>
<td>17.5 OO++ Interactions</td>
<td>153</td>
</tr>
<tr>
<td>17.6 Structure</td>
<td>154</td>
</tr>
<tr>
<td>17.7 Extended Participants</td>
<td>154</td>
</tr>
<tr>
<td>17.8 Collaborations</td>
<td>154</td>
</tr>
<tr>
<td>17.9 Consequences</td>
<td>154</td>
</tr>
<tr>
<td>17.10 Implementation</td>
<td>155</td>
</tr>
<tr>
<td>17.11 Sample Code</td>
<td>155</td>
</tr>
<tr>
<td>17.12 Related Patterns</td>
<td>156</td>
</tr>
<tr>
<td>18 OO++ Interpreter Extensions</td>
<td>157</td>
</tr>
<tr>
<td>18.1 Base Pattern - Extension Name</td>
<td>157</td>
</tr>
<tr>
<td>18.2 Motivation</td>
<td>157</td>
</tr>
<tr>
<td>18.3 Forces</td>
<td>157</td>
</tr>
<tr>
<td>18.4 Applicability</td>
<td>157</td>
</tr>
<tr>
<td>18.5 OO++ Interactions</td>
<td>157</td>
</tr>
<tr>
<td>18.6 Structure</td>
<td>158</td>
</tr>
<tr>
<td>18.7 Extended Participants</td>
<td>158</td>
</tr>
<tr>
<td>18.8 Collaborations</td>
<td>158</td>
</tr>
</tbody>
</table>
19 OO++ Iterator Extensions

19.1 Base Pattern - Extension Name

19.2 Motivation

19.3 Forces

19.4 Applicability

19.5 OO++ Interactions

19.6 Structure

19.7 Extended Participants

19.8 Collaborations

19.9 Consequences

19.10 Implementation

19.11 Sample Code

19.12 Related Patterns

19.13 Base Pattern - Extension Name

19.14 Motivation

19.15 Forces

19.16 Applicability

19.17 OO++ Interactions

19.18 Structure

19.19 Extended Participants

19.20 Collaborations

19.21 Consequences

19.22 Implementation

19.23 Sample Code

19.24 Related Patterns

20 OO++ Mediator Extensions

20.1 OO++ Mediator Discussion
### 21 OO++ Memento Extensions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1 Base Pattern - Extension Name</td>
<td>175</td>
</tr>
<tr>
<td>21.2 Motivation</td>
<td>175</td>
</tr>
<tr>
<td>21.3 Forces</td>
<td>176</td>
</tr>
<tr>
<td>21.4 Applicability</td>
<td>176</td>
</tr>
<tr>
<td>21.5 OO++ Interactions</td>
<td>176</td>
</tr>
<tr>
<td>21.6 Structure</td>
<td>176</td>
</tr>
<tr>
<td>21.7 Extended Participants</td>
<td>177</td>
</tr>
<tr>
<td>21.8 Collaborations</td>
<td>177</td>
</tr>
<tr>
<td>21.9 Consequences</td>
<td>177</td>
</tr>
<tr>
<td>21.10 Implementation</td>
<td>177</td>
</tr>
<tr>
<td>21.11 Sample Code</td>
<td>178</td>
</tr>
<tr>
<td>21.12 Related Patterns</td>
<td>179</td>
</tr>
</tbody>
</table>

### 22 OO++ Observer Extensions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1 Base Pattern - Extension Name</td>
<td>181</td>
</tr>
<tr>
<td>22.2 Motivation</td>
<td>181</td>
</tr>
<tr>
<td>22.3 Forces</td>
<td>181</td>
</tr>
<tr>
<td>22.4 Applicability</td>
<td>181</td>
</tr>
<tr>
<td>22.5 OO++ Interactions</td>
<td>181</td>
</tr>
<tr>
<td>22.6 Structure</td>
<td>182</td>
</tr>
<tr>
<td>22.7 Extended Participants</td>
<td>182</td>
</tr>
<tr>
<td>22.8 Collaborations</td>
<td>182</td>
</tr>
<tr>
<td>22.9 Consequences</td>
<td>183</td>
</tr>
<tr>
<td>22.10 Implementation</td>
<td>183</td>
</tr>
<tr>
<td>22.11 Sample Code</td>
<td>183</td>
</tr>
<tr>
<td>22.12 Related Patterns</td>
<td>184</td>
</tr>
</tbody>
</table>

### 23 OO++ State Extensions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.1 Base Pattern - Extension Name</td>
<td>187</td>
</tr>
<tr>
<td>23.2 Motivation</td>
<td>187</td>
</tr>
<tr>
<td>23.3 Forces</td>
<td>187</td>
</tr>
<tr>
<td>23.4 Applicability</td>
<td>188</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>23.5  OO++ Interactions</td>
<td>188</td>
</tr>
<tr>
<td>23.6  Structure</td>
<td>188</td>
</tr>
<tr>
<td>23.7  Extended Participants</td>
<td>188</td>
</tr>
<tr>
<td>23.8  Collaborations</td>
<td>189</td>
</tr>
<tr>
<td>23.9  Consequences</td>
<td>189</td>
</tr>
<tr>
<td>23.10 Implementation</td>
<td>189</td>
</tr>
<tr>
<td>23.11 Sample Code</td>
<td>189</td>
</tr>
<tr>
<td>23.12 Related Patterns</td>
<td>190</td>
</tr>
<tr>
<td>24 OO++ Template Method Extensions</td>
<td>191</td>
</tr>
<tr>
<td>24.1  Base Pattern - Extension Name</td>
<td>191</td>
</tr>
<tr>
<td>24.2  Motivation</td>
<td>191</td>
</tr>
<tr>
<td>24.3  Forces</td>
<td>191</td>
</tr>
<tr>
<td>24.4  Applicability</td>
<td>191</td>
</tr>
<tr>
<td>24.5  OO++ Interactions</td>
<td>192</td>
</tr>
<tr>
<td>24.6  Structure</td>
<td>192</td>
</tr>
<tr>
<td>24.7  Extended Participants</td>
<td>192</td>
</tr>
<tr>
<td>24.8  Collaborations</td>
<td>192</td>
</tr>
<tr>
<td>24.9  Consequences</td>
<td>192</td>
</tr>
<tr>
<td>24.10 Implementation</td>
<td>193</td>
</tr>
<tr>
<td>24.11 Sample Code</td>
<td>193</td>
</tr>
<tr>
<td>24.12 Related Patterns</td>
<td>193</td>
</tr>
<tr>
<td>25 OO++ Visitor Extensions</td>
<td>195</td>
</tr>
<tr>
<td>25.1  OO++ Visitor Discussion</td>
<td>195</td>
</tr>
<tr>
<td>26 Conclusion</td>
<td>197</td>
</tr>
<tr>
<td>26.1  OO++</td>
<td>198</td>
</tr>
<tr>
<td>26.2  Design Dysphasia</td>
<td>199</td>
</tr>
<tr>
<td>26.3  Design Patterns</td>
<td>199</td>
</tr>
<tr>
<td>26.4  Guiding Principles and Meta Patterns</td>
<td>200</td>
</tr>
<tr>
<td>26.5  Future Work</td>
<td>205</td>
</tr>
</tbody>
</table>
### A Pattern Redesign Documents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>213</td>
</tr>
<tr>
<td>A.1.1</td>
<td>213</td>
</tr>
<tr>
<td>A.1.2</td>
<td>214</td>
</tr>
<tr>
<td>A.1.3</td>
<td>215</td>
</tr>
<tr>
<td>A.2</td>
<td>216</td>
</tr>
<tr>
<td>A.2.1</td>
<td>216</td>
</tr>
<tr>
<td>A.2.2</td>
<td>217</td>
</tr>
<tr>
<td>A.3</td>
<td>218</td>
</tr>
<tr>
<td>A.3.1</td>
<td>218</td>
</tr>
<tr>
<td>A.3.2</td>
<td>218</td>
</tr>
<tr>
<td>A.3.3</td>
<td>219</td>
</tr>
<tr>
<td>A.4</td>
<td>221</td>
</tr>
<tr>
<td>A.4.1</td>
<td>221</td>
</tr>
<tr>
<td>A.4.2</td>
<td>222</td>
</tr>
<tr>
<td>A.5</td>
<td>222</td>
</tr>
<tr>
<td>A.5.1</td>
<td>222</td>
</tr>
<tr>
<td>A.5.2</td>
<td>223</td>
</tr>
<tr>
<td>A.5.3</td>
<td>224</td>
</tr>
<tr>
<td>A.6</td>
<td>226</td>
</tr>
<tr>
<td>A.6.1</td>
<td>226</td>
</tr>
<tr>
<td>A.7</td>
<td>226</td>
</tr>
<tr>
<td>A.7.1</td>
<td>226</td>
</tr>
<tr>
<td>A.7.2</td>
<td>227</td>
</tr>
<tr>
<td>A.7.3</td>
<td>228</td>
</tr>
<tr>
<td>A.8</td>
<td>229</td>
</tr>
<tr>
<td>A.9</td>
<td>229</td>
</tr>
<tr>
<td>A.9.1</td>
<td>229</td>
</tr>
<tr>
<td>A.9.2</td>
<td>230</td>
</tr>
<tr>
<td>A.9.3</td>
<td>230</td>
</tr>
<tr>
<td>A.10</td>
<td>231</td>
</tr>
<tr>
<td>A.11</td>
<td>231</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A.11.1 Metadata</td>
<td>231</td>
</tr>
<tr>
<td>A.11.2 Concrete Components</td>
<td>232</td>
</tr>
<tr>
<td>A.11.3 Metadata/Concrete Component Mapping</td>
<td>233</td>
</tr>
<tr>
<td>A.12 Adapter (Object Adapter Pattern Redesign)</td>
<td>234</td>
</tr>
<tr>
<td>A.12.1 Component Modification Possibilities</td>
<td>234</td>
</tr>
<tr>
<td>A.13 Analysis of the GOF Bridge Pattern</td>
<td>235</td>
</tr>
<tr>
<td>A.13.1 Metadata</td>
<td>235</td>
</tr>
<tr>
<td>A.13.2 Concrete Components</td>
<td>235</td>
</tr>
<tr>
<td>A.13.3 Metadata/Concrete Component Mapping</td>
<td>236</td>
</tr>
<tr>
<td>A.14 Bridge Pattern Redesign</td>
<td>238</td>
</tr>
<tr>
<td>A.14.1 New Component Possibilities</td>
<td>238</td>
</tr>
<tr>
<td>A.15 Analysis of the GOF Composite Pattern</td>
<td>238</td>
</tr>
<tr>
<td>A.15.1 Metadata</td>
<td>239</td>
</tr>
<tr>
<td>A.15.2 Concrete Components</td>
<td>239</td>
</tr>
<tr>
<td>A.15.3 Metadata/Concrete Component Mapping</td>
<td>240</td>
</tr>
<tr>
<td>A.16 Composite Pattern Redesign</td>
<td>242</td>
</tr>
<tr>
<td>A.16.1 Component Modification Possibilities</td>
<td>242</td>
</tr>
<tr>
<td>A.16.2 New Component Possibilities</td>
<td>242</td>
</tr>
<tr>
<td>A.17 Analysis of the GOF Decorator Pattern Pattern</td>
<td>242</td>
</tr>
<tr>
<td>A.17.1 Metadata</td>
<td>243</td>
</tr>
<tr>
<td>A.17.2 Concrete Components</td>
<td>243</td>
</tr>
<tr>
<td>A.17.3 Metadata/Concrete Component Mapping</td>
<td>244</td>
</tr>
<tr>
<td>A.18 Decorator Pattern Redesign</td>
<td>246</td>
</tr>
<tr>
<td>A.18.1 Component Modification Possibilities</td>
<td>246</td>
</tr>
<tr>
<td>A.19 Analysis of the GOF Facade Pattern Pattern</td>
<td>247</td>
</tr>
<tr>
<td>A.19.1 Metadata</td>
<td>247</td>
</tr>
<tr>
<td>A.19.2 Concrete Components</td>
<td>248</td>
</tr>
<tr>
<td>A.19.3 Metadata/Concrete Component Mapping</td>
<td>248</td>
</tr>
<tr>
<td>A.20 Facade Pattern Redesign</td>
<td>249</td>
</tr>
<tr>
<td>A.20.1 Component Modification Possibilities</td>
<td>249</td>
</tr>
<tr>
<td>A.21 Analysis of the GOF Flyweight Pattern</td>
<td>249</td>
</tr>
<tr>
<td>A.21.1 Metadata</td>
<td>249</td>
</tr>
<tr>
<td>A.21.2 Concrete Components</td>
<td>250</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>A.21.3 Metadata/Concrete Component Mapping</td>
<td>251</td>
</tr>
<tr>
<td>A.22 Flyweight Pattern Redesign</td>
<td>254</td>
</tr>
<tr>
<td>A.22.1 New Component Possibilities</td>
<td>254</td>
</tr>
<tr>
<td>A.23 Analysis of the GOF Proxy Pattern</td>
<td>254</td>
</tr>
<tr>
<td>A.23.1 Metadata</td>
<td>254</td>
</tr>
<tr>
<td>A.23.2 Concrete Components</td>
<td>255</td>
</tr>
<tr>
<td>A.23.3 Metadata/Concrete Component Mapping</td>
<td>256</td>
</tr>
<tr>
<td>A.24 Proxy Pattern Redesign</td>
<td>257</td>
</tr>
<tr>
<td>A.24.1 Component Modification Possibilities</td>
<td>257</td>
</tr>
<tr>
<td>A.25 Analysis of the GOF Chain of Responsibility Pattern</td>
<td>257</td>
</tr>
<tr>
<td>A.25.1 Metadata</td>
<td>257</td>
</tr>
<tr>
<td>A.25.2 Concrete Components</td>
<td>258</td>
</tr>
<tr>
<td>A.25.3 Metadata/Concrete Component Mapping</td>
<td>259</td>
</tr>
<tr>
<td>A.26 Chain of Responsibility Pattern Redesign</td>
<td>259</td>
</tr>
<tr>
<td>A.26.1 Component Modification Possibilities</td>
<td>259</td>
</tr>
<tr>
<td>A.26.2 New Component Possibilities</td>
<td>260</td>
</tr>
<tr>
<td>A.27 Analysis of the GOF Command Pattern</td>
<td>260</td>
</tr>
<tr>
<td>A.27.1 Metadata</td>
<td>261</td>
</tr>
<tr>
<td>A.27.2 Concrete Components</td>
<td>261</td>
</tr>
<tr>
<td>A.27.3 Metadata/Concrete Component Mapping</td>
<td>262</td>
</tr>
<tr>
<td>A.28 Command Pattern Redesign</td>
<td>263</td>
</tr>
<tr>
<td>A.28.1 Component Modification Possibilities</td>
<td>263</td>
</tr>
<tr>
<td>A.29 Analysis of the GOF Interpreter Pattern</td>
<td>263</td>
</tr>
<tr>
<td>A.29.1 Metadata</td>
<td>263</td>
</tr>
<tr>
<td>A.29.2 Concrete Components</td>
<td>264</td>
</tr>
<tr>
<td>A.29.3 Metadata/Concrete Component Mapping</td>
<td>265</td>
</tr>
<tr>
<td>A.30 Interpreter Pattern Redesign</td>
<td>266</td>
</tr>
<tr>
<td>A.30.1 New Component Possibilities</td>
<td>266</td>
</tr>
<tr>
<td>A.30.2 Metadata</td>
<td>266</td>
</tr>
<tr>
<td>A.30.3 Concrete Components</td>
<td>267</td>
</tr>
<tr>
<td>A.30.4 Metadata/Concrete Component Mapping</td>
<td>268</td>
</tr>
<tr>
<td>A.31 Iterator Pattern Redesign</td>
<td>271</td>
</tr>
<tr>
<td>A.31.1 Component Modification Possibilities</td>
<td>271</td>
</tr>
</tbody>
</table>

xxv
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.31.2 New Component Possibilities</td>
<td>272</td>
</tr>
<tr>
<td>A.32 Analysis of the GOF Mediator Pattern</td>
<td>273</td>
</tr>
<tr>
<td>A.32.1 Metadata</td>
<td>273</td>
</tr>
<tr>
<td>A.32.2 Concrete Components</td>
<td>274</td>
</tr>
<tr>
<td>A.32.3 Metadata/Concrete Component Mapping</td>
<td>275</td>
</tr>
<tr>
<td>A.33 Mediator Pattern Redesign</td>
<td>276</td>
</tr>
<tr>
<td>A.34 Analysis of the GOF Memento Pattern</td>
<td>276</td>
</tr>
<tr>
<td>A.34.1 Metadata</td>
<td>276</td>
</tr>
<tr>
<td>A.34.2 Concrete Components</td>
<td>277</td>
</tr>
<tr>
<td>A.34.3 Metadata/Concrete Component Mapping</td>
<td>278</td>
</tr>
<tr>
<td>A.35 Memento Pattern Redesign</td>
<td>280</td>
</tr>
<tr>
<td>A.35.1 New Component Possibilities</td>
<td>280</td>
</tr>
<tr>
<td>A.36 Analysis of the GOF Observer Pattern</td>
<td>281</td>
</tr>
<tr>
<td>A.36.1 Metadata</td>
<td>281</td>
</tr>
<tr>
<td>A.36.2 Concrete Components</td>
<td>281</td>
</tr>
<tr>
<td>A.36.3 Metadata/Concrete Component Mapping</td>
<td>283</td>
</tr>
<tr>
<td>A.37 Observer Pattern Redesign</td>
<td>285</td>
</tr>
<tr>
<td>A.37.1 Component Modification Possibilities</td>
<td>285</td>
</tr>
<tr>
<td>A.38 Analysis of the GOF State Pattern</td>
<td>286</td>
</tr>
<tr>
<td>A.38.1 Metadata</td>
<td>286</td>
</tr>
<tr>
<td>A.38.2 Concrete Components</td>
<td>286</td>
</tr>
<tr>
<td>A.38.3 Metadata/Concrete Component Mapping</td>
<td>287</td>
</tr>
<tr>
<td>A.39 Observer Pattern Redesign</td>
<td>289</td>
</tr>
<tr>
<td>A.39.1 New Component Possibilities</td>
<td>289</td>
</tr>
<tr>
<td>A.40 Analysis of the GOF Strategy Pattern</td>
<td>289</td>
</tr>
<tr>
<td>A.40.1 Metadata</td>
<td>289</td>
</tr>
<tr>
<td>A.40.2 Concrete Components</td>
<td>290</td>
</tr>
<tr>
<td>A.40.3 Metadata/Concrete Component Mapping</td>
<td>291</td>
</tr>
<tr>
<td>A.41 Strategy Pattern Redesign</td>
<td>292</td>
</tr>
<tr>
<td>A.42 Analysis of the GOF Template Method Pattern</td>
<td>292</td>
</tr>
<tr>
<td>A.42.1 Metadata</td>
<td>292</td>
</tr>
<tr>
<td>A.42.2 Concrete Components</td>
<td>293</td>
</tr>
<tr>
<td>A.42.3 Metadata/Concrete Component Mapping</td>
<td>294</td>
</tr>
</tbody>
</table>

xxvi
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>33</td>
</tr>
<tr>
<td>OO++ Python Logic Module Performance</td>
<td></td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>UML Structure of Iterator - Non-Deterministic Iterator</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>UML Structure of Iterator - Dynamic Interface</td>
<td>62</td>
</tr>
<tr>
<td>3.3</td>
<td>UML Diagram of GOF Iterator Pattern</td>
<td>69</td>
</tr>
<tr>
<td>3.4</td>
<td>UML Structure of Aggregated Elements Relation</td>
<td>74</td>
</tr>
<tr>
<td>3.5</td>
<td>UML Structure of Lazy Recursion Iterator</td>
<td>75</td>
</tr>
<tr>
<td>3.6</td>
<td>UML Structure of Parameterized Iterator Interface</td>
<td>76</td>
</tr>
<tr>
<td>3.7</td>
<td>UML Structure of Relation Extended ConcreteIterator</td>
<td>77</td>
</tr>
<tr>
<td>3.8</td>
<td>UML Structure of Dynamic Iterator Interface</td>
<td>78</td>
</tr>
<tr>
<td>4.1</td>
<td>UML structure of AbstractFactory - Extendable Kit</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>UML structure of Builder - Lazy Director</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>UML Structure of Factory Method - Non-Deterministic Constructor</td>
<td>95</td>
</tr>
<tr>
<td>10.1</td>
<td>UML Structure of Bridge - Relational Junction</td>
<td>111</td>
</tr>
<tr>
<td>12.1</td>
<td>UML Structure of Decorator - Double Dispatch</td>
<td>125</td>
</tr>
<tr>
<td>14.1</td>
<td>UML Structure of Facade - Dynamic Pseudo Instances</td>
<td>140</td>
</tr>
<tr>
<td>16.1</td>
<td>UML Structure of Chain of Responsibility - Relational Chain</td>
<td>148</td>
</tr>
<tr>
<td>17.1</td>
<td>UML Structure of Command - Command Function</td>
<td>154</td>
</tr>
<tr>
<td>19.1</td>
<td>UML Structure of Iterator - Non-Deterministic Iterator</td>
<td>163</td>
</tr>
<tr>
<td>19.2</td>
<td>UML Structure of Iterator - Dynamic Interface</td>
<td>169</td>
</tr>
<tr>
<td>21.1</td>
<td>UML Structure of Memento - Restoration Method</td>
<td>176</td>
</tr>
<tr>
<td>22.1</td>
<td>UML Structure of Observer - Descriptive Registry</td>
<td>182</td>
</tr>
<tr>
<td>23.1</td>
<td>UML Structure of State - Relational Table</td>
<td>188</td>
</tr>
<tr>
<td>24.1</td>
<td>UML Structure of TemplateMethod - Dynamic Hooks</td>
<td>192</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 Introduction

A paradigm is a set of assumptions, concepts, values, and practices that constitutes a way of viewing reality [1]. Software engineering practitioners use design paradigms to comprehend software tasks, to break these tasks down into smaller logical units, and to search for solutions to obstacles. They implement these tasks in a programming language, expressing their design through the features of the language. These programming language features are based on the programming paradigm of which the language is an instantiation. The ongoing evolution of programming languages occasionally produces a new class of languages with new features stretching the original programming paradigm or, in some cases, giving rise to a new one. When this happens, engineers must also evolve, or shift, the design paradigms they employ if they wish to fully utilize these additional features. Until this shift is complete, software engineers may experience some disability in their use of the new class of languages. Psycholinguists refer to a partial disability in the use of natural language as dysphasia. I use the term design dysphasia to describe a partial disability in the use of a programming language paradigm because of deficient design paradigms.

This thesis explores design dysphasia in object-oriented design patterns (a popular contemporary design method) and presents possible solutions. In the following sections, this introduction identify a recent paradigm shift in modern languages from object-oriented to multiparadigm and the resulting design dysphasia. Next, it illustrates the advantages of evolutionary maintenance in software design patterns for reducing design dysphasia. Finally, it discusses the application of these evolutionary
maintenance principles to the twenty-three object-oriented patterns described in [2] to produce multiparadigm pattern extensions suitable for implementation in modern programming languages.

1.2 Background and Motivation

Evolution in programming languages is well illustrated by the shift from imperative to object-oriented languages that occurred in the 1980’s and 90’s. As object-oriented languages became more common, imperative programmers who wished to use them had to shift the design paradigms they relied on to take advantage of the new object-oriented features. Until this design paradigm evolution was complete, they likely found it difficult to fully utilize these features; in other words, they experienced object-oriented design dysphasia.

More recently, there has been a subtle evolution in object-oriented languages to include features from other programming paradigms. In other words, although these languages are primarily object-oriented, they are in fact a subset of multiparadigm languages incorporating features from programming paradigms such as functional, logic, and visual. This subset of multiparadigm languages is characterized by a primarily object-oriented perspective plus features of one or more additional programming paradigms. I refer to this as the object-oriented++ (or OO++) programming paradigm. Illustrative examples of OO++ are Perl and Python which, in addition to object-oriented features, implement functional operators like map and filter, anonymous functions, and higher-order functions like currying. Microsoft Visual C++ augments the object-oriented paradigm with visual programming features and some functional capabilities as well. Logic programming is also found in many languages via database programming constructs like SQL. Indeed, most modern object-oriented languages incorporate features from multiple programming paradigms. This rise of OO++ is not surprising considering the valuable results achieved in multiparadigm
research (for examples, see [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]) and observations that object-oriented programming has important limitations [13] [14]. OO++ languages reinforce the conclusion that “object-oriented design is preferred ... for organizing systems and large subsystems” and that other programming paradigms “yield valuable results when applied within individual methods” [3].

The shift from object-oriented to OO++ is similar to the shift from imperative to object-oriented. Software engineers interested in using the OO++ language features must shift the design paradigms they employ to reflect the OO++ paradigm. Until this shift is complete, these engineers will experience design dysphasia with respect to the OO++ languages. A common result of this dysphasia is that the software engineer passes up multiparadigm solutions in contexts where they would be superior to traditional object-oriented solutions. Even worse, the software engineer might use the OO++ language features incorrectly producing a poor design and implementation.

1.2.1 Design Patterns and Design Dysphasia

Software design patterns name, explain, and evaluate important and recurring design solutions in a software system with the goal of capturing design experience in a form that other designers can use effectively [2]. The essence of a design pattern is expressed in many other terms such as “best practices” or even in the colloquialism “don’t reinvent the wheel.” The basic pattern concept is that the reuse of tested and proven concepts is a valuable design tool. Software patterns are a popular design tool and the subject of considerable research [15] [16] [17] [18] [19] [20].

However, current software design pattern philosophy can actually reinforce design dysphasia in pattern practitioners. A critical problem with the pattern philosophy is the emphasis on pattern “capture.” One source suggests that a pattern is not a pattern unless three naturally occurring instances can be identified [15]. In fact, the concept of creating a pattern from scratch is considered taboo. Consequently, there is
no notion of evolving existing patterns except through recapturing an evolved version of the original pattern. However, when programming languages evolve, it seems probable that pattern practitioners will be inclined to continue traditional pattern usage, ignoring new features (perhaps due to deadline pressures, lack of confidence with the new features, etc.). Under these circumstances, the only new patterns that would emerge would be from adventurous pattern practitioners or from those who had never learned patterns at all.

Christopher Alexander, whose research into architectural patterns provided a great deal of the inspiration for the software design patterns movement, believed that evolution was an essential characteristic of patterns. He said,

Since the patterns are independent, then you can change one at a time, and they can always get better, because you can always improve each pattern individually....

This is the eternal cycle of development. There is no hope of stilling it, and no need either. We must simply accept the fact that in the process of evolution, there is no final equilibrium. There are passing phases which approach equilibrium - but that is all. [21]

1.3 Contributions

In my research, I reincorporate Alexander’s notion of evolving patterns into software design pattern philosophy, and adapt patterns to the OO++ shift in programming paradigms. This not only prevents the reinforcement of design dysphasia, but also enables patterns to be an effective tool for eliminating it. By their definition, software design patterns represent mature and stable design solutions. Such solutions are ideal for evolutionary modification because the original solution is heavily tested and free from major design flaws. Additionally, practitioners of the original
patterns can use the evolved patterns to know when, where, and how to incorporate new language features into design solutions.

“Design Patterns” by Gamma, Helm Johnson and Vlissides (known as the Gang of Four or GOF) present a catalog of twenty-three object-oriented software design patterns. These patterns have become popular in academic and industrial circles as a convenient body of design knowledge. However, given the aforementioned shift in modern programming languages from object-oriented to OO++ feature sets, these patterns may be reinforcing design dysphasia in evolving languages. The later chapters of this thesis present evolved variants of the GOF patterns that take advantage of OO++ language features as concrete examples of pattern extensions to ameliorate design dysphasia.

One of the problems with extending software design patterns to make use of OO++ features is that OO++ features are not explicitly defined. Microsoft Visual C++, for instance, supports visual language features not supported in Perl. Leda, a broad multiparadigm language, has functional, logic and object-oriented features. Python supports currying while C++ does not, and some Python practitioners make use of extension modules that provide logic programming features.

To solve this problem, this thesis defines a pattern-extension system that catalogs OO++ extensions to the existing GOF patterns. These extended patterns include the language features required by the extension, the benefits and weaknesses of the extension over the base pattern, and implementation examples. The pattern-extension system is never closed; that is, pattern extensions can be added to the base pattern whenever additional language features are identified. Practitioners making use of such a system can choose to use the base pattern or any one of the extension variants given the circumstances of the design problem and the language features available.

In this thesis, I present pattern extensions that are limited to object-oriented,
logic, and functional language features, as these are the most historically relevant language paradigms [4] \(^1\). Sample implementations of each pattern are provided in Python. These patterns, extended to reflect the OO++ paradigm, demonstrates that object-oriented design patterns can be extended to effectively use the multiparadigm features of OO++ languages, thus reducing design dysphasia.

1.4 Thesis Organization

This thesis can be viewed as having three major divisions. The first section contains two chapters. 2 presents an in-depth investigation into the multiparadigm shift of contemporary object-oriented languages. 3 defines and describes design dysphasia in general and in the context of software design patterns and then defines a “maintenance cycle” for patterns that ameliorate the dysphasia. This part of the thesis represents the theoretical and philosophical investigation.

The next section, comprising 4 through 25, is a catalog of OO++ design pattern variants. These chapters represent the final “results” of this research endeavor.

The final section consists of conclusions in 26 and A which contains vital “intermediate” knowledge uncovered during the research process and utilized in creating the pattern extensions.

It should be noted that 2 and 3 are separate papers with their own abstracts, references and appendices. A short segment preceding each paper introduces the work, explains its role in the thesis as a whole, and identifies any repetitive sections that can be safely skipped by the reader.

\(^1\)In this research, the imperative paradigm is assumed to be a component of the object-oriented paradigm
2 INTRODUCING THE OBJECT-ORIENTED PLUS PLUS (OO++) PARADIGM

This chapter is essentially a reprint of a paper presented by the author at the Multi-paradigm Programming in Object-Oriented Languages (MPOOL) Workshop in June, 2004 (“OO++: Exploring the Multiparadigm Shift”). The format of this paper has been modified slightly to meet BYU Master’s Thesis requirements but retains independent references and appendicies.

The paper’s major contribution is the presentation and analysis of the OO++ paradigm. This chapter describes and defines the nature of the paradigm as groundwork for OO++ discussion and development. The paradigm presented is implicitly connected to the OO++ design pattern extensions presented in the latter chapters of this thesis.

The introductory sections of this paper present useful foundational information. The first section introduces the nature of the OO++ shift and the following section illustrates this shift through a “language-feature survey” of six contemporary languages. A short presentation of OO++ extensions for Python is then made. These extensions may be important to parties interested in examining the sample code of the OO++ design pattern variants in greater detail. The remainder of the chapter presents and analyzes the OO++ paradigm.
OO++: Exploring the Multiparadigm Shift

Seth J. Nielson, Charles D. Knutson

ABSTRACT: Programming languages and the programming paradigms they embody co-evolve over time. Within industrial and academic circles, for example, object-oriented programming has evolved from and effectively replaced imperative programming. More recently, many object-oriented languages have assimilated features from other programming paradigms, evolving into multiparadigm languages we refer to as "object-oriented plus-plus" or OO++. In this paper we survey the capabilities of six OO++ languages, present OO++ code samples in Python, and propose key characteristics of an OO++ programming paradigm.

2.1 Introduction

In the software development process, choosing a programming paradigm, and a programming language to support that paradigm, is a decision that affects all aspects of software creation including design strategies, implementation, and testing (see [1] for a pertinent example). The Sapir-Whorf hypothesis [2] would suggest that the language used to implement a software solution may influence the manner in which a developer analyzes and understands potential solutions.

Programming languages and programming paradigms evolve over time. Evolution within a community is characterized by a widespread language usage and must not be confused with isolated experimentation and theoretical research. Obviously, change within a community begins with an idea investigated by an individual or small group. However, we are interested in demonstrable trends that gain traction within the software development community at large.

An obvious example of this evolution is the shift from purely imperative programming to object-oriented programming. While object-oriented theory has existed since germinating in the 1960’s [3], widespread use of object-oriented languages did
not begin until the late 1980’s [4]. At present, object-oriented languages are a de facto standard for most industrial and academic software development and almost all languages support the object-oriented paradigm or have an object-oriented variant\(^1\).

More recently, there has been a noticeable shift from object-oriented programming to multiparadigm programming. Many modern “object-oriented” languages are evolving, or have evolved into, multiparadigm languages that include language features from non-object-oriented paradigms such as functional and logic programming. Such evolution has been facilitated either by native language features or through language extensions. These languages represent a special subset of all multiparadigm languages which share in common the support for object-oriented programming. To differentiate this subset of multiparadigm languages, we have chosen to identify them as *object-oriented plus plus* or *OO++* languages.

In this paper we briefly survey a number of OO++ languages and examine one of these languages (Python) in detail. Next, we identify key aspects of the OO++ programming paradigm. After presenting our conclusions, we also include two appendices at the end of this paper with technical information regarding multiparadigm extensions to Python.

2.2 A Survey of OO++ Languages

Historically, the imperative, functional and logic programming paradigms have been the most widely used paradigms next to object-oriented [5] and our survey of OO++ languages is consistent with this\(^2\). We found that, in these languages, functional programming is the most popular secondary paradigm (a “graft” into an otherwise object-oriented language) followed by logic programming. Other paradigms

\(^1\)Including object-oriented versions of such standards as Prolog, Fortran, Cobol, Lisp and Pascal

\(^2\)Since the object-oriented programming paradigm largely subsumes imperative programming (via data and code encapsulation in objects), we omit further discussion of imperative as an independent paradigm.
such as aspect-oriented and visual programming have benefited from active research communities, but are not as widespread, particularly in commercial off-the-shelf programming languages.

Among OO++ languages, we find some that natively support multiparadigm features. Python and Perl, for example, are two popular scripting languages that support the object-oriented paradigm and aspects of functional programming natively. These languages also support logic programming and additional functional language features through extension modules.

Ruby is another OO++ language that natively supports functional programming features. Ruby differs from Python and Perl in that object-orientation was the primary design goal and not an add-on to imperative programming. Despite the language’s drive for “pure object-orientation”, it supports anonymous functions and closures, which are definitely functional concepts.

C++ is also an OO++ language. While primarily object-oriented, C++ supports generic programming, a concept based in functional programming. Additionally, a number of libraries have been written for C++ that provide further multiparadigm language support. For example, FC++ [6] and LC++ [7] are two third-party libraries that augment C++ with extensive functional and logic programming capabilities.

Even some strictly object-oriented programming languages have been refitted into an OO++ variant. The Pizza language [8], for example, is a multiparadigm variant of Java. Pizza augments the Java programming language with functional constructs such as first-class functions, parameterized classes, and algebraic types. Pizza bytecode can operate on any Java virtual machine and the Pizza compiler can convert Pizza source code into standard Java.

Finally, Leda was designed from the start to be truly multiparadigm, seamlessly incorporating object-oriented, imperative, logic and functional programming features.
into a composite language that integrates the benefits of each. Because Leda was
designed to be a multiparadigm language, it excels at supporting collaborations between
the various paradigmatic features

2.3  OO++ Python

In this section we explore OO++ language features of Python in greater detail.
Python is an OO++ language with dynamic semantics. To illustrate OO++ code
in Python, we first provide examples of functional and logic-programming constructs
by themselves before presenting some examples of hybrid feature combinations. The
technical detail of the non-native features is described in two appendices at the end
of this paper.

In the Python code below, we demonstrate the following functional concepts:
closures, self-referential anonymous functions, and higher-order functions. This is a
small sample of Python’s functional capabilities (both native and extended), but it
is sufficient to demonstrate the breadth of functional support. Intricacies specific to
Python are explained in code comments (text following the ‘#’ character).

```python
# A Simple List Iterator. This demonstrates closures
# in Python. The one caveat of a closure in Python is
# that it only complex variables (like lists) maintain
# state across multiple calls to the function.
# For this reason we use the first element of a list
# (the listPosition variable) instead of a simple
# variable to store the current list position

def makeListIterator(someList):
    listPosition = [-1]  # The brackets mean list
    def nextFunction():
        listPosition[0] += 1
        return someList[listPosition[0]]
    return nextFunction

list1 = [2,4,6,8]
```
next = makeListIterator(list1)
next()     # Returns 2
next()     # Returns 4

# A self-referential lambda function. The "thisFunction"
# construct is not native to Python. It was
# implemented by the authors of this paper.

factorial = lambda x:((x==0) or (x*thisFunction(x-1))
factorial(3)    # Returns 6
factorial(4)    # Returns 24

# Finally, currying and compose are two examples
# of higher-order functions. While these two
# functions are not native to Python, their
# implementation (shown in Appendix A) is trivial.

plus = lambda x,y: x+y
plus(2,4)     # Returns 6
plus2 = curry(plus, 2)
plus2(5)      # Returns 7

square = lambda x: x*x
add_and_square = compose(square, plus) # square(plus(args))
add_and_square(3,2) # Returns 25

Logic programming in Python, unlike functional programming, is not supported
natively. We have implemented a Prolog-like module that supports logic constructs.
Following in typical Prolog fashion, we demonstrate the logic module with a family
relationships example.

parent = Relation()
parent.assert("Seth","Amy","Alex")
parent.assert("James","Aloyes","Seth")
parent.assert("Lloyd","Dorthy","James")

ancestor = Relation()
# The base case is that a parent is an ancestor
ancestor.inference += lambda ancestor, kid: (parent.query(ancestor, other, kid) | parent.query(other, ancestor, kid))

# The recursive case is that a parent of an ancestor is an ancestor.
ancestor.inference += lambda ancestor, kid: (parent.query(ancestor, temp) & ancestor.query(temp, kid))

X,Y = logicVar(), logicVar()
while ancestor.query(X,Y,Z):
    print X, "is ancestor of ", Y

Notice that the ancestor relationship is recursive. The output of the preceding script is shown below:

    Seth is ancestor of Alex
    Amy is ancestor of Alex
    James is ancestor of Seth
    Aloyes is ancestor of Seth
    Lloyd is ancestor of James
    Dorthy is ancestor of James
    James is ancestor of Alex
    Aloyes is ancestor of Alex
    Lloyd is ancestor of Seth
    Dorthy is ancestor of Seth
    Lloyd is ancestor of Alex
    Dorthy is ancestor of Alex

We now examine hybrid-programming constructs by presenting two simple examples.

The first example is an object-oriented-functional hybrid that we term method composition. *Method composition* is similar to functional composition except that side effects are expected (which are otherwise prohibited in functional theory). If
the objects to which the methods belong change state, the behavior of the composite function changes transparently.

Imagine that we are creating a computer strategy game. The game has a playing board with pieces that challenge one another if they attempt to occupy the same square. The process for moving a playing piece might be to register the move, resolve challenges (if any), update the game database, and update the screen. We could create this operation as a combination of the sub-operations:

```python
move = compose(screen.update, db.update,
    engine.challenge, game.move)
```

# The next two lines are equivalent
```python
screen.update(db.update(engine.challenge(game.move(
            piece, newX, newY))))
move(piece, newX, newY)
```

Notice that the `move` function is based on the state of the objects. If, for example, the user changes the preferences for the engine (e.g., tougher opponents, etc.) the `move` function is altered transparently.

The second example is a combination of the object-oriented and logic programming paradigms that we term *Non-deterministic Object Construction*. Instead of explicitly declaring the object we want constructed, we describe it by constraints. Our example scenario is a strategy game in which a player competes against the computer in a contest. In such a scenario, the player might wish to describe an opponent instead of requesting one. In the example below, the user competes against all opponents that meet his preferences.

```python
Define = Relation()
Define.assert(KnightClass, Attack(10), Armor(10), Move(2))
Define.assert(SamuraiClass, Attack(3), Armor(2), Move(1))
Define.assert(BoxerClass, Attack(5), Armor(1), Move(5))
```
Define.assert(NinjaClass, Attack(5), Armor(1), Move(10))

Prefs = ui.getUserPreferences()
LogicConstructor = Relation()
LogicConstructor.inference += lambda opponentClass: (Define.query(opponentClass, At, Ar, M) & Logic_operators.GT(Prefs.attack(), At) & #GT = GreaterThan Logic_operators.GT(Prefs.armor(), Ar) & Logic_operators.GT(Prefs.movement(), M))

badGuy = LogicVar()
while LogicConstructor.query(badGuy):
    badGuyClass = badGuy.value()
    badGuyInstance = badGuyClass()
    engine.fight(player, badGuyInstance)

2.4 Key Aspects of the OO++ Paradigm

Before attempting to characterize the OO++ paradigm, we emphasize the difference between a programming paradigm and a programming language. The former is a guiding philosophy in the creation of software while the latter is a collection of language tools for describing the software to the underlying computing platform. While, theoretically, any programming paradigm could be used to guide the creation of software in any language, in practice a developer uses a language that provides language features that reflect the philosophy of the paradigm. Indeed, “a language that supports a paradigm well is often hard to distinguish from the paradigm itself” [9].

A paradigm can be defined as a set of assumptions, concepts, values and practices that constitute a way of viewing reality [10]. We propose a characterization of the OO++ paradigm by describing key ideas in each of these areas. Our proposed programming paradigm is based on previous multiparadigm research.

Assumptions (Assumed without Proof):

15
• In OO++ languages, the object-oriented paradigm is always fully supported but support for secondary paradigms vary.

• A given programming paradigm leads more naturally to a solution than others for certain identifiable classes of problems.

• Multiple paradigms can be combined effectively within an object-oriented framework.

Concepts (*Ideas Derived from Instances)*:

• The object-oriented paradigm is suitable for construction of complex internal program structures and abstractions.

• Some secondary paradigm solutions can be hidden within methods and objects.

• Some secondary paradigm solutions can be integrated with object-oriented structures and abstractions.

Values (*Principles Considered Desirable)*:

• Multiparadigm solutions can be created that are more elegant, readable, and/or simpler than the alternative object-oriented solution.

• Secondary paradigms can be used to generate novel concepts and solutions.

Practices (*Customary Actions)*:

• The object-oriented paradigm can be used to analyze and design structures, states, and interactions.

• Multiple paradigms can be used to design data, operations and interfaces.

• Language features can be combined into multiple paradigm solutions.
• Language features can be combined into hybrid multiparadigm solutions.

We now explore each of these areas in greater detail.

2.4.1 Assumptions

An assumption is a principle that appears true and is accepted without proof. Similar to a mathematical axiom, assumptions provide a foundation for reasoning or decision-making and are essential to a programming paradigm. In the following paragraphs, we identify three critical foundational assumptions of the OO++ paradigm.

The first, and most critical, assumption is that in OO++ languages, the object-oriented paradigm is always fully supported\(^3\) while support for secondary paradigms may vary. All other characteristics of OO++ are affected by this assumption. It should be noted that this assumption holds true in the languages we surveyed earlier in this paper.

Secondly, we assume that for certain identifiable classes of problems a given programming paradigm may lead more naturally to a solution than other paradigms. Some of this information is available within the body of research investigating programming paradigms and the types of problems they solve.

Our third assumption is that multiple paradigms can be combined effectively within an object-oriented framework. Obviously the combination of multiple paradigms can result in a “paradigm soup” that is difficult to comprehend and utilize. We assert that the well-structured object-oriented paradigm is capable of integrating and incorporating other paradigms in a usable and comprehensible form.

2.4.2 Concepts

Whereas assumptions are foundational, the concepts are the building blocks. Not surprisingly, they are often directly related to the assumptions and sometimes overlap

\(^3\)Of course, not every “object-oriented language” supports the same set of features; the meaning of “fully supported” is somewhat subjective.
with them (assumptions can be concepts, for instance). But while assumptions are generally accepted without proof, a concept can be demonstrated. We identify three key conceptual building blocks of the OO++ paradigm.

The first concept is that the object-oriented paradigm is suited for construction of complex internal program structures and abstractions. The object-oriented paradigm is unique in that it provides numerous constructs specifically for defining and describing structure. Inheritance, information hiding, and encapsulation are examples of such features. In OO++ programming, the object-oriented paradigm is explicitly exploited for these structural definition features.

The second concept is that some secondary paradigm solutions can be embedded within methods and objects and provide solutions to sub-problems [11]. These problems, though small from an architectural perspective, may be daunting within their own limited contexts. By hiding these solutions within an object-oriented system, the "paradigm soup" challenge we referred to previously can be obviated.

We observe that the object-oriented paradigm subsumes the imperative paradigm. Within object-oriented programming, imperative concepts and structures are incorporated into the powerful architectural constructs of object classes and instances. With this in mind, the idea of allowing additional paradigms such as functional and logic programming to co-exist within object-oriented structures is reasonable.

The final concept we identify is that some secondary paradigm solutions can be integrated with object-oriented structures and abstractions. In these cases, multiparadigm solutions cannot be hidden within object-oriented abstractions because they are designed to augment and improve these abstractions. Non-Deterministic Object Construction, which we illustrated in the previous section for instance, is an example of this concept.
2.4.3 Values

While concepts and assumptions contribute necessary components to building complex thinking, values provide a paradigm with a framework for that building process by identifying desirable attributes at microcosmic and macrocosmic level. We suggest two such values within the OO++ paradigm.

The first value is the creation of multiparadigm solutions that are more elegant, readable, and/or simpler than the object-oriented solution. While this value may seem obvious, we emphasize what is not valued: speed, efficiency, and other performance factors. Also note that when multiparadigm solutions do not improve elegance, readability or simplicity in a software solution, then a purely object-oriented approach may be desirable.

The second value is the use of secondary paradigms to generate novel concepts and solutions. The first value we described focuses on improving specific software solutions. This value, on the other hand, drives improving problem comprehension and expanding solution possibilities for software development in general. Obviously, many novel solutions will be discarded because they are unviable, but even these may lead to better iterations or reincarnations that are effective.

2.4.4 Practices

Built on assumptions, concepts and values, a paradigm also includes practices that describe customary acts or common actions within the paradigm context. Obviously there is no way to describe all possible practices within a paradigm, but certain key practices help to define the paradigm. We identify four such OO++ practices.

The first practice is to use the object-oriented paradigm to analyze and design structures, states, and interactions. These components are important at all levels of design but focus on what the design is rather than how it works. We note that object-oriented design is a relatively mature discipline and fairly well understood.
Designing an object-oriented framework (either informally, or formally) provides a stable structure for inserting multiparadigm components.

The second practice is to use multiple paradigms to design data, operations and interfaces. While these three design issues overlap somewhat with the architectural design described in the preceding paragraph, their full exploration happens in later stages of design. When creating an architectural design, knowing that two objects interface is sufficient; how those objects interact is determined later.

Each programming paradigm defines data, operations and interfaces differently. Within the object-oriented paradigm, for example, data generally consists of simple and complex variables, and instances of classes. Operations are generally contained in methods, and the invocation of methods and the structure of classes are the interfaces. For a more complete discussion of the ways in which different paradigms conceptualize data, operations, and interfaces, see [5].

Data, operations, or interfaces from one or more secondary paradigms can replace object-oriented counterparts to improve the software solution. Determining when and how to make replacements requires an understanding of the secondary paradigms and an understanding of how the language features will interact with each other. In many cases, experimentation is necessary to generate effective solutions.

The third practice is a method of replacing object-oriented components resulting in multiple paradigm solutions. These solutions are characterized by each paradigm being used in such a way that it “is preserved within some narrow context” [5]. In other words, component pieces of data, operations or interfaces are still identifiable as belonging to one paradigm or another, even though they interact with data, operations or interfaces from a different paradigm. Intercommunication between paradigm

---

4In this reference, the author uses the terms data, operative units, and access points
components without “dramatically compromising the paradigmatic purity of either” [5] is generally the easiest way to combine paradigms.

The final practice is the other solution for replacing object-oriented components resulting in multiparadigm solutions\(^5\). Multiparadigm solutions are not ”paradigmatically pure” but combine elements from various paradigms to create a new hybrid synthesis. In this type of solution, the data, operations or interfaces may not be easily identifiable as belonging to a specific programming paradigm. In summary, a multiple paradigm solution is a combination of independent subcomponents each derived from individual paradigms, whereas a multiparadigm solution is an integral derivation from a hybrid paradigm and cannot be subdivided into separate paradigmatic components.

### 2.5 Conclusions

Multiparadigm languages are gaining traction in the software development community. We have observed that most modern “object-oriented” languages support at least one other paradigm either natively or through extension modules. We have chosen to call these multiparadigm object-oriented languages OO++. Python is a good example of an OO++ language, supporting multiparadigm features both natively and through extension modules.

In this paper we have proposed an OO++ programming paradigm in terms of assumptions, concepts, values and practices. Our OO++ paradigm is built around an object-oriented core for architectural design and multiparadigm solutions to solve specific solutions within a narrow context. Multiparadigm solutions generally replace an object-oriented data, operations, or interface component to improve readability or

\(^5\) Multiparadigm programming is different from a multiparadigm solution. Multiparadigm programming is a broad definition that may include both multiparadigm and multiple paradigm solutions.
simplicity. The results of these replacements are either *multiple paradigm* (interacting, but paradigmatically pure) solutions or multiparadigm (integral, hybrid) solutions.
LIST OF REFERENCES


2.6 Appendix: Functional Extensions in Python

2.6.1 Python Functional Background

Python is an object-oriented language with dynamic semantics. Python also supports the functional concepts of *first-class functions*, and *polymorphic typing*. Additionally, *map*, *filter*, and other list processing functions are natively provided.

Functions in Python can be defined to accept two classes of arguments. The first are known as *arguments* and the second are known as *keyword arguments*. When defining a function, a specific number of arguments can be enumerated and named or a list variable preceded by a single asterisk can be defined that accepts multiple arguments. Keyword arguments are always passed to a special Python dictionary object identified during definition by a preceding double-asterisk. Argument type is never explicitly declared and functions must provide their own type-checking if required. Functions and classes are legitimate data types. Functions optionally return data.

The code sample below illustrates these characteristics

```python
def simpleFunction():
    print "This function accepts nothing and returns nothing."

def simpleFunctionWithArgs(arg1, arg2):
    print "Function called with:", arg1, arg2
    return 1

def moreComplicatedFunction(arg1, arg2, **kargs):
    #The 'kargs' argument is the keywords argument dictionary
    #A dictionary is a unique-key/values pair Python object
    print "Function called with ", arg1, arg2
    print "Keywords/Values:",
    for key in kargs.keys():
        print key,"/",kargs[key]

#moreComplicatedFunction(1,2,X=3,Y=4) outputs:
```

```
# Function called with 1 2
# Keywords/Values:
# Y / 4
# X / 3

def variableArgsFunction(*args):
    # The 'args' argument is a python list object
    print "Function called with", args

#variableArgsFunction(1,"hi",3.4) outputs:
# Function called with (1,"hi",3.4)

def complexFunction(*args, **kargs):
    print "Function called with", args
    for key in kargs.keys():
        print key, "/", kargs[key]

#complexFunction combines variable arguments and keyword arguments

2.6.2 Higher-Order Function Extensions

Because Python allows functions to be passed and returned as data, creating higher-order functions is trivial. Below we define compose and curry\(^6\) in Python.

def compose(func1, func2):
    def composition(*args, **kargs):
        return func1(func2(*args, **kargs))
    return composition

def curry(originalFunction, *curriedArgs):
    def curriedFunction(*args, **kargs):
        return originalFunction(*((curriedArgs+args),**kargs))
    return curriedFunction

\(^6\)Of course, this is more of a pseudo-curry function. This function “freezes” a given argument. The true functional definition of curry is more complex
2.6.3 Self-Referential Lambda Calls

In Python, the dynamic nature of the language allows subroutines to introspect the interpreter to discover information about the call stack. We exploited this introspective ability to load the code of the calling subroutine and recreate the unnamed function.

```python
import new, sys

def thisFunction(*args, **kargs):
    previousCodeFrame = sys._getframe(1)
    callingFunction_CodeObject = previousCodeFrame.f_code
    callingFunction = new.function(callingFunction_CodeObject, globals())
    return callingFunction(*args, **kargs)
```

There are two major deficiencies to our extension. The first is that this extension is not guaranteed to be compatible with every version of Python. This is because it draws heavily on the current implementation of internal Python representations of code and functions. The other problem is that every call to `thisFunction` recreates the original function, thus wasting memory and degrading performance. In the future, we hope to create a more generic, efficient self-referential function.
2.7 Appendix: Logic Programming Extensions in Python

2.7.1 Python Logic Background

Logic programming is not supported at all in Python, requiring the implementation of a logic module from scratch. In creating such a module, we implemented a Prolog-like pseudo-syntax. We use the term “pseudo-syntax” to clarify that we will not modify the Python interpreter or language in any way. Instead, we will use existing Python syntax in a unique way to give the appearance of a new Prolog-like syntax.

In logic programming there are two distinct operations: defining data and querying. The syntax for defining data is relatively straightforward and maps well to Prolog syntax. Creating querying syntax, on the other hand, is more difficult because in Python, logic constructs must integrate with imperative/object-oriented features for which there are no counterparts in Prolog.

2.7.2 Defining Logic Programming Facts

The most basic data definition in Prolog’s version of logic programming is a fact. In Prolog, facts are defined within relationships explicitly, as in this example.

\begin{verbatim}
Father(Seth, Alex)
Father(James, Seth)
\end{verbatim}

This Prolog code declares that Seth is the father of Alex and that James is the father of Seth.

Our Python based pseudo-syntax is similar.

\begin{verbatim}
Father = Relation()
Father.assert("Seth","Alex")
Father.assert("James","Seth")
\end{verbatim}

Observe our use of the method “assert”. In Prolog, there is a definition phase and a query phase. Data can be defined in the query phase using an “assert” command.
In Python there are no forced distinctions for a definition phase and a query phase, hence we always use “assert” to declare facts.

Within our Relation object we store all the elements of each assertion in a unique Python list and then index the elements using Python dictionaries. These dictionaries, which use a hash table to map unique elements to values, serve to tie together equivalent elements in different lists. In other words, the “Seth” string in our example would exist in two separate lists at index zero and index one respectively. An internal dictionary stores the “Seth” string as well as its two locations within the relationship. See 2.7.6 for performance details.

2.7.3 Defining Logic Programming Inferences

Logic programming inferences are rules that define implicit connections between relationships and facts. In Prolog, data is defined in this manner:

\[
\text{Grandfather(grandfather, grandson) :-} \\
\text{Father(grandfather, son),} \\
\text{Father(son, grandson)}
\]

This inference for the Grandfather relationship implicitly ties the father of a father to his grandson. The comma in the code translates to an “and” in English.

Unfortunately, our pseudo-syntax for logic programming in Python is a bit awkward because of the constraints in declaring functions on the fly. We use lambdas to dynamically create the logical inference.

\[
\text{Grandfather = Relation()} \\
\text{Grandfather.inference = lambda grandfather, grandson: (} \\
\text{Father.query(grandfather, son) \&} \\
\text{Father.query(son, grandfather))}
\]

It is significant that the lambda function defined above will not operate correctly in Python if it is not attached to the inference of a Relation class. Python requires
that all variables be declared before use, which is not the case here. The Relation
class is written to intercept assignments to the inference member and dynamically
instantiate unknown variables as special LogicVariable instances.

Prolog allows multiple inferences to be assigned to any given relation and our
Python logic module is comparable. Any number of inferences can be assigned to a
Relation instance by using the += operator.

Grandfather.inference += lambda .

2.7.4 Logic Programming Queries

In the preceding section, we introduced query without explaining or defining it. The
query method of the Relation class returns an instance to a special iterator
class that we have created. Instances of this class can be used in truth tests and
returns true in these tests until the iterator is exhausted.

Father = Relation()
Father.assert("Seth","Alex")
Father.assert("James","Seth")

X,Y = LogicVariable(), LogicVariable()
while Father.query(X,Y):
    print X.value(), Y.value()

The query method of the Father relationship returns an iterator that returns
true as long as legitimate values can be assigned to the LogicVariable instances.

However, it is important to observe that every time the query method is called
(within the while loop, for example), an iterator is returned. To maintain state,
the query call monitors the variables being passed in and stores them. Every time
the same variables are passed in, the same iterator is returned until the iterator fails
(which causes a reset).
These special iterators are also operator-overloaded to handle and or operations. Inferences are built using these operations as described in the previous section.

When a relation composed of facts and inferences is queried, the relation first generates all possible answers from explicit facts. After all the facts are explored, inferences are used to generate answers in the order in which they were defined.

2.7.5 Special Operations

Prolog uses a special operation called cut to limit backtracking. In our implementation of logic programming, we support a cut method in the Iterator class. A call with a cut would look like this:

```prolog
Father.query(X,Y).cut()
```

However, we also allow the cut to be parameterized with the number of solutions that can be generated before cutting.

```prolog
Father.query(X,Y).cut(4)  # This will allow four solutions.
```

Comparison operators are also part of the Prolog language but because we rely on iterators for our system to work correctly, we require the use of a special set of comparison operators that we created for our Python logic programming module. The GT (greater than), LT (less than), NEQ (not equal), and EQ (equal) operators return iterators that can be used in inference construction.

The EQ operator is especially important because it can be used for logical assignment as well as for comparison. If bound logic variables are passed as parameters, EQ behave as a comparison operator. On the other hand, if an unbound logic variable is passed as one of the arguments, it will be bound to the value of the other parameter.

We also provide a special wrapper for any functions or callable objects. These wrapped functions return an iterator that can be used in inference construction.

```prolog
Relation.doFunction(<function name>, (<parameter list>))
```
This special wrapper allows us to insert print functions and other imperative/object-oriented constructs inside inferences of relationships.

2.7.6 Performance

Two major environmental issues hamper the performance of our logic module. The first is that the module is implemented in Python itself which significantly degrades performance. In future versions, a plug-in module written in the C programming language will solve this problem. The second issue is that we use complicated data structures to store and manipulate the logic data. This results in increased memory usage and degraded performance.

Table 2.1 shows the complexity of the major functions of the Logic module.
<table>
<thead>
<tr>
<th></th>
<th>Space</th>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fact Declaration</td>
<td>O(nm)</td>
<td>O(n log m)</td>
<td>$n =$ number of elements in the fact definition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$m =$ number of facts defined in the relationship</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n =$ number of elements in the fact definition</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$m =$ number of total unique elements defined in the relationship</td>
</tr>
<tr>
<td>Inference Declaration</td>
<td>O(n)</td>
<td>O(1)</td>
<td>$n =$ number of inferences in the relationship</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n =$ number of inferences in the relationship</td>
</tr>
<tr>
<td>Fact Queries</td>
<td>O(n log m)</td>
<td>O(n log m)</td>
<td>$n =$ number of elements in the query</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$m =$ number of total unique elements defined in the relationship</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$b =$ number of relationships composing the inference</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$a =$ number of inferences in the relationship</td>
</tr>
<tr>
<td>Inference Queries</td>
<td>O(a[n log m]b)</td>
<td>O(a[n log m]b)</td>
<td>$n =$ number of elements in the query</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$m =$ number of total unique elements defined in the relationship</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$b =$ number of relationships composing the inference</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$a =$ number of inferences in the relationship</td>
</tr>
</tbody>
</table>

Table 2.1: OO++ Python Logic Module Performance
3 DESIGN DYSPHASIA AND DESIGN PATTERNS

This chapter is a preprint of a journal paper written by the author and currently under review (“Design Dysphasia and the Pattern Maintenance Cycle”). The format of the paper has been modified slightly to meet BYU Master’s Thesis requirements but retains independent references and appendices. It should be noted that Section 6 and both appendices are duplicated in other parts of this thesis. Section 6, which provides extensions to the Iterator pattern, is reprinted as 19. The content of the appendices is repeated in A at the end of this thesis.

Sections 2 and 3 of this chapter describe design dysphasia and design patterns. These sections repeat somewhat the “Background and Motivation” section from 1 but provide greater detail. Specifically, Section 2 provides examples of design dysphasia inherent in the shift from imperative to object-oriented languages and Section 3 discusses the interaction of design patterns and design dysphasia in greater detail.

The most significant contributions in this chapter are found in Sections 4 and 5. The former identifies weaknesses in the maintenances of patterns and proposes a solution. The latter defines the methods employed in the development of the OO++ variants described in the following chapters.
Design Dysphasia and the Pattern Maintenance Cycle

Seth J. Nielson, Charles D. Knutson

ABSTRACT: Software developers utilize design methods that enable them to manipulate conceptual structures that correlate to programming language features. However, language evolution weakens the design-implementation interface introducing what we call “design dysphasia”—a partial disability in the use of programming language because of incongruous design methods.

Software design patterns are a popular design method that capture elements of reusable design within a specific context. When the programming languages that are part of pattern context evolve, patterns must adapt to the language change or they may reinforce design dysphasia in the practitioner. We assert that the current “capture/recapture” pattern maintenance model is suboptimal for adapting patterns to language evolution and propose a new “capture/modify/recapture” maintenance cycle as a more effective approach. We then suggest a concrete “modify” phase for current patterns to be adapted to object-oriented based multiparadigm language trends (which we refer to as OO++) and present an OO++ Iterator pattern example.

3.1 Introduction

The process of deriving a software solution from a given set of requirements includes design and implementation phases. These phases serve to drive a solution from abstract to concrete. An effective interface must exist between these phases such that an abstract design solution may be implemented using concrete language features. Of course there is rarely a perfect translation from design to implementation. The interface between phases represents the methods that elucidate the ambiguities of the abstract using concrete features.

In this paper, we focus on this interface between the design and implementation phases. The use of programming language features is inextricably linked to design
methodology. These design methods, whether formal or informal, are only effective when the design-implementation interface is coherent and complete. In other words, there must be a mapping from the design methods to features available within a language during the implementation phase.

However, even effective design-implementation interfaces must be adaptive or risk fractures because of shifts in programming languages. Programming languages and the paradigms they embody are not static but rather evolve over time. If design methods used to access the evolved features are deficient, they may produce a partial disability in the use of the programming language. *Dysphasia* is a psycholinguistic term that describes a partial disability in the use of natural language. We use *Design Dysphasia* to refer to a partial disability in the use of programming languages because of incongruous design methods.

In previous research, we investigated a recent shift in modern object-oriented languages to include features from other programming paradigms [1]. We termed these languages OO++ and described a corresponding programming paradigm. In this paper we suggest that object-oriented design methods with OO++ languages may lead to design dysphasia.

Software design patterns are one such design method. These patterns, which capture reusable design solutions, have been used in various academic and industrial settings to create object-oriented software. At present, equivalent and mature OO++ patterns do not exist.

In this paper, we investigate the impact of software design patterns on design dysphasia. We begin by discussing design dysphasia and design patterns in detail. We then propose a design pattern maintenance cycle to respond to evolutionary language changes and apply this method in the adaptation of patterns to OO++ languages.
We illustrate our proposals by extending the *Iterator* design pattern to an OO++ variant and then present our conclusions.

### 3.2 Design Dysphasia

Design dysphasia is a partial impairment in the use of a programming language because of incongruous design methods. We emphasize that design dysphasia is more complex than design ignorance because the design methods that are incongruous with one paradigm may be compatible with another. Evolutionary shifts in programming languages induce design dysphasia that is insidious because the design methods that cause the disability within an evolved language may be effective within the context of the original language paradigm.

The full impact of design dysphasia is not known, but we suggest three possible consequences. The first is that the power of new features is overlooked and ignored because they are not part of the practice of the existing design methodology. The second, and more damaging, consequence is that new features might be used incorrectly. Finally, because existing design methods are appropriate and effective under certain circumstances, design dysphasia might prevent acquisition of effective methods for designing; in other words, design dysphasia can be self perpetuating.

Design dysphasia is reminiscent of certain challenges in learning a second natural language. Linguistic researchers have noted that the linguistic structures formed in learning a first language can interfere with the acquisition of a second language. This is because “superficial similarities mislead the learner and are the cause of many errors” [2, italics added].

One of the best examples of design dysphasia is seen in the shift from structured and imperative programming to object-oriented programming and design. During this transition, industrial, research and pedagogical circles have scrambled to adapt to the language changes and the resulting paradigm shift. [3] discusses the challenges
of shifting to the object-oriented paradigm from an instructor’s perspective and observes that some professors failed to grasp the fundamental shift in object-oriented programming because of (superficial) similarities to imperative programming (e.g., seeing a method as a glorified function).

Fortunately, a number of factors may reduce design dysphasia over time. Experience, communication and research slowly replace obsolete design methods in dysphasia. In addition, educators begin to teach the new design methods in introductory classes helping to eliminate dysphasia from the next generation of students. This has been witnessed in the shift from the imperative to the object-oriented paradigm.

On the other hand, there are various influences that can reinforce design dysphasia within an individual or community. First, subtle language shifts that draw little attention and remain mostly unnoticed result in design dysphasia that likely remain unresolved. Second, languages that add ancillary features hardly demand new designs because original design methods continue to work well with the original features. This is especially true when the original design methods are popular and successful.

We observed in previous research [1] that there has been a recent shift from object-oriented to multiparadigm languages. Because this new class of language partakes primarily of the object-oriented paradigm, we refer to it as OO++. In exploring this shift, we suggested an OO++ programming paradigm and suggested possible uses. We also noted that the multiparadigm shift was both subtle and additive, or ancillary, in most cases.

Even though object-oriented designs are forward compatible with OO++ languages, we assert that until OO++ design methods are developed, understood, and employed, there will be a general design dysphasia with respect to OO++ languages. Observe that many OO++ languages are incorrectly described as object-oriented because of superficial similarities. Overlooking or misunderstanding multiparadigm
features may cause powerful OO++ solutions to be ignored or ineffective solutions to be generated. The popularity and success of the object-oriented paradigm and its accompanying design methods reinforces design dysphasia with respect to OO++ languages.

3.3 Software Design Patterns

Software design patterns name, explain, and evaluate important and recurring design solutions in a software system with the goal of capturing design experience in a form that other designers can use effectively [4]. The essence of a design pattern is expressed in many other terms such as “best practices” or even in the colloquialism “don’t reinvent the wheel.” The basic pattern philosophy is that the reuse of tested and proven concepts is a valuable design method. Software patterns are a popular design tool and are the subject of considerable research, especially within the object-oriented community.

One significant characteristic of design patterns is that they are not “created” per se. Instead, it is expected that a true software design pattern be captured from at least three different real-world systems [5]. Another popular metaphor is pattern mining where “nuggets of treasure...are separated from the surrounding residue” [6]. Another source states, “Great pattern writers are miners, they create nothing except the wonderful explanation” [7].

Patterns provide a number of benefits to practitioners. First, because they are captured from a number of viable sources, they represent “mature” designs. Secondly, they provide a vocabulary for describing designs within the documentation. Because of these advantages, patterns are touted as helping a “designer get a design ‘right’ faster” [4].

Using a design pattern effectively requires an understanding of the forces that affect the design solution. Forces are considerations that make the problem hard
and obvious solutions invalid. A good pattern resolves unaddressed forces within the system by canceling out undesirable forces and reinforcing stabilizing forces [8].

The context of a pattern includes any and all constraints placed on the system and helps to prioritize strong and weak forces [8]. Without a context definition, the pattern may make little sense. One aspect of context is the choice of programming language, which “influences one’s point of view” and directly impacts the appropriateness of a design pattern [4]. The programming languages employed have a significant impact on the forces within a system. The programming paradigms embodied by these languages, and the language features themselves, affect the design and implementation of software solutions. The connection between language and pattern are so strong that changes in language may result in a reconfiguration of forces possibly rendering the pattern obsolete [5].

The term “pattern” has been used to represent both the abstract design solution and an actual instantiation (or implementation) of that solution. This can be confusing, for example, when we say that a pattern has become obsolete. For our purposes, a pattern is the abstract design solution and a pattern that is obsolete means that the abstract design solution is no longer considered effective even though an implementation of that solution might function correctly.

Richard Helm stated:

There is also a danger in viewing the patterns in Design Patterns as the “Gospel according to the Gang of Four.” It must be realized that the patterns therein reflect our views about “reusable” OOD at a specific point in time. Other patterns do exist, and the existing patterns improve with use. [9]

Notice the claim that patterns should improve with use, or in other words, pat-
terns (should) have a maintenance cycle. Specifically, the abstract design solutions embodied in a pattern should get better over time.

Because the abstract design solutions of patterns are captured, rather than created, the concept of modifying the abstract design is deemphasized. When pattern maintenance is discussed, it often takes the form that a pattern is “mined” from existing software sources and continues essentially unmodified until it is made obsolete by another pattern or contextual evolution [5]. This cycle allows “bottom-up” pattern evolution that is driven from implementation changes and largely ignores “top-down” evolution in which the abstract pattern is changed directly. The reasoning for this capture/recapture sequence seems to be that these abstract designs represent mature, stable solutions that cannot be modified directly without introducing immature, unstable elements, thus reducing its efficacy.

We believe that this view of the pattern maintenance cycle is limiting and incomplete. Without “top-down” evolution, an abstract solution cannot be refactored or extended to adapt to changes in context, including language evolution. We assert that modifications to the abstract design solution are no more dangerous to the integrity of a pattern than “bottom-up” evolution; good changes are kept and poor ones discarded in both methods of change. The abstract solution of a pattern often represents the distillation of some significant “goodness” or “truth” in a design. Neglecting to dissect, explore, modify or extend this concentrated design principle inhibits innovation. Discarding or retiring a pattern because of perceived obsolescence may be unnecessarily wasteful.

Not only does such a maintenance cycle limit pattern development, it may also reinforce design dysphasia in pattern practitioners. The only way that new OO++ patterns might emerge, for example, is by capturing them from OO++ developers. It seems probable, however, that developers who use patterns will be inclined to
continue traditional object-oriented pattern usage, ignoring new multiparadigm fea-
tures (perhaps due to deadline pressures, lack of confidence with the new features,
etc.) Under these circumstances, the only new OO++ patterns that might emerge
would be from adventurous pattern practitioners or from those who had never learned
object-oriented patterns at all.

3.4 Improving the Design Pattern Maintenance Cycle

We propose a pattern maintenance cycle that supports adapting existing abstract
design solutions within a pattern in the face of contextual change. Top-down evolu-
tion of patterns increases pattern flexibility and prevents reinforcement of design
dysphasia.

In this maintenance cycle, a capture/modify/recapture sequence replaces the cap-
ture/recapture model. Using this variation, the evolution of a pattern alternates
between top-down and bottom-up evolution phases. A pattern begins by being “cap-
tured” or “mined” from existing sources. Over time the context in which it was cap-
tured begins to shift at which point the existing abstract design solution is extended
or adapted to the new configuration of forces. This intermediate phase inhibits design
dysphasia and encourages exploration of the new context and accompanying forces
which, in turn, generates new solutions. As newer solutions mature, new patterns are
captured, replacing the intermediate patterns.

The goal of the modify phase is to adapt existing design solutions of related pat-
terns to new or evolved forces that result from context shifts. Effective adaptation
requires an understanding of the nature of the patterns, their strengths and weak-
nesses, the forces that influence them, and an understanding of the context and forces
evolution. Our proposed four stage modify phase is designed to extend a context-
related family of patterns.

1. Analyze the contextual change and identify the resulting configuration of forces.
Before developing a pattern adaptation methodology, the changes in context must be correctly understood. Some changes are additive, or ancillary while others fundamentally alter the existing contextual fabric. In either case, new forces may emerge and others may fade resulting in a new configuration. These forces are not specific to a particular pattern, but decidedly impact how a family of patterns adapts to the context changes.

2. **Re-evaluate the existing pattern family in the evolved context.** The goal of this stage is to understand how the new forces resulting from the changed context affect the patterns as a group. If, for example, the context change is sufficiently dramatic, the associated pattern family may be entirely obsolete. In other circumstances (when changes are primarily additive or ancillary, for example) the patterns are made less effective, but are still useful.

3. **Develop and apply an adaptation methodology based on existing design knowledge.** It is very unlikely that contextual changes such as language evolution or software architecture occur independent of design theories. In this stage, our knowledge of the evolved forces and the resulting impact on patterns are integrated with context-appropriate design knowledge. This information is harnessed to create an adaptation methodology that is applied to individual patterns to produce extended variants.

4. **Capture and analyze high-level patterns.** As multiple patterns are modified to the evolved context, patterns of modification emerge. These higher-level patterns provide additional insight into the original patterns, the contextual shift, and the forces that play out in pattern maintenance. As these higher-level patterns are identified, they can be used to facilitate the continued adaptation of other non-evolved patterns.

The adapted patterns that result from the *modify* phase prime the *recapture* phase in the maintenance cycle. These intermediate patterns are, by definition, not proven,
stable designs. However, they help prevent design dysphasia by encouraging exploration of evolved contexts and providing insight at higher levels of abstraction. As they are used and implemented, practitioners will adapt or discard them in favor of better solutions until mature patterns replace the intermediate versions.

3.5 Adapting Object-Oriented Patterns for OO++ Languages

In this section, we apply the *modify* phase to the object-oriented family of patterns in the face of the OO++ language shift. As we noted in [1], the multiparadigm features of OO++ are not specifically enumerated. For the purposes of this paper, we limit our multiparadigm features to the object-oriented, functional and logic programming paradigms.

3.5.1 Analysis of OO++ Shift

The OO++ language shift is a primarily additive extension to traditional object-oriented programming. Many object-oriented languages today include language features from non-object-oriented paradigms. The added features and the reasons behind their inclusion vary widely from one language to another. The object-oriented paradigm is the one component these languages have in common.

In this context shift the following forces affect object-oriented patterns:

- *Object-Oriented Suitability:* The object-oriented programming paradigm is well-suited to system architecture and design.

- *Imperative Influence:* Within object-oriented classes and methods, much of the object-oriented idiom is based on imperative programming.

- *Idiomatic Multiparadigm Suitability:* Functional, logic and multiparadigm programming idioms are capable of replacing imperative elements within object-oriented classes and methods.
• **Architectural Multiparadigm Suitability:** Functional, logic and multiparadigm architectural features are capable of improving object-oriented structures.

• **Non-Uniform OO++ Features:** OO++ languages do not uniformly support the same multiparadigm language features.

• **Insufficient Functional/Logic Design:** The functional and logic programming paradigms have not yet developed extensive design knowledge and, therefore, it is difficult to know when to use features from these paradigms instead of traditional object-oriented counterparts.

• **Insufficient Multiparadigm Design:** There is limited design knowledge for combining elements from different paradigms into a cohesive architecture. Without this knowledge, multiparadigm combinations may result in “paradigm soup.”

• **Object-Oriented Dogmatism:** Some cultural influences try to suggest that object-oriented programming is good and non-object-oriented programming is bad regardless of circumstance or context.

### 3.5.2 Pattern Re-Evaluation in an OO++ Context

Because of the additive nature of the OO++ shift, object-oriented design patterns are forward compatible. OO++ languages support object-oriented programming, therefore existing object-oriented patterns can be fully utilized in these contexts.

However, OO++ patterns must resolve the OO++ forces identified in the preceding section. A suitable adaptation methodology must support a primary emphasis on object-oriented architecture while incorporating improvements from multiparadigm features when appropriate. A clear set of design principles must be enumerated that guides the identification of “appropriate” multiparadigm features to avoid the paradigm soup. Effective usage of non-object-oriented features reduces the force of
object-oriented dogmatism, although no technical solution can fully resolve this. Finally, the adaptation must support variability in object-oriented pattern extensions to resolve the force of non-uniform OO++ features.

3.5.3 OO++ Adaptation Methodology

A multiparadigm design methodology was presented in [10]. We have modified this methodology to 1) reflect the notion that pattern extensions are partial pattern redesigns and not designs created from scratch and 2) to resolve the forces identified in 3.5.1 and 3.5.2. A summary of this redesign methodology is presented below and then discussed in detail in the subsections that follow.

1. **Determine the Overall pattern architecture**

   (a) Identify pattern nature

      i. Purpose

      ii. Forces

      iii. Strengths

      iv. Weaknesses

   (b) Identify Data

   (c) Identify Structure

   (d) Identify Interaction

   (e) Identify Behavior

2. **Redesign the Pattern** - For each of the following, identify strengths, weaknesses, forces; evaluate improving strengths, eliminating weaknesses or stabilizing forces by incorporating logic and/or functional components

   (a) Redesign Data
3. **Create a Sample Implementation**

(a) Identify required language features  
(b) Implement Data  
(c) Implement Structure  
(d) Implement Behavior

4. **Record the Pattern Extension(s)**

3.5.3.1 **Determine the Overall Pattern Architecture**

Most object-oriented patterns provide metadata about the design solution they captures through UML diagrams, a description of purpose, a list of impacting forces, and a list of strengths and weaknesses. These elements describe the relationship between the pattern and its context and are essential to understanding the pattern as a whole. The concrete components of the pattern are the data, structures, interactions, and behaviors of the participant objects.

The goal of this phase can be summarized as the creation of a mapping between the decomposition of the pattern’s metadata, and the decomposition of the pattern’s concrete components. This analysis is captured in written form that describes each of the key components of the design solution and how that component stabilizes forces and contributes to the pattern’s purpose, strengths, and weaknesses.

3.5.3.2 **Redesign the Pattern**

The analytical results of the first phase drive the conceptual redesign of the pattern. Each of the key components described in the analysis document provides a
sufficiently fine granularity for incorporating features from the logic or functional paradigm. Because each pattern extension remains primarily object-oriented, few components change. Most often, the original components interface with new ones, although some of the original components may be replaced or redesigned. After the components of the pattern are modified, the pattern as a whole is reevaluated. The redesigned pattern must provide some identifiable benefit before proceeding to the next phase.

Another important aspect of the redesign phase is determining how and when functional and logic programming components should be used. This determination is made by analyzing the conceptual design of the original pattern and evaluating it against guiding principles of functional or logic analysis or design. Without guiding principles, multiparadigm features might be added where it is feasible technically but conceptually unsound. While some intuitive and unguided modification may be acceptable, widespread modification in this manner results in a pattern that cannot stabilize the forces previously identified in 3.5.1.

A summary of guiding principles is shown below (these principles are presented in [10]):

- Logic Analysis and Design
  - Data Refinement
    * Fact Refinement
    * Rule Refinement
  - Query Refinement
    * Think Non-Deterministically
  - Control Design
Based on the analysis of the pattern, new components are introduced and original components are evaluated for redesign based on these principles. This information is captured in a written design document describing the new and/or modified components. The description of participants includes the relevant guiding principles.

During the redesign of the object-oriented patterns some modifications occur that are not based in these guiding design principles. In some cases, a new design principle may emerge. If this occurs, the new principle is also recorded in the design document along with a short description. In other cases where no design principle is identified, the description of the participant will state that the modification was created ad hoc.

3.5.3.3 Create a Sample Implementation

After completing the redesign of the pattern, a sample implementation is created. The essential aspect of this phase is that conceptual design features are adequately expressed through language features. We have chosen to use Python (utilizing the extension modules proposed in [1]) as the implementation language, since it can
support all of the language features necessary to express object-oriented, functional and logic programming concepts.

Python, with extension modules, supports the following functional language features:

1. Functions as first class citizens
2. Higher-order functions (e.g., curry, map, filter, reduce, etc.)
3. Anonymous functions (including anonymous recursive functions)
4. Limited support of closures
5. Limited support of bindings that prevent overwriting of variables
6. Pseudo-support of infinite sets and lazy evaluation

While the last three features are provided with limited support, they are adequate for the purposes of the pattern sample implementations.

Python, with a logic module designed in [1], supports:

1. Unification
2. Cut operation (stops backtracking)
3. Logical operations (and, or, not)
4. Comparison operations (less than, greater than, equal to, etc.)

These features are sufficient to provide sample implementations for OO++ design patterns that combine object-oriented programming with functional and/or logic programming.
3.5.3.4 Record the Pattern Extension(s)

Finally, the pattern extension is documented in a meaningful way. This written form of the extension is similar in structure to traditional object-oriented patterns.

- **Base Pattern:** The base pattern being extended
- **Pattern Extension Name:** The name of the extension
- **Motivation:** Motivation for the extension
- **Forces:** Factors that make the solution non-obvious
- **Applicability:** When to use the extension
- **OO++ Interactions:** Explains the multiparadigm components and the required language features
- **Structure:** The structure of the design solution (including UML diagram)
- **Extended Participants:** Verbally describes the components of the pattern that are new or that have been modified
- **Collaborations:** How to interface to the pattern
- **Consequences:** Details the altered strengths and weaknesses from the base pattern
- **Implementation and Sample Code:** Describe important aspects of implementation
- **Related Patterns:** Relationships to other patterns
All the information necessary to create the pattern extension document is generated in previous phases of the pattern redesign. We have used this methodology to adapt the **Iterator** pattern for OO++ languages. The complete pattern variant is provided in Section 3.6.1.

### 3.5.4 Capture High-Level Patterns

Until more patterns are adapted for OO++, patterns of pattern modification cannot be captured or identified. We did observe in our extension of the **Iterator** pattern that both the functional and logic modifications tended towards filtration. It is possible that patterns have natural attractors in modification.

Additional work in [11] has demonstrated that functional concepts can be used to reduce class hierarchies. Certain patterns such as decorator and observer have class hierarchies simply for wrapping a single function. When functions can be passed as data (first class functions), the need for these classes disappears and the class hierarchy is simplified.

### 3.6 The OO++ Iterator Extensions

In the following subsections, we present two extensions to the **Iterator** pattern. The first extension incorporates elements of logic programming while the second incorporates functional constructs. These extensions require the reader to be familiar with the original GOF **Iterator** pattern. The analysis and redesign notes in the appendices may also be useful.

#### 3.6.1 Iterator - Non-Deterministic Iterator

**Base Pattern - Extension Name**

Iterator - Non-Deterministic Iterator

**Motivation**
Iteration over the elements of aggregated data is sometimes order independent. In circumstances where the amount of aggregated data is very large, some traversal policies serve as filters. These iterators are not necessarily concerned about a specific order of iteration because the goal is to guarantee that all of the data elements that meet certain constraints are visited. We term this form of iteration constraint-based iteration.

Logical relations are well-suited for this form of data traversal. This is because a relation can be viewed as a constraint-based iterator. The relation is created by describing rules (or constraints) and the relational query produces all data elements that satisfy the constraints.

Another advantage of using a relation as an iterator is that it is easily extended through new relations. A client can easily create new traversal policies simply by incorporating the original traversal relation into new rules. This form of extension requires neither an understanding of the aggregated data structure nor basic iteration.

We note that the basic iterator of the aggregated data does not have to be a logical relation. If the language supports wrapping non-logical elements into a logical relation, then the basic iterator can be a traditional object-oriented class instance. This iterator is then wrapped into a logical relation and the aggregated data can then be traversed in both non-deterministic and traditional fashions.
Forces

- Large amounts of data are difficult to filter and manage.
- Queries on logical relations are similar to iteration.
- Clients want to be able to iterate without understanding the process of iteration.
- The object-oriented approach to iteration requires an understanding the process of iteration.

Applicability

Use a Non-Deterministic Iterator pattern

- to provide traversal that is not directed.
- to support traversal of a describable subset of data elements
- to allow the Client to extend the traversal without understanding the aggregated data structure or basic iteration.

OO++ Interactions

Conceptually, the logical relation can be viewed as a non-deterministic iterator. It should be noted that wrapping a traditional iterator into a logical relation is a functional concept.

This pattern extension requires the following language features:

- Logical relations
- Wrapping traditional iterators into a relation [optional]

Structure

Figure 3.1 captures the structure of this extension in UML.

Extended Participants
Figure 3.1: UML Structure of Iterator - Non-Deterministic Iterator

- **WrappedIterator** (relation)
  - Wraps the ConcreteIterator into a logical relation.
  - Iterates across the same elements as the ConcreteIterator.

- **NonDeterministicIterator** (relation)
  - Combines the WrappedIterator relation with additional rules.
  - Iterates across all data elements that meet the defined constraints.

**Collaborations**

- A ConcreteIterator supports traditional directed traversal over aggregated data.
- Querying ConcreteAggregateElements relation returns the same elements as the ConcreteIterator.
- Querying ConstrainedElements returns all elements that match the relation’s constraints.
Consequences

Non-Deterministic Iterators have the following strengths and limitations:

1. *It allows iteration to be defined by constraints.* In a logical relation, defining search constraints is a straightforward process. Conceptually, the relation describes the subset of aggregated data to be iterated.

2. *It abstracts away the structure of the aggregated data from the Client.* The WrappedIterator relation only guarantees traversal of all aggregated elements without suggesting how the traversal will occur and without implying what the underlying structure is. The Client can write NonDeterministicIterator relations without having to learn anything about the aggregate.

3. *Changes to the aggregated data structure have minimal impact on dependent logical relations.* As long as WrappedIterator returns every element of the aggregate, no dependent relations will be adversely impacted by changes to the aggregated data structure.

4. *Non-deterministic iterators cannot be used for directed iteration.* Non-deterministic iterators do not control the order of traversal. On the contrary, they operate under the assumption that traversal order is irrelevant and possibly dynamic. If, however, the ConcreteAggregate supports both a ConcreteIterator and WrappedIterator, then both methods of traversal are available to the Client.

Implementation and Sample Code

The primary issue in implementing a non-deterministic iterator is creating an implementation of WrappedIterator. In the most simple case, the CreateIterator() method of ConcreteAggregate returns a logical relation and there is no need for
WrappedIterator at all. NonDeterministicIterator can be extended directly from ConcreteIterator.

If, however, ConcreteIterator is a traditional iterator, then it must be wrapped into WrappedIterator. This wrapping process is language specific. In the sample code that follows, we use a function provided by our Python Logic module that converts a function and its associated arguments into a logical relation.

In the sample code below, we show the implementation of a WrappedIterator and a NonDeterministicIterator.

For the WrappedIterator, we chose to implement a subclass of relation rather than simply create a new relationship. We did this because it allows us to instantiate a predefined WrappedIterator for any existing ConcreteIterator.

```python
class WrappedIterator(Relation):
    def __init__(self, iterator):
        Relation.__init__(self)
        self.__iterator = iterator
        self.__iterator.first()
        self.inference = lambda Element: (  
            Relation.doFunction(lambda: not iterator.isDone(),()) &  
            EQ(Element,  
                Relation.doFunction(iterator.currentElement,()) &  
                Relation.doFunctionOnce(iterator.next,())))

    def reset(self):
        Relation.reset(self)
        self.__iterator.first()
```

The client uses the WrappedIterator like this:

```python
i = ConcreteIterator(concreteAggregate)
wi = WrappedIterator(i)
element = LogicVar()
while wi.query(element):
    # Do stuff with element
```
Instances of WrappedIterator can be included in other logical definitions. In the example below, assume that customerList stores thousands of customers and their commercial data. The client wants to iterate over the customers who should be sent mass mailings about student-debt consolidation through home equity. This can be done easily.

\[
\begin{align*}
I &= \text{CustomerIterator(customerList)} \\
wi &= \text{WrappedIterator}(i) \\
\text{TargetedIterator} &= \text{Relation()}
\end{align*}
\]

\[
\text{TargetedIterator.inference} = \lambda \text{Element}: (wi(\text{Element}) \quad \text{HasHome}(\text{Element}) \quad \text{HasGoodCredit}(\text{Element}) \quad \text{HasStudentLoans}(\text{Element}))
\]

Of course, the relations HasHome, HasGoodCredit, and HasStudentLoans must be defined. But the important aspect of this is that changes in their definition have little impact on the validity of TargetedIterator. HasGoodCredit, for example, could initially check a raw credit score against some number. Later, a more complicated logical relation could be implemented that determines good credit based partially on factors such as the customer’s business relationship with the commercial entity, income, family status and college education. While improving HasGoodCredit increases the accuracy of TargetedIterator, it does not affect the interface between the two.

**Related Patterns**

WrappedIterator is a specialized form of the Adaptor pattern.

**3.6.2 Iterator - Dynamic Interface**

**Base Pattern - Extension Name**

Iterator - Dynamic Interface
Motivation

Iterators provide an abstract interface for traversing aggregated data. The traditional interface supports methods for resetting the iterator to the first element, advancing to the next element, checking to see if iteration is complete, and a method for returning the current element. If the Client requires more specialized methods than ConcreteIterator already supports, a new subclasses must be created.

However, in circumstances where new traversal methods are required for a limited time or within a limited scope, extra subclasses are very expensive. For example, if a client requires additional traversal methods within a method or a single class only, then the subclass is highly coupled with that class and may exist solely to provide a single function. If the extended traversal method does not require privileged access to the ConcreteIterator or ConcreteAggregate, a better solution is for the client class to implement a (possibly private) method that performs the additional operation. However, this solution may lead to an awkward and complex interface. Also, because the client is implementing methods to serve another class, it may be difficult to keep the intent of the methods clear and their definitions cohesive.

Dynamic methods stored as data within the client class provide a solution to these problems. In this case, dynamic functions are created through currying, composition or other higher-order functions to create the extended interface. These constructs clarify intent and improve cohesiveness by enforcing a functional view of the extended methods. Creating an Iterator interface that is parameterized facilitates this process.

For example, an aggregated structure may contain a large number of bank accounts and associated data. Using traditional iterators, a client interested in traversal of overdrawn bank accounts must use a subclass of the basic iterator or implement internal methods that iterate over the desired elements. But conceptually, the client
is now providing methods that operate on iterators, which can be confusing and non-cohesive.

On the other hand, if the basic iterator provides methods that accept an optional filter function, specialized filtered traversal methods are no longer required. If a specialized method is desired for readability or cohesion, currying provides an effective method for dynamically creating the function. This function can be stored as a data variable so that it is not confused as a method that operates on the client.

**Forces**

- Clients need extendable iterators.
- An individual client may require an iteration interface that is unique to that client.
- Subclasses may introduce unacceptably high overhead.
- Methods defined for the client may be confused as methods that operate on the client.
- Functions can be manipulated independent of the client or the iterator.

**Applicability**

Use Dynamic Interface when

- static changes to interface are too expensive.
- specialized methods or operations will only be used in a limited scope.

**OO++ Interactions**

Dynamic functions allow a client to manipulate the interface of an iterator to its own needs without introducing global constructs. These functions operate on the
iterator but are stored as data within the client class. If the iterator interface allows it, functions can also be passed as policy-modifiers to the iterator.

This pattern extension requires the following language features:

- First-Class Functions
- Higher-Order Functions

**Structure**

Figure 3.2 captures the structure of this extension in UML.

**Extended Participants**

- ConcreteIterator
  - Modified to accept policy modification functions.
- DynamicInterface (Multiple Functions)
  - Created using higher-order functions.
– Are data of the Client.

– Operate on ConcreteIterator.

Collaborations
Client creates DynamicInterface from the ConcreteIterator.

Consequences
Dynamic Interface has two primary benefits:

1. *It limits extended interfaces to a specific Client.* Because the functions belong to the Client as data, no other actors within the system know about them. This keeps the ConcreteIterator interface and class hierarchy simple.

2. *It prevents confusion about method purpose.* Conceptually, a method operates on a class; Client methods that operate on an Iterator may cause confusion. Dynamic methods that serve as data for the Client reduce this confusion.

Implementation and Sample Code
There are a number of factors that affect implementation of a Dynamic Interface.

1. *Does the Iterator support policy modifying function arguments?* If it does, currying and composition are easy to use. If it does not, composite functions and anonymous functions are the best options for a dynamic interface.

2. *What other functions will be combined with the iterator’s methods?* For example, in currying the `next()` method with a policy modification function, it is possible that a function already exists that can serve as a policy modifier. In the case of the overdrawn bank account mentioned previously, the bank account class probably implements an `isOverdrawn()` method.
3. *What higher-order function will be used?* Currying is an excellent tool for combining methods with policy modification functions. Composition can combine a method with a parameterized policy modification function. Anonymous functions serve as more general-purpose dynamic functions but are less cohesive than curried or composite functions.

In the sample code below, we generate composite and curried functions from a parameterized iterator interface.

```python
i = BankAccountIterator(bankAccountList)

firstOverdrawnAccount = curry(i.first, BankAccount.isOverdrawn)
nextOverdrawnAccount = curry(i.next, BankAccount.isOverdrawn)

firstAccountGreaterThan = compose(i.first,
        lambda min: lambda a: a.balance() > min)
nextAccountGreaterThan = compose(i.next,
        lambda min: lambda a: a.balance() > min)

firstOverdrawnAccount() while not i.isDone():
    print i.currentElement()
    i.nextOverdrawnAccount()

firstAccountGreaterThan(100) while not i.isDone():
    print i.currentElement()
    i.nextAccountGreaterThan(100)
```

**Related Patterns**

Composite Functions are similar to Composite Classes.

Composite Functions are functional Decorators.

### 3.7 Conclusions

Design dysphasia, a disability in the use of programming language because of incongruous design methods, inhibits effective software development. Shifts in programming language induce design dysphasia and the greater the shift, the greater
the impact of the dysphasia. The shift from imperative programming languages to object-oriented languages is an illustrative example of language shifts and the resulting dysphasia.

Design Patterns are an effective software design tool used significantly by object-oriented developers. However, design patterns may reinforce design dysphasia because they are generally not adapted to contextual shifts such as language evolution. We suggested that adapting patterns to such changes can and should be part of the pattern maintenance cycle.

In this paper, we demonstrated a possible methodology for adapting object-oriented patterns to the OO++ paradigm and presented two OO++ extensions to the *Iterator* pattern. We assert that such extensions can reduce design dysphasia and induce more OO++ programming designs, eventually resulting in more mature and stable OO++ patterns. We have identified a few possible patterns of pattern modification but expect to capture better ones as additional object-oriented patterns are modified and extended.
LIST OF REFERENCES


3.8 Appendix: Analysis of the GOF Iterator Pattern

In this appendix, we apply the analysis phase described in 3.5.1 to the Iterator pattern described in [4].

The purpose of the Iterator pattern is to “access the elements of an aggregate object sequentially without exposing its underlying implementation” [4].

- Metadata

  - Forces

    * Information hiding is valued
    * High cohesion is valued
    * Loose coupling is valued
    * Client expects interface for sequential access
    * Client may require multiple iteration policies

  - Strengths
• Supports variations in aggregate traversal
• Simplifies the aggregate interface
• Allows more than one pending traversal
• Abstracts internal structure of aggregate

− Weaknesses
  • May require aggregate to export methods (high coupling)
  • May require access to private aggregate data (violates encapsulation)
  • May be difficult to traverse recursive composite structures

• Concrete Components

  − Data
    • ConcreteAggregate reference (object)
    • Aggregated elements references (objects)
    • Traversal State
    • ConcreteIterator Reference

  − Structure
    • ConcreteAggregate implements the Aggregate interface
    • ConcreteIterator implements the Iterator interface
    • ConcreteAggregate creates ConcreteIterator
    • ConcreteIterator operates on ConcreteAggregate

  − Interaction
    • Aggregate publicly provides createIterator() to the Client
    • Iterator publicly provides first(), next(), isDone(), and currentItem() to the Client (additional methods are optional)
Behavior

* ConcreteAggregate defines the correct iterator to be returned from createIterator()

* ConcreteIterator defines a traversal policy
  - The first() message sets the current internal state to point to the first aggregate element according to policy
  - The next() message set the current internal state to point to the next element from the current element according to policy
  - The isDone() message requests a Boolean determination of complete transversal of all elements that meet policy requirements
  - The currentItem() message requests the current aggregate element pointed to by the current internal state

3.8.1 Metadata/Concrete Component Mapping

ConcreteAggregate Reference - In implementation, instances of the ConcreteIterator class are explicitly tied to the ConcreteAggregate they traverse. Internally, they generally access the ConcreteAggregate through a pointer.

References to Aggregated Elements - The traversing operations require access to the elements within the aggregate structure. This data is the source of the three weaknesses of the Iterator pattern that we have identified.

- Privileged Access Violates Encapsulation - If the ConcreteIterator has access to the private data of the ConcreteAggregate, then modifying the ConcreteAggregate produces cascading side effects in the Iterator and its subclasses.

- Special Methods Create a Fragile Interface - If the ConcreteAggregate exports access methods, then defining new traversals (through new ConcreteIterator
classes) may require new methods to be exported in the ConcreteAggregate. While it is expected that the ConcreteIterator depends on the ConcreteAggregate, making the ConcreteAggregate dependent on the ConcreteIterator introduces an awkward coupling.

- **Recursive Composite Traversal Complicates Implementation** - If the ConcreteIterator is to traverse a recursive composite structure, it must store a traversal path through the composite making the implementation challenging.

*Traversal State* - Multiple traversals over the aggregate data is possible because each instance of ConcreteIterator maintains a separate traversal state. However, aggregated data that is recursive may complicate storing the traversal state if storing a path through the recursive data is necessary.

*ConcreteIterator Reference* - The client accesses the ConcreteIterator directly to control iteration through the data of the ConcreteAggregate.

*ConcreteAggregate Implements the Aggregate Interface* - This enables the use of the createIterator() factory method.

*ConcreteIterator Implements the Iterator Interface* - This enables interchangeable iterators.

*ConcreteAggregate Creates ConcreteIterator* - This frees the Client from having to know which ConcreteIterator corresponds to a given ConcreteAggregate.

*ConcreteIterator Operates on ConcreteAggregate* - This structure is the heart of the pattern and allows the traversing of the aggregate to be separated from the structure of the aggregate. This reinforces high cohesion by allowing the ConcreteAggregate class to be responsible for the elements and the ConcreteIterator class to be responsible for traversal.

*Aggregate publicly provides createIterator() to a client* - createIterator() is a fac-
tory method. The client calls the createIterator() of a specific ConcreteAggregate to instantiate the appropriate ConcreteIterator. This factory method can be parameterized allowing ConcreteAggregate classes the ability to create instances from different ConcreteIterator classes each with different traversal policies.

*Iterator publicly provides first(), next(), isDone(), and currentItem() to a client*

- The Iterator interface is inherited by all ConcreteIterator classes. These methods abstractly define sequential iteration without reference to any specific concrete implementation.

*ConcreteAggregate defines the correct iterator to be returned from createIterator()*

- Each ConcreteAggregate overrides the createIterator() Factory Method to return an appropriate ConcreteIterator instance.

*ConcreteIterator defines a traversal policy* - The details of the traversal policy are hidden from the Client but define a concrete implementation for the abstract methods that are defined by the Iterator interface.
3.9 Appendix: Iterator Pattern Redesign

In this appendix we apply the redesign phase described in 3.5.2 to the analysis generated in previous section.

3.9.1 Component Modification Possibilities

Use a relation for aggregated elements references (Logic)

- **Description** - The ConcreteAggregate provides a logical relation that can query all of the aggregated elements. This relation can operate as a ConcreteIterator but can also be easily extended by new relations created by the Client. This kind of iterator cannot traverse in a directed fashion (e.g., sequential, pre-order, etc.).

- **Guiding Principle** - Think Non-Deterministically

- **Benefit** - Extendable traversal policies without subclasses

- **Drawback** - Cannot direct traversal policies

- **Structure** - Figure 3.4 captures the structure of this extension in UML.

Store recursive traversal state in lazy recursive function (Functional)

- **Description** - Normal recursive functions have been used to support internal iterators for recursive composite structures. Lazy recursive functions allow recursion to be delayed and can be used to provide an external iterator for these structures.
Figure 3.5: UML Structure of Lazy Recursion Iterator

- **Guiding Principle** - Adapting Similar Problems

- **Guiding Principle** - Recursive Refinement

- **Benefit** - Improved traversal of recursive composite structures

- **Structure** - Figure 3.5 captures the structure of this extension in UML.

Extend Iterator Interface methods to accept functions that partially modify traversal policy

- **Description** - The next(), first(), and isDone() methods are modified to accept a function as an optional argument. This function serves as a filter and can modify the traversal policy.

- **Guiding Principle** - Adapting Similar Problems

- **Guiding Principle** - Pipes and Filters
3.9.2 New Component Possibilities

Extend ConcreteIterator with logical relation (Logic, Functional)

- **Description** - Additional iterators can be created that are logical relations. The ConcreteIterator can be wrapped into the relations even if the ConcreteIterator is not a relation itself.

- **Guiding Principle** - Think Non-Deterministically

- **Guiding Principle** - Rule Refinement

- **Guiding Principle** - Adapting Similar Problems

- **Guiding Principle** - Type Refinement

- **Guiding Principle** - Functional Composition

- **Benefit** - If the ConcreteIterator is not a logical relation, then both directed iteration and non-deterministic iteration could be applied to the aggregated structure.
- **Benefit** - The Client can extend the ConcreteIterator without understanding the aggregated data structure or the internals of the ConcreteIterator.

- **Structure** - Figure 3.7 captures the structure of this extension in UML.

**Dynamically create additional iteration methods (Functional)**

- **Description** - The Client can dynamically create a new function through currying, composition or anonymous functions.

- **Guiding Principle** - Adapting Similar Problems o Guiding Principle - Type Refinement

- **Guiding Principle** - Functional Composition

- **Benefit** - New functionality within a limited scope (e.g., an individual method) without modifying interfaces or creating new subclasses.

- **Structure** - Figure 3.8 captures the structure of extension in UML.
Figure 3.8: UML Structure of Dynamic Iterator Interface
4  OO++ ABSTRACT FACTORY EXTENSIONS

4.1 Base Pattern - Extension Name

Abstract Factory - Extendable Kit

4.2 Motivation

Abstract Factory “Provide[s] an interface for creating families of related or dependent objects without specifying their concrete classes.” While this pattern has sundry advantages, it is a known limitation that adding new products is difficult. Adding a new product requires a change to the AbstractFactory class and, subsequently, all the ConcreteFactory classes.

We can solve this problem by eliminating the entire AbstractFactory/ConcreteFactory class structure and replace it with a single “Kit” class. This class, which we call ExtendableKit, is instantiated with a type parameter that determines which ConcreteProduct will be returned when an createProduct() method is invoked. The createProduct() methods are created by a higher-order function that takes all possible ConcreteProduct instances as arguments.

ExtendableKit solves the extendibility problem effectively. It should be noted that even though the the createProduct() methods are created dynamically, they can be created in a clear and maintainable manner. First, they can be created in a way to clearly communicate type information and enforce type-safety. Second, they can be defined at a single point in the program (possibly even in the class definition) to maintain readability and cohesiveness. The result is a design that is flexible and safe.
4.3 Forces

1. Interchangeable components are an important component in many software systems

2. Many times, interchangeable components are constructed from a family of sub-components; each subcomponent is instantiated independently

3. The construction process is often limited to use subcomponents from a single family

4. And yet, these families must be interchangeable

5. These families may need to be extended to include additional subcomponents later

6. These families may not have perfect subcomponent/interface isomorphism

[Unresolved]

4.4 Applicability

Use Extendible Kit in designs where the number and natures of ConcreteProduct classes is prone to change.

4.5 OO++ Interactions

A higher-order function is used to define the createProduct() methods of the ExtendableKit class. Once these methods are defined, they operate indistinguishably from “normal” methods. In other words, the functional components of this design are contained within the ExtendableKit class and are hidden from the Client.

This pattern extension requires the following language features:

1. First-class functions

2. Dynamic function definition
Figure 4.1: UML structure of AbstractFactory - Extendable Kit

4.6 Structure

Figure 4.1 captures the UML structure of this extension.

4.7 Extended Participants

- **ExtendableKit**

  - Replaces AbstractFactory and ConcreteFactory classes
  
  - Is instantiated with a parameter that specifies type; because the AbstractFactory class has been eliminated, the instances of ExtendableKit must be instantiated with typing information

  - Has an interface defined by a higher-order function

4.8 Collaborations

The ExtendableKit class must be fully defined before instantiation. After instantiation, it will function identically to a ConcreteFactory.
4.9 Consequences

The Extendable Kit pattern behaves almost exactly like the Abstract Factory pattern and has most of the same consequences. The major exception is that Extendable Kit can be easily fitted with new products.

4.10 Implementation

When implementing an Extendable Kit, there is no need for the AbstractFactory or ConcreteFactory classes. This considerably simplifies the design structure.

There are a few issues to consider during implementation.

1. *Is type-safety required?* Many OO++ languages have dynamic typing. An important aspect of the Abstract Factory is guaranteeing type safety. The dynamic methods of ExtendableKit may need to be written to also provide this guarantee.

2. *Determine if classes are “first-class citizens”*. If classes can be passed as data, the higher-order function that creates the createProduct() methods takes the classes as parameters. If, however, classes cannot be passed to a function, it may be necessary to pass prototypical objects that can self-replicate (see the Prototype pattern).

3. *Does the language support dynamic methods?* Just because first-class functions are supported does not mean that first-class methods are. A method has access to the private data of the object, but a function does not. If dynamic methods are not supported, dynamic functions can be used but must be designed with the capability to know the state of the ExtendableKit object to determine which type of product should be returned.
4.11 Sample Code

In our Python sample implementation, we illustrate *Extendable Kit* using first-class classes, dynamic methods, and dynamic typing.

Below is the code for the *ExtendableKit* class (the code for the actual products is not shown). We have chosen to guarantee type-safety.

class ExtendableKit:
    #private internal class
class __Enumerate:
    def __init__(self, collector):
        collector.append(self)

__allTypes = []
TYPE_ONE = __Enumerate(__allTypes)
TYPE_TWO = __Enumerate(__allTypes)
__allTypes = tuple(__allTypes) # makes __allTypes read-only

def __init__(self, kitType):
    # ensure that kitType is instance of __Enumerate
    if not isinstance(kitType, self.__Enumerate):
        raise Exception("Invalid Kit Type")
    self.__kitType = kitType

def HOF_makeCreateFunction(abstractProductClass, *concreteProductClasses):
    if len(self.__allTypes) != len(concreteProductClasses):
        raise Exception("Invalid number of products")
    for productClass in concreteProductClasses:
        if not issubclass(productClass, abstractProductClass):
            raise Exception("Invalid Type")
    def DYN_createNewProduct(self, *args):
        typeIndex = self.__allTypes.index(self.__kitType)
        concreteProductClass = concreteProducts[typeIndex]
        concreteProduct = concreteProductClass(*args)
        return concreteProduct
    return DYN_createNewProduct

createProductA = HOF_makeCreateFunction(ProductA, ProductA1, ProductA2)
createProductB = HOF_makeCreateFunction(ProductB, ProductB1, ProductB2)

kit = ExtendableKit(ExtendableKit.TYPE_ONE)
kit.createProductA()    # returns an instance of ProductA1

Notice that we can us subclasses for convenience.

class TypeOneKit(ExtendableKit):
    def __init__(self):
        ExtendableKit.__init__(self, ExtendableKit.TYPE_ONE)

kit = TypeOneKit()
kit.createProductA()    # returns an instance of ProductA1

4.12 Related Patterns

Extendable Kit is similar to other dynamic interface extensions (Iterator - Dynamic Interface, Proxy - Molded Proxy, etc.). The major difference is that the dynamic interface of the Extendable Kit pattern can be defined strictly at one location and at one time which is arguably more coherent and easier to maintain.
5 OO++ BUILDER EXTENSIONS

5.1 Base Pattern - Extension Name

Builder - Lazy Director

5.2 Motivation

The object-oriented Builder pattern has two basic components. The Director class, which contains the static build algorithm, and the ConcreteBuilder classes, which are interchangeable components that use the same build algorithm to produce different complex object products.

We can extend the Director class to produce a function containing build instructions. In other words, the Director builds a construction function. Now instead of containing a build algorithm, the Director class contains a build template. Actual construction occurs later.

For example, if the Director is responding to a specific input, the output function of the Lazy Builder is a representation of the input in a constructible format. Take, for example, an RTF document. A standard OO Director class will process the file and use a ConcreteBuilder to produce a single output. The Lazy Director processes the file producing a construction function with hard-coded build instructions for that specific document. Without further reference to the actual document, this construction function now represents the RTF document and can be used with various ConcreteBuilder instances to produce different outputs (like printing, saving, viewing, etc.).

5.3 Forces

- The build process for a given complex object may be variable
• A Client may be ready to define the build process but not ready to actually build

• The various build processes may need to be storable and reusable

5.4 Applicability

Use the Lazy Builder pattern extension

• when the build algorithm itself must be templetized

• when various build algorithms are produced and need to be stored for repeated use

5.5 OO++ Interactions

The pattern requires the use of dynamically created functions within the LazyDirector. When a LazyDirector constructs, it combines these dynamic functions into a new function that is returned to the Client. The Client is unaware of the dynamic functions and uses the construction function with a ConcreteBuilder just as it would normally use a Director.construct() method.

This pattern extension requires the following language features:

• First-class methods

• Dynamic function definition

5.6 Structure

Figure 5.1 captures the UML structure of this extension.

5.7 Extended Participants

• LazyDirector

  – Replaces Director class
Figure 5.1: UML structure of Builder - Lazy Director

- Has a `construct()` method, but does not require a `ConcreteBuilder`
- Produces a construction function that does require a `ConcreteBuilder`

- `constructionFunction()`

  - Produced by the `construct()` method of `LazyDirector`
  - Directs a `ConcreteBuilder` to build the `Product`

### 5.8 Collaborations

The Client uses a `LazyDirector` to construct the `constructionFunction()`. This process may require inputs or environmental data.

At a later time, the Client uses the `constructionFunction()` with a `ConcreteBuilder` to build a specific `Product`.

### 5.9 Consequences

The Lazy Builder pattern extension has two significant consequences.

1. *The definition of the variable parts of a build algorithm and the actual process of building are separated in time and possibly in space.* The process of defining the
build algorithm occurs at a separate time than the actual build event. However, it is also important to note that these two events can also be separated by space. That is, one client can create a `constructionFunction()` and then pass it to another client (who is oblivious to its details) for execution.

2. **Building a `constructionFunction()` is more complicated than simply directing a `ConcreteBuilder`.** The original `Builder` pattern establishes a static director and a variable builder. Making the `Director` to be variable introduces greater complexity to the pattern.

### 5.10 Implementation

The central challenge in implementing the `LazyBuilder` pattern is determining how to implement the creation of the `constructionFunction()`. There are two major issues.

1. **What determines the variability of the build?** Will the `LazyBuilder.construct()` method be passed parameters or will it have access to environmental data? In the RTF example, it seems likely that the RTF document is the input. In a videogame, however, the human player’s decisions may alter the environment that would subsequently determine how appropriate components and obstacles should be built.

2. **How should the `constructionFunction()` be built?** The `constructionFunction()` can be built in many ways. In our sample code below, we demonstrate storing the build functions in a list and defining a dynamic function that executes them in order. Another method would be function composition.
5.11 Sample Code

class HTML_Builder:
    def __init__(self):
        self.product=None

    def processText(self, textString):
        pass

    def process_a_tag(self, tag):
        pass

### Process All HTML tags...

    def process_unknown_tag(self, tag):
        pass

    def processEndTag(self, tag):
        pass

    def getHtmlProduct(self):
        return self.product

class HTML_PrintBuilder:
    def __init__(self):
        HTML_Builder.__init__(self)
        self.product = ""

    def processText(self, textString):
        textString = textString.strip()
        if len(textString) > 0:
            self.product += "TEXT: " + textString + "\n"

    def process_a_tag(self, tag):
        self.product += "LINK: " + tag + "\n"

### Process All HTML Tags....

    def process_unknown_tag(self, tag):
        self.product += "UNKNOWN TAG: " + tag + "\n"
def process_end_tag(self, tag):
    self.product += "END TAG: " + tag + "\n"

class LazyDocBuilder:
    def tokenize(self, document):
        tokens = []
        while len(document) > 0:
            spot = document.find("<")
            if spot > 0:
                nextToken = document[:spot-1]
                document = document[spot:]
            elif spot == 0:
                spot = document.find">
                if spot >= 0:
                    nextToken = document[:spot+1]
                    document = document[spot+2:]
                else:
                    nextToken = document + ">"
                    document = ""
            else:
                nextToken = document
                document = ""
            tokens.append(nextToken)
        return tokens

    def findEnd(self, token):
        # Find the end of a token

    def getFuncFor(self, token):
        if token[0] == "<":
            tagType = token[1:self.findEnd(token)]
            if tagType.lower() == "a":
                return lambda Builder:
                    Builder.process_a_tag(token)
            elif tagType.lower() == "br":
                return lambda Builder:
                    Builder.process_br_tag(token)
            elif tagType.lower() == "p":
                return lambda Builder:
                    Builder.process_br_tag(token)
return lambda Builder:

Builder.process_p_tag(token)

### Process all other HTML tags...
if token[0:2] == "<\":
tagType = token[2:self.findEnd(token)]
return lambda Builder: Builder.processEndtag(tagType)
else:
return lambda Builder: Builder.processText(token)

def construct(self, document):
tokens = self.tokenize(document)
lazyDocFuncList = [lambda Builder: Builder]
for token in tokens:
lazyDocFuncList.append(self.getFuncFor(token))
def lazyDocFunc(Builder):
    for f in lazyDocFuncList:
        f(Builder)
return lazyDocFunc

bulder = LazyDocBuilder()
htmlDoc = bulder.construct(open(someHtmlFileName).read())
htmlDoc(HTML_PrintBuilder) # This constructs a text representation
                        # Of the HTML Document someHtmlFileName

5.12 Related Patterns

Memento - Restoration Method also creates a function dynamically for clients
but its function is state-modifying while the constructionfunction() produced by
this extension produces a complex product.
6  OO++ FACTORY METHOD EXTENSIONS

6.1 Base Pattern - Extension Name

Factory Method - Non-Deterministic Constructor

6.2 Motivation

One might suggest that a constructor is a degenerate form of a Factory Method. That is, a constructor is a method that produces a single type of object, rather than a Factory Method that produces an instance of various possible object types. A Factory Method serves as an enhanced constructor that allows a client to create a specific object without knowing its concrete type. The Factory Method guarantees that the object is the right type within a given static context.

Non-Deterministic Constructor extends Factory Method just as Factory Method extends a constructor. It allows a client to produce an object without specifying (or even knowing) its concrete type in a dynamic environment. In other words, the type of object created by the Non-Deterministic Constructor can change. These changes might be triggered by the client (re)configuration or by environmental changes. This allows the client to be able to continue production of objects without having to worry about concrete types.

A possible scenario where this might be useful is in a computer combat simulation game. In games of this type, it is common for the AI and the player to control various types of military units. These units are often configurable. If the client producing the AI's pieces uses the Non-Deterministic Constructor pattern, unit construction can adapt to changes in player's strategy as the game progresses.

The Non-Deterministic Constructor has two additional benefits. First, back-
tracking allows multiple objects to be created from a single definition. Second, the definition is descriptive rather than explicit.

6.3 Forces

- Object construction may change during run-time
- Client instances creating the objects may not need or wish to be aware of the construction changes
- Client instances may be satisfied to describe the types of objects required rather than explicitly define them

6.4 Applicability

Use the Non-Deterministic Constructor pattern extension

- when the build type of objects may change at run-time and the Client creating the objects cannot or should not know the exact type of the objects being constructed
- when a constructor should produce multiple objects in a single call
- when you want to describe the object to be constructed instead of specifying it explicitly

6.5 OO++ Interactions

This pattern masks a logic relation as a constructor. This hybrid is accessed like a constructor but behaves like a logic relation.

This pattern extension requires the following language features:

- Logic-programming Relations
- First-class Classes
6.6 Structure

Figure 6.1 captures the structure of this extension in UML.

6.7 Extended Participants

- NonDeterministicConstructor
  
  - Rules can be altered/configured to change the types of objects being constructed
  
  - Knows all the different types to construct

6.8 Collaborations

The NonDeterministicConstructor is self-configured or client-configured. This configuration can be changed between calls to makeProduct(). Clients call the NonDeterministicConstructor to produce objects.

6.9 Consequences

The Non-Deterministic Constructor pattern extension has two significant consequences.

1. Clients are no longer required to know when to change types of objects constructed. They may or may not be required to describe the type of the object they would like constructed.
2. *Clients must determine what to do if the NonDeterministicConstructor produces nothing.* The `NonDeterministicConstructor` may produce zero, one, or multiple object instances. Generally, it will be the responsibility of clients to determine what to handle this variance.

### 6.10 Implementation

There are two challenges in the implementation of the Non-Deterministic Constructor pattern extension.

1. *Who defines the facts and rules?* It is possible to create a fully self-contained `NonDeterministicConstructor`. In this case, all of the rules are configured internally (and possibly reconfigured in response to messages passed to the object). In these cases, the `NonDeterministicConstructor` can be designed to respond to changes in environment or signals from other objects. Alternatively, rules and facts can be defined by clients who can directly configure the `NonDeterministicConstructor` with a composite rule. This scenario is illustrated in the sample code in the next section.

2. *How does object instantiation work?* One method is to store classes in the logic facts. When a match is found to the construction rule, a class is returned and the object instantiated. Another alternative is to use the `Prototype` pattern. In this scenario, a successful search returns a prototypical instance to create new objects. In either case, dealing with non-one (zero or many) matches requires tailoring the `NonDeterministicConstructor` to the requirements of clients.

### 6.11 Sample Code

In the code below, a skeleton implementation of the pattern is shown. The `NDC` is the Non-Deterministic Constructor. It’s implementation only produces a single
match (negating many of the benefits of logic design). However, producing multiple
matches is easily implemented and is not shown because of the design specific nature
of such an implementation.

class Unit(object):
    pass

class Infantry(Unit):
    pass

class Tank(Unit):
    pass

class Fighter(Unit):
    pass

class NDC(object):
    def __init__(self):
        self.constructionRule = Relation()

    def __call__(self, *args):
        classArg = LogicArg()
        if self.constructionRule.query(classArg):
            concreteClass = classArg.value()
            return concreteClass(*args)
        else:
            return None

    def createRule(self, ruleName):
        rules[ruleName] = Relation()
        return rules[ruleName]

    def rule(self, ruleName):
        return rules[ruleName]

    def setRule(self, rawRelation):
        self.constructionRule.inference = rawRelation

constructor = NDC()
Attack = constructor.createRule("Attack")
Defense = constructor.createRule("Defense")
Movement = constructor.createRule("Movement")
FlyingUnit = constructor.createRule("FlyingUnit")

Attack.ASSERT(Infantry,1)
Attack.ASSERT(Tank,4)
Attack.ASSERT(Fighter,2)
Defense.ASSERT(Infantry,1)
Defense.ASSERT(Tank,5)
Defense.ASSERT(Fighter,2)
FlyingUnit.ASSERT(Fighter)

constructor.setRule(lambda class:
constructor.rule("Attack").query(class, rating) &
GT(rating, 2) &
constructor.rule("Movement").query(class, movement) &
GT(rating, 1))

instance = constructor(args)

6.12 Related Patterns

The Non-Deterministic Constructor can be configured as an Observer. When
the observed Subject change, the NonDeterministicConstructor changes the ob-
jects it produces.
7 OO++ PROTOTYPE EXTENSIONS

7.1 OO++ Prototype Discussion

The Prototype pattern tends very strongly to an object-oriented only solution. The purpose of the pattern is to use objects to represent types rather than classes and to self-replicate these objects to instantiate a new type. Neither functional or logic programming paradigms tend to support types in a compatible manner.

A possible functional approach is to dynamically create functions that produces the correct object (or type) when executed. But this approach is not very different from implementing additional constructors or construction class methods. Additionally, it loses the ability of being able to clone an object that has accumulated state.

The guiding principles of logic programming also suggest nothing new. Prototype appropriate contexts necessitate a specific solution, so the non-deterministic principles of logic design do not apply.

Finally, neither the functional or logic programming paradigm provide any improvements in the existing Prototype structure. The primary weakness of the pattern, the difficulty in implementing a clone() operation, is not a problem that either functional or logic programming solves effectively.
8.1 OO++ Singleton Discussion

This GOF pattern is one of the few patterns where the class is designed to control access to itself and its objects. At present, there does not appear to be an effective way to improve this control with functional or logic programming. Neither paradigm appears to ameliorate the major difficulty of subclassing Singleton classes.

Not only does the Singleton pattern structure not benefit from these ancillary paradigms, but it also has fundamental incompatibilities with the philosophies behind them. For instance, in pure functional programming where side-effects are considered undesirable, this pattern (where only one instance can exist) is entirely not appropriate. Additionally, the concept of a Singleton is unique to object-oriented programming because other paradigms are more willing to accept global data and operations.
9.1 Base Pattern - Extension Name

Adapter - Dynamic Adaptation

9.2 Motivation

The Adapter pattern has two traditional variants: the Object Adapter and the Class Adapter. The Object Adapter simply wraps the Adaptee instance in an instance of Adapter for use with the Target. The Class Adapter uses multiple inheritance to mask the Adaptee into an Adapter. Each has different strengths and weaknesses. Among others, the Class Adapter cannot adapt subclasses of the Adaptee, and the Object Adapter cannot override Adaptee behavior. A full discussion is included in GOF’s presentation of Adapter.

An OO++ solution in a dynamic language is to actually modify the Adaptee instance’s interface. Methods can be dynamically assigned to the Adapter instance for use with the Target. A class can be constructed with two operations, one for adding the new methods and the other for restoring the Adaptee to its original interface.

This solution combines the advantages of both the Object Adapter and Class Adapter. The major drawback is that clients that maintain a reference to the Adaptee instance before it is modified may experience a change of behavior after adaptation.

9.3 Forces

• Some solutions require adaptation to be dynamically made

• The Adaptee instance may need to be “un-adapted”

• The Adaptee instance may need to pass object identity tests
• The **Adaptee** instance may need to simultaneously operate with multiple targets that expect different interfaces

### 9.4 Applicability

Use the Dynamic Adaptation extension

• When the **Adaptee** instances need to be adapted and un-adapted

• If the **Adaptee** instance must pass object identity tests

• If the **Adaptee** instance must work with multiple targets requiring different interfaces (this requires that the original methods of the **Adaptee** instance are partitioned and preserved)

### 9.5 OO++ Interactions

A **DynamicAdapter** class supports two methods that dynamically assign and remove new methods to an **Adaptee** instance. The class may need to store original methods of the class that are overwritten for restoration at un-adaptation time.

This pattern extension requires the following language features:

• Semantics for supporting dynamic method assignment to a class

### 9.6 Structure

UML digrams do not clearly describe the structure of this pattern. UML structure diagrams cannot illustrate a temporally transient interface like that seen in the Dynamic Adaptation pattern extension.

### 9.7 Extended Participants

• **DynamicAdapter**

  – Dynamically extends or restores the interface of an **Adaptee** instance.
9.8 Collaborations

The DynamicAdapter extends the interface of the Adaptee instance. The Adaptee instance is then passed to the Target for their interactions. After the Target is no longer actively interacting with the Adaptee instance, the DynamicAdapter removes the extended interface.

9.9 Consequences

The Dynamic Adapter extension has four significant consequences.

1. *Adaptee* instances can be “un-adapted”. If the changes required are temporary, they can be undone.

2. *Adaptee* instances can be used simultaneously with targets that expect different interfaces. If the expected interfaces are completely partitioned (no two targets expect the same interface with different behavior), an Adaptee can be adapted to several targets and used by them simultaneously. This may be particularly advantageous if the Adaptee instance is a singleton.

3. *Adaptee* instances will pass object identity tests. Because the object instance is still the same, it can be tested for identity.

4. If *Adaptee* methods are overwritten, then clients interacting with the original interface may fail after adaptation. If a client is using the original interface of an Adaptee instance and that instance’s methods are overwritten, the client’s interactions with the instance will probably fail.

9.10 Implementation

Creating the DynamicAdapter class is straight-forward in a dynamic language. There are two major issues.
1. How will methods be restored? In Python, a straightforward solution is to use a dictionary to map an object to its original methods.

2. Invalidation of modified methods. Depending on the design solution, there may be a way to invalidate connections between Clients and Adaptee instances if the instance’s methods have been modified. In some languages (like Python), it is possible to use Proxy classes to manage invalidation. However, there are no general solutions.

9.11 Sample Code

class Adaptee:
    def interface1(self):
        print 'Adaptee interface1'

    def interface2(self):
        print 'Adaptee interface2'

class AdapteeAdapter:
    def __init__(self):
        self.adaptations = {}

def convert(self, adaptee):
    self.adaptations[adaptee] = [adaptee.interface1, adaptee.interface2]
    storeInterface1 = adaptee.interface1
    storeInterface2 = adaptee.interface2
    def DYN_newInterface1(self, adaptedArg1):
        print 'Adapted Interface1', adaptedArg1
        print 'calls adaptee interface 1'
        storeInterface1()
    def DYN_newInterface2(self):
        print 'Adapted Interface2'
        print 'calls adaptee interface 1 and 2'
        storeInterface1()
        storeInterface2()
    adaptee.interface1 = new.instancemethod(DYN_newInterface1, adaptee)
    adaptee.interface3 = new.instancemethod(DYN_newInterface2, adaptee)
def undo(self, adaptee):
    (original1, original2) = self.adaptations[adaptee]
    adaptee.interface1 = original1
    del adaptee.interface3

adapteeInstance = Adaptee()
adapter = AdapteeAdapter()
adaptedInstance = adapter.convert(adapteeInstance)

9.12 Related Patterns

The Decorator - Double-Dispatch extension modifies the behavior of an interface without directly modifying the interface itself. However, it is conceivable that the adapter could also be fitted with a double-dispatch variation to handle multiple adaptations on a single object.

The Molded Proxy pattern extension is the reverse of Dynamic Adaptation. While both dynamically create an interface, Molded Proxy attempts to use the same interface as the object it proxies for while modifying behavior. Adapter - Dynamic Adaptation attempts to modify the interface without changing behavior.
10.1 Base Pattern - Extension Name

Bridge - Relational Junction

10.2 Motivation

The original Bridge pattern is designed (generally) for a single abstraction class and a single implementation class to be used together. However, under some circumstances, an abstraction may want the implementation to vary dynamically based on various previously defined criteria. Under these circumstances the Bridge pattern behaves more like a Strategy pattern except that the behavior is not what changes, but the implementation of the behavior.

A possible use for such a structure might be remotely executed code. If, for example, the abstraction class hierarchy is used for drawing graphical windows and the implementation classes draw such windows under various platforms, then transmitting code to be executed on another machine requires that the abstraction class must be refitted with a new implementation object upon arrival. On the other hand, using a junction point that “floats” between implementation classes based on some set of rules, the entire code component can be transferred between multiple platforms without modification.

However, the previous example could be solved with a simple look up table (a non-relational junction). The power of a relational solution is in the rules and facts that can be defined and described. A better example of the Relational Junction might be computational abstract classes and implementation classes that perform computations with varying orders of magnitude in time and space. Logical rules
that describe these implementations (in terms of speed, memory requirements, etc.) can be used at run-time to determine which implementation should be used. These rules could be extended, based on the needs of the client, to group implementations according to heat, battery usage, and so forth. All of the complexity of these decisions is hidden from the client.

Another possible use for this type of pattern is when the various operations of the Abstraction classes can vary from one another in terms of their implementation. In this scenario, each method uses relational facts and rules to determine the correct implementation for that individual method.

10.3 Forces

- In highly variable environments, the needed implementation for certain classes of operations can vary during execution

- All classes within a hierarchy may require the implementation to switch simultaneously

- Different methods may wish to use different implementations from one another

10.4 Applicability

Use the Relational Junction extension

- When the Implementation class to be used by an Abstraction class may vary at run-time.

- When the Implementation class to be used by each method varies and the variance can be described relationally.

10.5 OO++ Interactions

The Abstraction base class defines and implements an implementation switching scheme. Alternatively, each method in the ConcreteAbstraction class can define a
switching scheme for itself. The switching scheme is described by logical rules that are passed to the Relational Junction to obtain the appropriate implementation instance. The Relational Junction processes the logical rules through its relational tables. The implementation classes are responsible for describing themselves to the RelationalJunction.

This pattern extension requires the following language features:

- Relational rules and facts

10.6 Structure

Figure 10.1 captures the UML structure of this extension.

10.7 Extended Participants

- RelationalJunction
  - Connects the Abstraction and ConcreteImplementation interfaces through relational rules and facts
– Accepts rules defined by Client classes or the Abstraction classes to look for appropriate implementation classes
– Accessed by the implementation classes for defining facts about themselves
– Creates instances of ConcreteImplementation classes when needed

10.8 Collaborations

Abstraction classes or methods define rules for their implementation needs at a given point in time. After determining what they require, they submit these rules to the RelationalJunction for application. The RelationalJunction applies the rules defined by the Abstraction classes to its own facts and rules searching for a solution. After a solution is found, it is returned to the Abstraction class or method.

10.9 Consequences

The Relational Junction extension has 2 significant consequences.

1. Both Abstraction and ConcreteImplementation classes must provide additional information. This extension requires more information be provided from both types of classes. The Abstraction classes must provide the rules for searching for implementations (alternatively the Client can do this in their place for a global setting) and the ConcreteImplementation classes must provide facts (and possibly rules) about themselves.

2. Overhead from relational lookup may invalidate the pattern in highly dynamic circumstances. Depending on the complexity of the rules and facts, the lookup cost may be prohibitive, especially if each method call looks up an implementation each time it is called. This pattern extension is, of course, highly theoretical and requires “shake-down” time to determine value and appropriate use.
10.10 Implementation

There are two major implementation issues.

1. Default cases. Generally, the Abstraction classes will require a solution to be found. If a search fails, who handles the failure? The Abstraction classes could assign themselves a default case, or the Junction could be configured with a default implementation when it cannot find a solution to a search.

2. Who defines the search rules? Client or Abstraction classes can define the search rules. It is possible that it is best for the Client to define the rules if the implementation is to apply to all implementation classes simultaneously. On the other hand, each Abstraction class (or method within that class) might define its own rules based on individual needs and circumstances.

10.11 Sample Code

```python
class Junction:
    def __init__(self):
        pass

    def findImplementation(self, rule):
        implementation = LogicVar()
        if rule.query(implementation):
            print 'Found Implementation'
        else:
            print 'need to assign default implementation'
        return implementation.value()

theJunction = Junction()
theJunction.speedRelation = Relation()
theJunction.memoryRelation = Relation()
```

```python
class BaseImplementation:
    def __init__(self):
        pass
```
def reset(self):
    pass

def swap(self, el1, el2):
    pass

def isSorted(self):
    pass

class ArrayImplementation:
    def __init__(self):
        self.__array = []

    def reset(self, elements):
        self.__array = elements[:]

    def swap(self, el1, el2):
        temp = self.__array[el1]
        self.__array[el1] = self.__array[el2]
        self.__array[el2] = temp

    def getElement(self, el):
        return self.__array[el]

    def toElements(self):
        return self.__array[:]

theJunction.speedRelation.ASSERT(ArrayImplementation,"reset","O(1)"
theJunction.speedRelation.ASSERT(ArrayImplementation,"swap","O(1)"
theJunction.speedRelation.ASSERT(ArrayImplementation,"getElement",
  
"O(1)"
theJunction.speedRelation.ASSERT(ArrayImplementation,"toElements",
  
"O(n)"

theJunction.memoryRelation.ASSERT(ArrayImplementation,"reset","O(n)"
theJunction.memoryRelation.ASSERT(ArrayImplementation,"swap","O(1)"
theJunction.memoryRelation.ASSERT(ArrayImplementation,"getElement",
  
"O(1)"
theJunction.memoryRelation.ASSERT(ArrayImplementation,"toElements",
  
"O(n)"
class AbstractSort:
    def implementation(self):
        if self.__imp == None:
            impClass =
            theJunction.findImplementation(self.impRule)
            self.__imp = impClass()
        return self.__imp

    def __init__(self, elements):
        self.__imp = None
        self.implementation().reset(elements)
        self.count = len(elements)

    def swap(self, el1, el2):
        self.implementation().swap(el1, el2)

    def isSorted(self):
        for count in range(1, self.__count):
            if (self.implementation().getElement(count) <
                self.implementation().getElement(count-1)):
                return False
        return True

    def sort(self):
        pass

    def toElements(self):
        return self.implementation().toElements()

class BubbleSort(AbstractSort):
    def __init__(self, elements):
        self.impRule = Relation()
        self.impRule.rule = lambda implementation: (
            theJunction.speedRelation.query(implementation,
            "getElement","O(1)"))
    AbstractSort.__init__(self, elements)

    def sort(self):
        for count1 in range(0, self.count-1):
for count2 in range(count1+1, self.count):
    if (self.implementation().getElement(count1) >
        self.implementation().getElement(count2)):
        self.implementation().swap(count1,
        count2)

if __name__ == '__main__':
    sort = BubbleSort([3,2,1])
    sort.sort()

10.12 Related Patterns

The RelationalJunction class might be defined as a Singleton.

This extension crosses over into the realm of the Strategy Pattern.
11.1 Base Pattern - Extension Name

Composite - Descriptive Instantiation

11.2 Motivation

Structures that implement the Composite pattern are very common. One difficult aspect of this pattern is the actual process for creating the composed structure. This composition process is may be done at instantiation of the composite object or later on during execution. In either case, the imperative step-by-step methods of composing children into the Composite can be tedious.

Logic programming is very descriptive in nature and offers a nice interface for defining composite relationships as well as a few related benefits. First of all, the logic programming interface supports variation (through the “or” operation) allowing multiple types of composites to be constructed from a single call. Secondly, logic programming statements can be used for verifying the “type” of the composite as well as creating it. This allows the logic operations to partition types of composites. This “dynamic” creation of types may serve as an alternative to the Prototype pattern.

11.3 Forces

- Creating composite structures can be complicated
- Composite structures are very common

11.4 Applicability

Use the Descriptive Instantiation extension

- Creation of composite structures is common throughout the design solution
• The same creation patterns occur repeatedly

• A composite can be described by simple and/or combinations

• Multiple composites should be created by a single description

11.5 **OO++ Interactions**

This pattern extension requires significant object-oriented/logic hybrid combinations. Many languages may not support these. The primary combination is an extension to the normal logic relation to handle object creation. Additionally, the backtracking mechanism must know how to “undo” the previous construction before proceeding down a new path. More information is provided in the implementation section about implementing this in Python.

This pattern extension requires the following language features:

• A logic/rule system that can be extended to create objects when rules evaluate to true and undo this creation when backtracking occurs.

11.6 **Structure**

The UML diagram for this extension is equivalent to the original structure presented in [2]. This pattern extension alters the construction of the Composite, not the structure of the Composite itself.

11.7 **Extended Participants**

• Composition System

  – Combination of relational rules and facts

  – Extended semantics for object construction

  – Composite may be composed by Composition System
11.8 Collaborations

Clients use the Composition System to define composite creation rules. These rules create complex objects and (later) test them to see if they match the creation rules.

11.9 Consequences

The Descriptive Instantiation extension has 3 significant consequences.

1. *It requires (possibly) complex changes to the relational system.* If the relational system is not extendible (or not easily extendible) than this pattern extension is not suitable. The more seamless the system, the greater the complexity of extensions. The benefits from describing the composites must outweigh the complexity of defining the system. The sample code below demonstrates a minimal implementation of the hybrid combination.

2. *It allows the composite structure to be described and the description stored.* The logical rules are sufficient for describing many composite structures. The easiest example is composites whose sole interface with other components are `addChild()` `removeChild()` methods. The `addChild()` method is easily representable with ‘&’. By wrapping these descriptions to be stored in relational rules, common composite types can be easily recreated. Logic programming generally supports set or test semantics on the logical operators depending on if the variable passed to the operator is bound. This means that using a stored description, it is possible to test a variable to see if it meets the logical rules. This is akin to type checking.

3. *It allows multiple composites to be created from a single rule.* Using the ‘—’ operator, a logical rule can be defined that can create two separate composites.
This can simplify composites whose structures are mostly similar but vary at a few junction points.

11.10 Implementation

Implementing the structure to support descriptive composition is complex. There is a single major issue is how should the extended relational system be defined? In our sample code, we use a system that wraps various relationships into a class, masking the logic structure behind object-oriented code. The code is complex and very Python-specific. It is likely that the implementation of this extension will be unique in every OO++ language.

11.11 Sample Code

```python
class FoodRelation:
    def __init__(self):
        self.basefacts = Relation()
        self.composites = {}

    def __createChildren(self, compositeLVar, lVars):
        compositeLVar.value().children = []
        for logicVar in lVars:
            compositeLVar.value().addChild(logicVar.value())
        return True

    def __call__(self, foodType, assignment):
        if self.composites.has_key(foodType):
            (relation, argCount) = self.composites[foodType]
            iValue = LogicVar.reduceLV(assignment)
            if isinstance(iValue, ComplexFoodObject):
                return relation.query(*iValue.children)
            else:
                lVars = []
                for x in range(0, argCount):
                    lVars.append(LogicVar())
                tempRule = (relation.query(*lVars) &
                            EQ(assignment, ComplexFoodObject(foodType)) &
                            Relation.doFunctionOnce(self.__createChildren, (assignment, tuple(lVars))))
```

120
return tempRule

else:
    cost = LogicVar()
    tempRule = (self.basefacts.query(foodType, cost) &
                EQ(assignment, SimpleFoodObject(foodType, cost.value())))
    return tempRule

def make(self, foodtype):
    obj = LogicVar()
    if self(foodtype, obj):
        return obj.value()
    else:
        return None

def makeAll(self, foodtype):
    obj = LogicVar()
    i = self(foodtype, obj)
    while i:
        yield obj.value()

def define(self, name, costOrComposite):
    if isinstance(costOrComposite, types.LambdaType):
        tempRelation = Relation()
        tempRelation.rule = costOrComposite
        argCount = costOrComposite.func_code.co_argcount
        self.composites[name] = (tempRelation, argCount)
    else:
        self.basefacts.ASSERT(name, costOrComposite)

def isType(self, instance, instanceType):
    if self.composites.has_key(instanceType):
        (relation, argCount) = self.composites[instanceType]
        return relation.query(*instance.children)
    else:
        return self(instance.name, instance.cost)

Food = FoodRelation()

Food.define("Beef Patty",".25")
Food.define("Chicken Patty",".20")
Food.define("Lettuce",".05")
Food.define("Tomato",".05")
Food.define("Ketchup",".05")
Food.define("Buns",".05")
Food.define("Cheese",".10")

Food.define("meat", lambda meat: (Food("Beef Patty",meat) | Food("Chicken Patty", meat)))

Food.define("burger",lambda buns, meat: (Food("Buns",buns) & Food("meat",meat)))

Food.define("cheese burger", lambda cheese, burger: (Food("Cheese",cheese) & Food("burger",burger)))

cb = Food.make("cheese burger")
b1 = Food.make("Buns")
if Food("Buns",b1):
    print 'b1 is buns'
else: print 'not buns'
if Food("cheese burger", cb):
    print 'cb Is cheese burger'
print cb

for cbInstance in Food.makeAll("cheese burger"): print "instance ", cbInstance

11.12 Related Patterns

This pattern extension is very similar to Interpreter - Descriptive Definition.
12.1 Base Pattern - Extension Name

Decorator - Double Dispatch

12.2 Motivation

The Decorator pattern allows an object to be dynamically assigned additional responsibilities by wrapping objects within other objects. In a language that supports functional-like semantics, a similar pattern exists for composing functions. Function composition could be seen as decoration for functions.

Within an OO++ context, simple functional composition has a significant weakness. Who maintains the composed function? If it replaces a method of an object, than the old undecorated functionality is lost (the Decorator pattern does not have this problem because the decorated instance is not the same as the original instance). This problem is pronounced when there are several clients operating on the same decorated object and each one requires different decorated functionality. On the positive side, composing functions for decorative purposes does maintain object identity which decoration through object-wrapping does not. (It also eliminates the Decorator class hierarchy which may or may not be a good thing).

Within an OO++ language, we can combine the functional and object-oriented approaches to decoration in a way that preserves object identity but still allows for different (decorated) behavior depending on the client. This multiparadigm solution basically creates a “double dispatch” system wherein behavior is determined by two types of receivers instead of one. While some languages support double dispatch
directly, this pattern extends such functionality to OO++ languages that support
functions as first-class citizens.

In this pattern, an object supports a method that can be decorated. A client then
specifies decorations to be applied to that method when called by that client. If the
client decorates a method it has already decorated, the result is cumulative. Calls
to the decorated method require that the client be passed as the first argument to
trigger the double-dispatch. In the end, each client gets the behavior it decorated all
while preserving object identity on the decorated instance. It should be noted, that
the default pattern extension only extends a single method, but it could be modified
to extend the entire interface.

12.3 Forces

- Decoration of some objects is limited to a single function

  - Client instances decorate an object differently depending on their needs

  - It may be advantageous to preserve object identity on decorated objects

12.4 Applicability

Use the Double Dispatch extension

- Decoration interface is limited to a single function

- Multiple Client instances will decorate and access the same instance of an object

- Object identity should be preserved on the decorated object

12.5 OO++ Interactions

This is a very multiparadigm pattern. The decorations of the methods of the
object are functionally composed. The double dispatch system binds Client instances
to decorated functions (which may be functions or callable objects depending on
Figure 12.1: UML Structure of Decorator - Double Dispatch

implementation). Finally, the actual calls to the decorated operations appear entirely object-oriented to external Client instances.

This pattern extension requires the following language features:

- First class functions (preferably, support for first-class methods)

- First class classes

12.6 Structure

Figure 12.1 captures the structure of this extension in UML.

12.7 Extended Participants

- DecoratableFunction
  
  - A callable object that wraps the decorated operation
  
  - To clients, it appears like a normal function
  
  - Offers methods for decorating the original operation
  
  - Requires the Client instance as an argument for double-dispatch

- Decorator
  
  - A function or callable object
– Adds functionality (responsibilities) to another (possibly) decorated function

– Only assumes a function interface in the naive implementation, but may be modified to only decorate certain types of callable objects

12.8 Collaborations

First, the object is made available for decoration and access to potential Client instances. Each Client than decorates the object appropriately and makes calls to the object normally (except that the Client instance must be passed as the decorated operation’s first parameter).

12.9 Consequences

The Double Dispatch extension has 2 significant consequences.

1. Object identity is preserved. Unlike the traditional Decorator pattern, the object is never wrapped within another object.

2. Type clarity is reduced. The price of preserving object identity is a loss of clear type information. Code maintainers reviewing the code must now understand the type of the object as well as the type of the method depending on the caller.

12.10 Implementation

Except for two significant issues, the implementation of this extension is straightforward.

1. How is the operation made decoratable? Our implementation below uses a callable object that handles the double dispatch, storage of the decorators, etc. This is probably the easiest implementation if the language supports callable objects being used in the place of methods.
2. **Possible restrictions on decoration.** In the original Decorator pattern, the Decorator class can only decorate classes with which it shares a common ancestor. In this extension, any callable object or function can theoretically decorate an operation. If restrictions are needed, the operation can be wrapped in a callable object that shares a common ancestor with a Decorator class. Using this scheme, some type-checking can be enforced to control access to decoration of classes and methods.

### 12.11 Sample Code

```python
class SomeReadClass:
    def read(someFile):
        # ...

class DecoratableMethod:
    def __init__(self, method):
        self.__baseMethod = method
        self.__types = {}

    def decorate(self, caller, decorator):
        curMethod = self.__types.get(caller, self.__baseMethod)
        self.__types[caller] = compose(decorator, curMethod)

    def __call__(self, caller, *args):
        theFunction = self.__types.get(caller, self.__baseMethod)
        return theFunction(*args)

    def getFunction(self, caller):
        return self.__types.get(caller, self.__baseMethod)

r1 = someReadObject()
r1.read = DecoratableMethod(r1.read)
r1.read.decorate(someCaller, decorator1)
r1.read.decorate(otherCaller, decorator2)
r1.read.decorate(otherCaller, decorator3)
r1.read(someCaller, filename)
```
12.12 Related Patterns

This pattern extension is similar to others that accept first-class functions for alteration of behavior (e.g., *Iterator - Dynamic Interface*). However, the goal of this pattern is decorated (or extended) behavior determined by the caller of the operation.
13  OO++ FACADE EXTENSIONS

13.1 Base Pattern - Extension Name

Facade - Relational Parameters

13.2 Motivation

Compilation can often be achieved with various optimizations for speed or memory, etc. If a compile() request is sent to a Facade object, optimization parameters may require a more intricate knowledge of the subsystems than is desirable. For example, certain optimizations are based on parameters passed to the CodeGenerator component that, if required of the client, break down some of the encapsulation of the Facade.

We can solve this problem by creating interchangeable components within the Facade, each of which is tagged with facts and rules usable within relational constructs. These facts and rules are used to create implementation independent parameters that can be used to configure the correct component to serve as the subsystem. The interchangeable components might be separate objects, or calls to objects configured with separate parameters.

One nice feature of this solution is that client can extend the rules for the Facade by combining the rules provided by the Facade into new rules that represent additional parameters. If in the compiler example, for example, the Compiler facade provides rules for fast compilation and memory efficient compilation, then a client can build these two rules into a new synthesis that is moderately fast and moderately memory efficient. In all cases, the client remains blissfully unaware of the nature and interface of the subsystems in use within the Facade.
13.3 Forces

- **Facade** pattern’s purpose is to simply the interface of a complex system of subsystems

- This purpose is broken when parameters are tightly coupled with a subsystem

- Relational rules are descriptive and can be defined in one context and interpreted in another

13.4 Applicability

Use the Relational Parameters extension

- When the **Facade** pattern is broken by parameters that require a detailed understanding of individual subsystems.

- When the **Facade** might use one of many interchangeable subsystems.

13.5 OO++ Interactions

This pattern extension requires that clients of the **Facade** use and access relational rules and facts for use as parameters. These rules are passed within the **Facade** to a logic programming style system that translates the rules into the appropriate subsystem to use or the appropriate parameters to pass to the subsystem.

This pattern extension requires the following language features:

- Relational rules and facts

- Storage of object instances, functions, and/or parameter sets within relational tables

- Dynamically assigned relational rule within another rule
13.6 Structure

The original pattern structure presented in [2] is sufficient. This pattern extension alters configuration of the structure, without altering the structure itself.

13.7 Extended Participants

- SubsystemManager
  - Logic-programming style structure
  - Accepts other relational rules as a parameter and uses it in the search for the appropriate subsystem and/or subsystem configuration

- RelationalParameter
  - May be provided by the Facade/SubsystemManager, or may be created by the Client
  - Describes system configurations

13.8 Collaborations

The Facade is defined, either at instantiation or later, with various subsystems. The SubsystemManager maps various relational rules to the subsystems. These rules (or at least a subset of them) are provided publicly to Client classes.

When a Client accesses the Facade, it can use the publicly provided rules as parameters for system configuration. Additionally, the Client can use these publicly provided rules to build more complex and/or more specific rules for its own purposes. When sending a request to the Facade, it sends a RelationalParameter. The Facade (using the SubsystemManager) determines the appropriate subsystem to use and the appropriate subsystem parameters to apply.
13.9 Consequences

The Relational Parameters extension has 3 significant consequences.

1. *It allows implementation independent parameterization of the Facade.* The Facade provides the parameters with out disclosing how they will be used.

2. *It enables the use of interchangeable subsystems.* Multiple variants of a subsystem can be registered or defined within the Facade, each with different descriptions attached. Because the parameters of the system are implementation independent, they can be mapped to interchangeable subsystems without concern for compatibility (the compatibility is part of the mapping to each subsystem).

3. *Extending parameters is still limited to those provided by the Facade.* Even though the rules provided by the Facade can be used to create new rules, they are the only true building blocks for parameterization. If the Compiler Facade does not provide a rule for memory optimization, a Client cannot create one without subsystem knowledge.

13.10 Implementation

Creating a Facade with Relational Parameters can be done in three stages. First, the conceptual Facade is laid out. This stage defines the interactions of the subsystems and the facade interface. Second, the interchangeable subsystems are defined and their associated parameters. Finally, the SubsystemManager is added to manage the relationship between the parameters and the subsystems.

In crafting these components, there are 4 major issues.

1. *Are the interchangeable subsystems logical or physical?* The same subsystem with one set of parameters might be viewed as a logical partition from the same
subsystem with other parameters. Physically interchangeable subsystems might be created using subclassing.

2. What parameters should be defined? These are the building blocks of later parameters and should be defined with care. What parameters are defined should be based on the different types of interchangeable components.

3. How should the parameters map to subsystems? A number of the parameters might map to all the possible subsystems with an ordering (the fast parameter in the compiler example, for instance) while others would only map to a few.

4. How should the Facade react to failures in parameters? There is no guarantee that parameters will succeed. The compiler example might be designed to fail with an error message describing the failure while other systems might warrant a default case.

13.11 Sample Code

In our simple system below, the relationships defined for each of the subsystems are qualitative. The subsystems are ordered from best to worst of the given quality. Our implementation requires that the relational system preserve ordering.

class CompilerSubsystemManager(object):
    def __init__(self, *parameterNames):
        self.parms = {}
        self.parms['default'] = Relation()
        for name in parameterNames:
            self.parms[name] = Relation()

    def getRelation(self, key):
        return self.parms.get(key, None)

    def defineSubsystem(self, subsystemKey, **parameters):
        #print parameters
for parmName in parameters.keys():
    parmList = parameters[parmName]
    #print parmList
    for realParm in parmList:
        self.parms[parmName].ASSERT(subsystemKey, realParm)

def get(self, subsystemKey, rule):
    realSubsystem = LogicVar()
    if rule.query(subsystemKey, realSubsystem):
        return realSubsystem.value()
    elif self.parms["defualt"].query(subsystemKey, realSubsystem):
        return realSubsystem.value()
    else:
        raise Exception("Could not get subsystem")

class CompilerFacade(object):
    def __init__(self):
        gen1 = Generator("Type1")
        gen2 = Generator("Type2")
        gen3 = Generator("Type3")
        scan1 = Scanner("Type1")
        scan2 = Scanner("Type2")
        self.manager = CompilerSubsystemManager("compilespeed",
                                               "codesize","compilememory")
        #gen1 is fastest compile speed, gen3 slowest, etc.
        self.manager.defineSubsystem("generator",
                                     compilespeed=(gen1, gen2, gen3),
                                     codesize=(gen2, gen3, gen1),
                                     compilememory=(gen3, gen2),
                                     default = (gen1,))
        self.manager.defineSubsystem("scanner",
                                     compilespeed=(scan1, scan2),
                                     codesize=(scan1, scan2),
                                     compilememory=(scan2, scan1),
                                     default = (scan1,))
        self.CompileSpeed = self.manager.getRelation("compilespeed")
        self.CompileMemory = self.manager.getRelation("compilememory")
        self.CodeSize = self.manager.getRelation("codesize")
134
def compileCode(self, code, optimizationParameter=None):
    scanner = self.manager.get("scanner", optimizationParameter)
    builder = ProgramNodeBuilder()
    parser = Parser()

    parser.parse(scanner, builder)

    output = ByteCodeStream()
    generator = self.manager.get("generator", optimizationParameter)
    parseTree = builder.getRootNode();
    parseTree.traverse(generator)
    return output

compiler = CompilerFacade()

midpoint = Relation()
midpoint.rule = lambda key, subsystem:
    compiler.CompileSpeed.query(key, subsystem) &
    compiler.CompileMemory.query(key, subsystem) &
    compiler.CodeSize.query(key, subsystem))

compiler.compileCode(codeAsText, compiler.CompileSpeed)

13.12 Related Patterns

The Facade - Relational Parameters extension is relatively unique. While it is similar to other descriptive pattern extensions (e.g. Factory Method - Non-Deterministic Constructor), it is unique in its use of descriptive relations as parameters.
14.1 Base Pattern - Extension Name

Flyweight - Dynamic Pseudo Instances

14.2 Motivation

Consider a computer game wherein a player may have hundreds, or even thousands, of playing pieces. Despite this vast number, there are less than twenty types. Considering that each piece has a significant amount of intrinsic state, sharing (the Flyweight pattern) is optimal and is used to implement the pieces. Extrinsic state (consisting of location, status, and moves remaining) is stored in a single table by a managing object within the game.

However, under certain circumstances within the game, it might be optimal not to share. Under these circumstances, a client may wish to maintain a pointer to a specific instance for a short while. In these limited relationships, it would be optimal for a unique instance to exist for the duration of the relationship (defined by the client) and then released, allowing the playing piece to resume existing as shared data.

In this scenario, a unique instance would be defined by the combination of the shared object and the extrinsic state data. We might use the Proxy pattern to create the temporary unique instance. If the Proxy instance stores the extrinsic state and points to the correct shared object, it behaves like a unique instance. Assuming that the extrinsic state is updated automatically from the managing object, the Proxy does not even have to be responsible for state changes. Its sole purpose is to serve as an object-identifier.
However, the extrinsic state may not be updated automatically. The client accessing the **Proxy** may be computing the current extrinsic data with help from the managing object as needed. Now, the naive **Proxy** pattern is unable to operate properly if any aspect of extrinsic state changes.

A **Proxy** with a dynamic interface can solve this problem through function composition. If the client provides extrinsic state through a function it can be designed as the first argument to the flyweight methods that require extrinsic state. The composed function can be assigned to the **Proxy** instance and the methods that do not require extrinsic state can be assigned directly.

In the game example, consider a friendly playing piece at some two-dimensional location. For its own purposes, it may wish to keep pointers to the adjacent enemy playing pieces. It can get the initial extrinsic data from the managing object and construct the proxies. The friendly playing piece provides a `getExtrinsic()` method to the proxies to provide them with appropriate extrinsic state (assume that the friendly piece has something to contribute to the state of the other pieces). When an adjacent enemy piece moves a square, the managing object alerts the friendly piece that passes the appropriate information on through `getExtrinsic()` and the proxies update transparently.

### 14.3 Forces

- Shared objects do not have individual identity except with context (extrinsic state)

- For convenience and clarity of design, some circumstances may benefit from temporary unique identifiers representing shared objects with context

- These context may not be computable by the temporary unique identifier alone
14.4 Applicability

Use the Dynamic Pseudo Instances extension

- When Client instances need temporary pointers to “unique” instances of shared data

- And when a proxy cannot store extrinsic data or it cannot access a function for obtaining the extrinsic data a priori.

14.5 OO++ Interactions

The DynamicProxy object will generally be a blank class (no interface). Methods are assigned from the shared object or composed from the shared object and the function providing the extrinsic data.

This pattern extension requires the following language features:

- Function Composition

- Dynamic Method Assignment

14.6 Structure

Figure 14.1 captures the structure of this extension in UML.

14.7 Extended Participants

- DynamicProxy
  - Acts as the unique instance of the shared data
  - Interface is created automatically
14.8 Collaborations

A Client creates a DynamicProxy for some instance with which it desires a limited relationship. The Client is also probably computing and passing extrinsic data to the DynamicProxy through one of its methods. The interface of the DynamicProxy is created dynamically by assigning methods from the shared object to the proxy and also from composing functions.

14.9 Consequences

The Dynamic Pseudo Instances extension has one significant contribution: short time object instances. The goal of the flyweight pattern is to share as much as possible to eliminate unnecessary object creation. This pattern extension allows objects to be created from the shared objects when needed for limited relationships. Thus memory is conserved but instances are available when needed.

14.10 Implementation

There are 2 major implementation issues.

1. Creating the DynamicProxy interface. The solution presented below is to copy
every method from the shared object to the Dynamic Proxy and then override
the necessary methods. This is done in the DynamicProxy constructor.

2. Composing the extrinsicState with the sharedObject methods. It may be neces-
sary to use currying to setup the extrinsicState method so that it does not need
a parameter passed to it.

14.11 Sample Code

```python
def getExtrinsicData(self, locationX, locationY):
    exData = manager.getExtrinsicData(locationX, locationY)
    # Add to extrinsic data
    return exData

def getAdjacentPieces(self, ...):
    # get all extrinsic states from manager
    # for states
    proxy = copyInterface(sharedObject)
    proxy.operation = compose(sharedObject.operation,
                               self.getExtrinsicData)
```

14.12 Related Patterns

This extension is a specialized Proxy - Molded Proxy.
15  

**15.1  Base Pattern - Extension Name**

Proxy - Molded Interface

**15.2  Motivation**

The motivation for the *Flyweight - Dynamic Pseudo Instance* pattern extension is also motivation for *Proxy - Molded Interface*.

**15.3  Forces**

- Some circumstances are appropriate for the *Proxy* pattern but the exact implementation details are not known a priori

**15.4  Applicability**

Use the *Molded Interface* extension when some or all of the proxy’s methods must be defined on the fly.

**15.5  OO++ Interactions**

The *MoldedProxy* instance is an object. Some or all of its methods are defined at instantiation through dynamic function definition, currying, composing, etc.

This pattern extension requires the following language features:

- Dynamic function definition

- First class functions

**15.6  Structure**

The structure of a *Molded Interface* is completely dynamic and cannot be illustrated statically. All (or most) of the methods of the *Molded Proxy* are defined on the fly.
15.7 Extended Participants

- MoldedProxy

  - At instantiation is either devoid of interface or all methods defined simply
    pass the request to the RealSubject (see implementation)
  
  - Interface and/or behavior is defined by the Client as needed
  
  - A single function or class might be used as a Macro for MoldedProxy
    creation and configuration

15.8 Collaborations

The Client instantiates the MoldedProxy and configures its interface and/or be-
behavior. After configuration, the MoldedProxy behaves like a traditional Proxy.

15.9 Consequences

The Molded Proxy extension has two significant consequences.

1. MoldedProxy instances can be configured to meet specific needs. Clients can
determine proxying needs and modify their instance accordingly.

2. Clients that create the MoldedProxy must understand the proxying process. This
possibly breaks encapsulation because Clients may need to understand detailed
implementation issues of both the RealSubject and the needed MoldedProxy.

15.10 Implementation

In the implementation of the MoldedProxy, there are two major decisions.

1. How are the dynamic methods to be attached to the MoldedProxy? In Python,
   methods can be dynamically assigned, so this is very simple. In C++ (using the
   FC++ library), it in necessary to pass functions into the MoldedProxy through
   a configuration method. These pseudo-methods will not be accessable directly
   and must be called through a (probably) klunky interface.

144
2. *How are the dynamic methods to be created?* These methods can be created through dynamic definition (including lambdas), currying, and composition.

### 15.11 Sample Code

See the sample code for **Flyweight - Dynamic Pseudo Instance**.

### 15.12 Related Patterns

The **Flyweight - Dynamic Pseudo Instance** uses a Molded Proxy to create it’s pseudo instances.

The **Iterator - Dynamic Interface** pattern extension is very similar in construction and usage. However, for the iterator, the goal is not proxying, but dynamically creating an extended interface for targeted clients whereas a proxy serves as a placeholder.
16.1 Base Pattern - Extension Name

Chain of Responsibility - Relational Chain

16.2 Motivation

It is possible to restructure a Chain of Responsibility into a relational design. The underlying premise of a Chain of Responsibility is searching and searches are what relational components are designed for.

In the relational design, a request is passed to a relational store that holds all possible handlers. The request can be passed alone, or with descriptive information if supported. The relational store searches for handlers that are registered to handle that type of request. The implementation can support applying the request to multiple handlers.

The main advantage of this solution is maintainability. A Relational Chain is a clearer solution than the traditional Chain of Responsibility. In addition to maintainability, the Relational Chain may be more efficient because handlers who do not intend to handle never become active (reduced context switching) and complex rules can be developed to further refine searches.

16.3 Forces

- Logic programming is ideal for searches

16.4 Applicability

Use the Relational Chain extension
Whenever imperative optimized tasks (ordering, etc.) are not extensive in the design

Or, if the design lends itself to complex handling rules.

16.5 **OO++ Interactions**

Each Handler registers itself with the RelationalStore along with the requests it wishes to handle. The requests are Command objects and when a request is issued, it is passed to the relational store that subsequently returns the appropriate handlers for handling. Client classes can also define additional (and more complex) rules for request handling.

This pattern extension requires the following language features:

- Relational rules and facts

16.6 **Structure**

Figure 16.1 captures the structure of this extension in UML.
16.7 Extended Participants

- RelationalStore
  - Maps request types to their handlers
  - Can be extended with additional rules

16.8 Collaborations

Handler instances map themselves into the RelationalStore before requests are made. Client instances can make requests that are processed at the RelationalStore and handled based upon the results.

16.9 Consequences

The Relational Chain extension is generally easy to implement, understand and extend. It does have one significant liability. Imperative constraints are difficult. Constraints that are not easily expressed in a search (e.g., ordering) are possible, but difficult to implement and maintain. If a design solution calls for significant constraints that are difficult to express in a search, a traditional Chain of Responsibility (or some hybrid) may be the better solution.

16.10 Implementation

In implementing the Relational Chain extension, there are three major issues.

1. When and how is a handler registered? Registration can be done at instantiation or later on in the code, but instantiation is preferable if possible.

2. Multiple handling? If multiple handlers will be allowed to handle the request, this must be configured in the call to RelationalStore.

3. Additional Rules? Client instances that wish to create additional search rules must have access to the RelationalStore.
16.11 Sample Code

In the code below, we have implemented this relational Chain to search for handlers in two specific ways. First, if a starting handler is specified, it proceeds from that handler to its successor in a manner similar to the original Chain of Responsibility pattern. If a starting point is not specified, it begins at a non-deterministic location and proceeds in a non-deterministic fashion to allow all appropriate handlers a chance to handle it.

class Handler:
    store = logic.relation()
    def __init__(self, successor, *listenForCommands):
        for commandClass in listenForCommands:
            Handler.store.assert(self, successor, commandClass)

def createHandlerRule(self, rule):
    Handler.store.rule += rule

def handle(self, Request):
    #Code for handling a request here

def emit(request, startingHandler = None):
    Handler = LogicVar()
    Successor = LogicVar()
    Command = LogicVar()
    if startingHandler == None:
        curHandler = startingHandler
        while Handler.store.query(curHandler, Successor, Command):
            if Command.value() == request.command:
                curHandler.handle(request)
                curHandler = Successor.value()
            else:
                while Handler.store.query(Handler, Successor, request.command):
                    realHandler = Handler.value()
                    realHandler.handle(request)
    emit = staticmethod(emit)
16.12 Related Patterns

The Observer - Descriptive Registry is similar to this extension in that it uses relational rules to process communication. However, the Relational Chain is focused on multiple handling of requests while the Descriptive Registry is designed for managing general communication between two or more objects.
17.0 COMMAND EXTENSIONS

17.1 Base Pattern - Extension Name

Command - Command Function

17.2 Motivation

The entire Command pattern can be reduced to passing and storing a function to the Invoker. The Command class primarily encapsulates a function and serves for typing. If typing the function is not needed, than in an OO++ context, the entire Command class hierarchy is not needed. Instead of creating a Command instance that calls some method of a Receiver, the method is passed directly to the Invoker.

This scenario was first described by [22].

17.3 Forces

• Simpler solutions make code easier to maintain and understand

17.4 Applicability

Use the Command Function extension

• When type-checking does not necessitate the Command class hierarchy.

• When the Command class does not need to store state (for undoable operations, etc).

17.5 OO++ Interactions

Any number of objects can pass a method or a function as a CommandFunction to an Invoker. The Invoker calls the stored function in place of Command.execute().

This pattern extension requires the following language features:

• First class functions
17.6 Structure

Figure 17.1 captures the structure of this extension in UML.

17.7 Extended Participants

- **CommandFunction**
  - Can be from any function or method that supports the correct number of arguments

- **Invoker**
  - Accepts a function (or method) in place of a Command class.

17.8 Collaborations

A **Client** passes a method of a receiver to the Invoker. The Invoker calls the method when necessary.

17.9 Consequences

The **Command Function** extension has four significant consequences.
1. *It eliminates the Command class hierarchy.* This significantly simplifies the design solution.

2. *Macro commands are still supported.* Using composition and/or dynamic definition, macro commands are easily created.

3. *Command functions are generally not type-checkable.* In an introspective language the CommandFunction can be checked for argument count, but that is the extent of type-checking a function.

4. *Command state is unavailable.* Without an encapsulating class, command state necessary for undo operations is unavailable.

### 17.10 Implementation

Implementing the Command Function extension is quite easy. However, it is worth noting that the Command pattern could be implemented as a cross between the traditional Command pattern and the Command Function extension.

We can create a Command class hierarchy that accept a method as a parameter instead of a receiver. If the Command class supports an `undo()` operation, than it can be passed in as well. The advantage of this extension is that we can still have type-checking and state within the Command class while still being able to turn any function into a command.

### 17.11 Sample Code

In our sample code we illustrate a hybrid object-oriented/functional Command implementation. Note that HybridCommand can be subclassed to enable type-checking.

```python
class HybridCommand(object):
    def __init__(self, command, undoCommand=None):
        self.__command = command
        self.__undoCommand = undoCommand
```
def execute(self):
    self.__command()

def undo(self):
    if self.__undoCommand is None:
        raise Exception("Can't undo this command")
    else: self.__undoCommand()

pasteCommand = HybridCommand(app.paste, app.undoPaste)

17.12 Related Patterns

Memento - Restoration Method can be seen a a degenerate form of this extension. The restoration method can be seen a a command of a single type and purpose.
18.1 Base Pattern - Extension Name

Interpreter - Descriptive Definition

18.2 Motivation

The grammar for a simple language can easily be defined using logic programming constructs. We can define an interpreter very easily using such constructs in an OO++ language.

18.3 Forces

- Logic programming constructs map almost perfectly to grammar definitions

18.4 Applicability

Use the Descriptive Definition extension

- When interpretation is the primary function (AST’s provide other functionality like replace)
- Grammar’s have short look-ahead sets

18.5 OO++ Interactions

This pattern is primarily logic programming based, but requires some imperative style constructs to control flow. These statements handle string processing, etc.

This pattern extension requires the following language features:

- Logic programming constructs for rules and facts
- The logical rules can have hooks to imperative statements
18.6 Structure

The structure of the original pattern shown in [2] remains the same. The new pattern extension simply alters the construction process.

18.7 Extended Participants

- Interpreter
  - Is a collection of logic programming rules and facts
  - Can interpret strings

18.8 Collaborations

Expressions are passed to the Interpreter for processing.

18.9 Consequences

The Descriptive Definition extension has 2 significant consequences.

1. *Functionality beyond interpretation is limited.* The nature of the logic programming rules and facts makes interpretation easy. Implementing additional functionality, on the other hand, may be challenging. If the cost of imperative hooks is prohibitive, than a non-logic solution is preferable.

2. *The Grammar is easily changed and/or extended.* To alter the grammar, just rewrite the logic rules.

18.10 Implementation

There are 2 major issues.

1. *Language Support* This pattern extension broke the prototype logic module used to implement the other logic programming style constructs in other patterns. The current implementation of the module could not handle the recursion issues. Logic programming in Leda is more robust and should be capable of
implementation of this pattern extension. Additionally, Leda’s implementation of logic programming has built-in state restoration making backtracking with string processing easy.

2. **Implementing the imperative components.** Both Python and Leda support easy integration of imperative statements into logic programming rules. These functions must be crafted to handle string processing and possibly look-ahead states.

### 18.11 Sample Code

While the current Python logic module broke implementing this extension, the desired code would look something like what is shown below. This is the basic implementation of interpreter with logic syntax; however, to function properly, the interpreter must be defined with some form of look-ahead to prevent infinite recursion. The other significant problem is that “eating” a symbol is not restored during backtracking in the current Python logic module.

```python
eat.rule = lambda eatee, symbol:
    doFunctionOnce(eatFunction, eatee, symbol)
expression.rule = lambda exp:
    literal(exp) | alternation(exp) | sequence(exp) |
    repetition(exp) | eat(exp,'(') & expression(exp) & eat(exp,')')
ealternation.rule = lambda exp:
    expression(exp) & eat('|') & expression(exp)
sequence.rule = lambda exp:
    expression(exp) & eat(exp, '&') & expression(exp)
literal.rule = lambda exp:
    doFunction(isLiteral, exp)
```

### 18.12 Related Patterns

The **Composite - Descriptive Instantiation** extension is very similar to this extension to **Interpreter**. However, an **Interpreter** interprets an expression while a **Composite** processes or creates a complex object.
19.1 Base Pattern - Extension Name

Iterator - Non-Deterministic Iterator

19.2 Motivation

Iteration over the elements of aggregated data is sometimes order independent. In circumstances where the amount of aggregated data is very large, some traversal policies serve as filters. These iterators are not necessarily concerned about a specific order of iteration because the goal is to guarantee that all of the data elements that meet certain constraints are visited. We term this form of iteration constraint-based iteration.

Logical relations are well-suited for this form of data traversal. This is because a relation can be viewed as a constraint-based iterator. The relation is created by describing rules (or constraints) and the relational query produces all data elements that satisfy the constraints.

Another advantage of using a relation as an iterator is that it is easily extended through new relations. A client can easily create new traversal policies simply by incorporating the original traversal relation into new rules. This form of extension requires neither an understanding of the aggregated data structure nor basic iteration.

We note that the basic iterator of the aggregated data does not have to be a logical relation. If the language supports wrapping non-logical elements into a logical relation, then the basic iterator can be a traditional object-oriented class instance. This iterator is then wrapped into a logical relation and the aggregated data can then be traversed in both non-deterministic and traditional fashions.
19.3 Forces

- Large amounts of data are difficult to filter and manage.
- Queries on logical relations are similar to iteration.
- Clients want to be able to iterate without understanding the process of iteration.
- The object-oriented approach to iteration requires an understanding the process of iteration.

19.4 Applicability

Use a Non-Deterministic Iterator pattern

- to provide traversal that is not directed.
- to support traversal of a describable subset of data elements
- to allow the Client to extend the traversal without understanding the aggregated data structure or basic iteration.

19.5 OO++ Interactions

Conceptually, the logical relation can be viewed as a non-deterministic iterator. It should be noted that wrapping a traditional iterator into a logical relation is a functional concept.

This pattern extension requires the following language features:

- Logical relations
- Wrapping traditional iterators into a relation [optional]

19.6 Structure

Figure 19.1 captures the structure of this extension in UML.
19.7 Extended Participants

- **WrappedIterator** (relation)
  
  - Wraps the **ConcreteIterator** into a logical relation.
  
  - Iterates across the same elements as the **ConcreteIterator**.

- **NonDeterministicIterator** (relation)
  
  - Combines the **WrappedIterator** relation with additional rules.
  
  - Iterates across all data elements that meet the defined constraints.

19.8 Collaborations

- A **ConcreteIterator** supports traditional directed traversal over aggregated data.

- Querying **ConcreteAggregateElements** relation returns the same elements as the **ConcreteIterator**.
• Querying **ConstrainedElements** returns all elements that match the relation’s constraints.

### 19.9 Consequences

The **Non-Deterministic Iterator** has the following strengths and limitations:

1. *It allows iteration to be defined by constraints.* In a logical relation, defining search constraints is a straightforward process. Conceptually, the relation describes the subset of aggregated data to be iterated.

2. *It abstracts away the structure of the aggregated data from the Client.* The WrappedIterator relation only guarantees traversal of all aggregated elements without suggesting how the traversal will occur and without implying what the underlying structure is. The Client can write NonDeterministicIterator relations without having to learn anything about the aggregate.

3. *Changes to the aggregated data structure have minimal impact on dependent logical relations.* As long as WrappedIterator returns every element of the aggregate, no dependent relations will be adversely impacted by changes to the aggregated data structure.

4. *Non-deterministic iterators cannot be used for directed iteration.* Non-deterministic iterators do not control the order of traversal. On the contrary, they operate under the assumption that traversal order is irrelevant and possibly dynamic. If, however, the ConcreteAggregate supports both a ConcreteIterator and WrappedIterator, then both methods of traversal are available to the Client.
19.10 Implementation

The primary issue in implementing the Non-Deterministic Iterator is creating an implementation of WrappedIterator. In the most simple case, the createIterator() method of ConcreteAggregate returns a logical relation and there is no need for WrappedIterator at all. NonDeterministicIterator can be extended directly from ConcreteIterator.

If, however, ConcreteIterator is a traditional iterator, then it must be wrapped into WrappedIterator. This wrapping process is language specific. In the sample code that follows, we use a function provided by our Python Logic module that converts a function and its associated arguments into a logical relation.

In the sample code below, we show the implementation of a WrappedIterator and a NonDeterministicIterator.

For the WrappedIterator, we chose to implement a subclass of relation rather than simply create a new relationship. We did this because it allows us to instantiate a predefined WrappedIterator for any existing ConcreteIterator.

19.11 Sample Code

class WrappedIterator(Relation):
    def __init__(self, iterator):
        Relation.__init__(self)
        self.__iterator = iterator
        self.__iterator.first()
        self.inference = lambda Element: (
            Relation.doFunction(lambda: not iterator.isDone(),()) &
            EQ(Element, Relation.doFunction(iterator.currentElement,())) &
            Relation.doFunctionOnce(iterator.next,()))

    def reset(self):
        Relation.reset(self)
        self.__iterator.first()
The client uses the WrappedIterator like this:

```python
i = ConcreteIterator(concreteAggregate)
iw = WrappedIterator(i)
    element = LogicVar()
while iw.query(element):
    # Do stuff with element
```

Instances of WrappedIterator can be included in other logical definitions. In the example below, assume that customerList stores thousands of customers and their commercial data. The client wants to iterate over the customers who should be sent mass mailings about student-debt consolidation through home equity. This can be done easily.

```python
I = CustomerIterator(customerList)
iw = WrappedIterator(i)
TargetedIterator = Relation()

TargetedIterator.inference = lambda Element: (
    iw(Element)
    HasHome(Element) &
    HasGoodCredit(Element) &
    HasStudentLoans(Element))
```

Of course, the relations HasHome, HasGoodCredit, and HasStudentLoans must be defined. But the important aspect of this is that changes in their definition have little impact on the validity of TargetedIterator. HasGoodCredit, for example, could initially check a raw credit score against some number. Later, a more complicated logical relation could be implemented that determines good credit based partially on factors such as the customer’s business relationship with the commercial entity, income, family status and college education. While improving HasGoodCredit increases the accuracy of TargetedIterator, it does not affect the interface between the two.
19.12 Related Patterns

*WrappedIterator* is a specialized form of the *Adaptor* pattern.

19.13 Base Pattern - Extension Name

*Iterator - Dynamic Interface*

19.14 Motivation

Iterators provide an abstract interface for traversing aggregated data. The traditional interface supports methods for resetting the iterator to the first element, advancing to the next element, checking to see if iteration is complete, and a method for returning the current element. If a client requires more specialized methods than *ConcreteIterator* already supports, a new subclasses must be created.

However, in circumstances where new traversal methods are required for a limited time or within a limited scope, extra subclasses are very expensive. For example, if a client requires additional traversal methods within a method or a single class only, then the subclass is highly coupled with that class and may exist solely to provide a single function. If the extended traversal method does not require privileged access to the *ConcreteIterator* or *ConcreteAggregate*, a better solution is for the client class to implement a (possibly private) method that performs the additional operation. However, this solution may lead to an awkward and complex interface. Also, because the client is implementing methods to serve another class, it may be difficult to keep the intent of the methods clear and their definitions cohesive.

Dynamic methods stored as data within the client class provide a solution to these problems. In this case, dynamic functions are created through currying, composition or other higher-order functions to create the extended interface. These constructs clarify intent and improve cohesiveness by enforcing a functional view of the extended methods. Creating an *Iterator* interface that is parameterized facilitates this process.
For example, an aggregated structure may contain a large number of bank accounts and associated data. Using traditional iterators, a client interested in traversal of overdrawn bank accounts must use a subclass of the basic iterator or implement internal methods that iterate over the desired elements. But conceptually, the client is now providing methods that operate on iterators, which can be confusing and non-cohesive.

On the other hand, if the basic iterator provides methods that accept an optional filter function, specialized filtered traversal methods are no longer required. If a specialized method is desired for readability or cohesion, currying provides an effective method for dynamically creating the function. This function can be stored as a data variable so that it is not confused as a method that operates on the client.

19.15 Forces

• Clients need extendable iterators.

• An individual client may require an iteration interface that is unique to that client.

• Subclasses may introduce unacceptably high overhead.

• Methods defined for the client may be confused as methods that operate on the client.

• Functions can be manipulated independent of the client or the iterator.

19.16 Applicability

Use Dynamic Interface when

• static changes to interface are too expensive.

• specialized methods or operations will only be used in a limited scope.
19.17 OO++ Interactions

Dynamic functions allow a client to manipulate the interface of an iterator to its own needs without introducing global constructs. These functions operate on the iterator but are stored as data within the client class. If the iterator interface allows it, functions can also be passed as policy-modifiers to the iterator.

This pattern extension requires the following language features:

- First-Class Functions
- Higher-Order Functions

19.18 Structure

Figure 19.2 captures the structure of this extension in UML.

19.19 Extended Participants

- ConcreteIterator
  - Modified to accept policy modification functions.
- **DynamicInterface** (Multiple Functions)
  
  - Created using higher-order functions.
  - Are data of the **Client**.
  - Operate on **ConcreteIterator**.

19.20 Collaborations

Client creates **DynamicInterface** from the **ConcreteIterator**.

19.21 Consequences

**Dynamic Interface** has 2 primary benefits:

1. *It limits extended interfaces to a specific Client*. Because the functions belong to the Client as data, no other actors within the system know about them. This keeps the **ConcreteIterator** interface and class hierarchy simple.

2. *It prevents confusion about method purpose*. Conceptually, a method operates on a class; Client methods that operate on an Iterator may cause confusion. Dynamic methods that serve as data for the Client reduce this confusion.

19.22 Implementation

There are a number of factors that affect implementation of a Dynamic Interface.

1. *Does the Iterator support policy modifying function arguments?* If it does, currying and composition are easy to use. If it does not, composite functions and anonymous functions are the best options for a dynamic interface.

2. *What other functions will be combined with the iterator’s methods?* For example, in currying the **next()** method with a policy modification function, it is possible that a function already exists that can serve as a policy modifier. In
the case of the overdrawn bank account mentioned previously, the bank account class probably implements an `isOverdrawn()` method.

3. What higher-order function will be used? Currying is an excellent tool for combining methods with policy modification functions. Composition can combine a method with a parameterized policy modification function. Anonymous functions serve as more general-purpose dynamic functions but are less cohesive than curried or composite functions.

### 19.23 Sample Code

In the sample code below, we generate composite and curried functions from a parameterized iterator interface.

```python
i = BankAccountIterator(bankAccountList)

firstOverdrawnAccount = curry(i.first, BankAccount.isOverdrawn)
nextOverdrawnAccount = curry(i.next, BankAccount.isOverdrawn)

firstAccountGreaterThan = compose(i.first,
    lambda min: lambda a: a.balance() > min)
nextAccountGreaterThan = compose(i.next,
    lambda min: lambda a: a.balance() > min)

firstOverdrawnAccount() while not i.isDone():
    print i.currentElement()
    i.nextOverdrawnAccount()

firstAccountGreaterThan(100) while not i.isDone():
    print i.currentElement()
    i.nextAccountGreaterThan(100)
```

### 19.24 Related Patterns

The composite functions in **Dynamic Interface** are similar to **Composite**.

The composite functions in **Dynamic Interface** are functional versions of **Decorator**.
20.1 **OO++ Mediator Discussion**

The traditional **Mediator** pattern serves as a communication hub for various objects. The definition of the communication is “hard wired” within the **Mediator** class, which can be difficult to maintain and/or understand. Even still, the increased complication may be worth it for the trade off of decoupled classes.

In considering how **OO++** could extend or refine the **Mediator** pattern, logic-programming is an immediate candidate for improving the connections between communicating objects. But on closer inspection, we find that using a relational store to connect objects in the **Mediator** pattern’s context provides little additional functionality than storage. If the **Mediator** pattern’s communication system is extended to be more powerful, supporting various kinds of communication connections, rules become a more viable method of defining communication relationships. However, at this point, the system is general enough it might function as a **Mediator**, as an **Observer**, or as an even more general communication hub.

In the end, the **OO++** programming cannot extend the **Mediator** pattern without having it become something more than a **Mediator**. To see how **OO++** programming might be used to implement the **Mediator**, look at the **Observer** redesign, where an **OO++** solution is implemented that can be used as a **Mediator** or an **Observer**.
21.1 Base Pattern - Extension Name

Memento - Restoration Method

21.2 Motivation

The Memento pattern provides a convenient design solution for restoring state to an object at a later point in time. Functional programming techniques can refine this pattern by simplifying the class structure and the Memento interfaces.

In many OO++ languages, methods (like functions) can be passed as data. Interestingly, the method can be permanently bound to an instance (or the receiver). Additionally, methods (and not just functions) can be created dynamically. This means that an instance can dynamically create methods bound to themselves and pass that method to someone else. When the function is executed, it triggers a request to the object that dynamically created it.

This hybrid object-oriented/functional combination can be used to create a functional Memento, or a Restoration Method. When state must be saved, an object saves it state by dynamically creating a method that will restore its state when called and passes that method (bound to itself) to the Caretaker. When state should be restored, the Caretaker executes the function.

This system eliminates the issue of narrow and wide interfaces (the function is really a method which has access to all the private members of the Originator) and eliminates the Memento class hierarchy. The functions themselves are generally more lightweight than Memento classes, reducing space requirements.
21.3 Forces

• Bound methods, passed as function, bind a receiver and a request

• Functions require less overhead in code and memory than classes and objects\(^1\)

21.4 Applicability

Use the Restoration Function extension

• When Mementos do not require types and type-checking

21.5 OO++ Interactions

This pattern extension requires the following language features:

• Dynamic methods

• First class functions

21.6 Structure

Figure 21.1 captures the structure of this extension in UML.

\(^{1}\)Though this benefit is lost in languages like Python where functions are actually instances of a special Function class.
21.7 Extended Participants

- RestorationMethod
  
  - Created by the Originator
  
  - Can be called by Client classes

  - Is actually a bound method to the Originator and has access to Originator private members

21.8 Collaborations

The Caretaker calls for the Originator to save state. The Originator creates a function that will restore its state and passes it to the Caretaker. The Caretaker stores the function until it needs to restore state and then executes the function.

21.9 Consequences

The Restoration Method extension has 3 significant consequences.

1. *Simplifies the class hierarchy.* Restoration methods eliminated the need for Memento classes

2. *Strengthens encapsulation.* Without Memento classes, there is no need for “narrow interfaces”

3. *Loses the advantages of classes.* Without a Memento class, the saved state cannot provide additional methods for auxiliary functions like returning the Originator, getting a date stamp of when state was saved, and so forth. Additionally, the saved state cannot be type checked.

21.10 Implementation

When implementing Restoration Method extension, consider these 2 issues.
1. **Saving State.** How state is saved may be non-trivial. The *Originator* might already provide functions (public or private) that restore state. Some state might be storable, and other state might require computation. It will vastly depend on the nature of the structure of the *Originator*.

2. **Restoring State.** The *RestorationMethod* can be called by anyone (because it is a function and not subtypable class). If this kind open access is unacceptable, the access can be limited by designing the *RestorationMethod* with special hooks that guard access. For instance, the *RestorationMethod*, when executed, can make a call to a function that will raise an *Exception* unless a special flag has been set by the *Originator*.

### 21.11 Sample Code

```python
class Point:
    def __init__(self, X, Y):
        self.__location = (X,Y)

    def makeMemento(self):
        localLocationCopy = self.__location
        def DYN_undo():
            self.__location = localLocationCopy
        return DYN_undo

    def shift(self, deltaX, deltaY):
        self.__location = (X + deltaX, Y + deltaY)

    def getLocation(self):
        return self.__location

p1 = Point(3,4)
undoFunction = p1.makeMemento()
p1.shift(1,1)  # p1.getLocation() = 4,4
undoFunction() # p1.getLocation() = 3,4
```

21.12 Related Patterns

This extension dynamically creates and exports a function similar to the Builder - LazyBuilder extension. However, the Lazy Builder’s purpose is to construct a complex object while Restoration Method restores an object to a given state.
22.1 Base Pattern - Extension Name

Observer - Descriptive Registry

22.2 Motivation

Relational constructs define by nature how two or more items are related given certain facts and rules. The emphasis of the Observer pattern, as well as the Mediator pattern, is defining the relationships between objects in terms of communication. A general system of communication can be created using logic-programming style relationships that can be used to implement communication patterns like Observer and Mediator.

22.3 Forces

• Relational constructs can easily describe communication between objects

22.4 Applicability

Use the Descriptive Registry extension

• When you want to describe the connections between objects

• When the descriptions (which are rule based) should be extendable

22.5 OO++ Interactions

This pattern extension requires the following language features:

• Relational constructs for rules and facts

• Relational constructs can store object instances, classes, and functions
22.6 Structure

Figure 22.1 captures the structure of this extension in UML.

22.7 Extended Participants

- Hub
  - Supports a relational store
  - Supports registration in the relational store
  - Supports rule extensions to the relational store either to Client instances or to Subclasses

22.8 Collaborations

This pattern extension has similar collaborations to a traditional Observer solution. The major difference is that the rules for connection subjects and observers can possibly be extended by Client instances.
22.9 Consequences

The Descriptive Registry extension has two significant consequences.

1. Rules can be extended by Client instances. If Client instances are provided access to the communication rules, they can be modified (or even replaced). If the logical system is dynamic enough, some rules could be inserted at run-time. These rule extensions can be used to route traffic and group signals.

2. Multiple Impact. The nature of relational programming can result in multiple matches. If functions are tied to Subject/Observer pairs, than multiple update functions can be applied to the same signal. This can sometimes be useful when different clients want the same Subject/Observer pair to observe different things.

22.10 Implementation

The implementation is very simple. For general details, see the sample code below.

The complex issue not dealt with in the sample code is how rules can and should be extended. This issue requires design and testing, but can be very effective for describing traffic routes for communication of messages. The major problem is how to extend the rules. If replacement of rules is allowed, existing communication flows might be broken. If different behavior is needed for a specific Client, try writing this into the existing rules first. If total replacement is necessary, consider setting up a double-dispatch call. If replacement is not allowed, the Client must have some way of hooking his rule extensions into the relational store.

22.11 Sample Code

class Hub:
    def __init__(self):
        self.registry = logic.Relation()
class Mediator(Hub):
    def __init__(self):
        Hub.__init__(self)
        self.registry.assert(changed, someWidget, function)
        self.registry.assert(changed, otherWidget, function)
    def widgetChanged(self, widget):
        while self.registry.query(changed, widget, function):
            function()

class ChangeManager(Hub):
    def __init__(self):
        Hub.__init__(self)
    def register(self, subject, observer):
        self.registry.assert(observing, subject, observer, function)
    def notifyAll(self):
        while self.registry.query(observing, subject, observer, function):
            function(subject, observer)

class PostOffice(Hub):
    def registerCaller(self, caller, *provides):
        # Caller defines what services it provides
        # Listener registers to listen to
        # objects providing certain services

22.12 Related Patterns

The Descriptive Registry extension serves as a general communication post-office. It is designed to direct messages from one object to another based on routing primitives (relational facts) and more complex routing instructions (relational rules). Similar patterns extensions include Chain of Responsibility - Relational Chain (which is similar in design and structure but has a more narrow purpose) and Factory Method - Non-Deterministic Constructor (which uses relational data to determine the appropriate object to construct).
This pattern extension subsumes OO++ Mediator extensions.
23   STATE EXTENSIONS

23.1 Base Pattern - Extension Name

State - Relational Table

23.2 Motivation

The traditional State pattern allows an object to appear to change class as internal state changes. It does this by creating state objects that are used internally by the Context class to determine functionality. In [2], it emphasizes the benefits of using state classes over a table approach. It lists the three disadvantages of table driven behavior as poor efficiency, difficulty understanding the transitions, and difficulty adding actions to accompany the state transitions.

Observe that using classes to represent state is not perfect either. Specifically, it introduces many new classes (which can be difficult to maintain), tightly couples the State classes together (if they determine state transitions), and it is difficult to extend (one new function in the context class propagates interface and behavior extensions to all of the state classes).

A relational construct can be used to create a table driven approach that is explicit in its transitions and easy to attach actions. While its efficiency may still be worse than a (virtual) function call, it also reduces the number of classes and is easy to extend.

23.3 Forces

- Relational tables can clearly describe relationships between transitions and actions
23.4 Applicability

Use the Relational Table extension whenever the efficiency differences between the relational lookup and a virtual function call are not important.

23.5 OO++ Interactions

This pattern extension requires the following language features:

- Relational constructs of rules and facts
- Relational rules and facts can store functions (or methods)

23.6 Structure

Figure 23.1 captures the structure of this extension in UML.

23.7 Extended Participants

- StateRelationalTable
  - Stores relationships between states, operations and state transitions
  - Can be modified by subclasses

Figure 23.1: UML Structure of State - Relational Table
23.8 Collaborations

Each call to a method of the Context instance is routed through the StateRelationalTable to look for the appropriate, state-specific operation. After the operation is executed, the output of the operation is used to determine transition to the next state. This can be done by another table, or by the operation itself.

23.9 Consequences

The Relational Table extension has 2 significant consequences.

1. *It makes extensions to the Context class (including subclassing) easier.* The relational rules for state can be easily modified within the class (extensions) or in subclasses. The traditional State pattern requires modification to classes (possibly including interfaces). If, for example, the Context class is subclassed, and the subclass adds a single new operation, than all of the State classes must also be subclassed or decorated.

2. *It eliminates the State class hierarchy.* This reduces maintenance and possibly improves overall operation efficiency.

23.10 Implementation

The primary issue in implementing a Relational Table for state transition is in how state transitions are defined. They might be defined in the StateRelationalTable or they might be defined as part of the state-specific operation.

It should be noted that the use of the logic-style table for rules (instead of simple facts) is possible, but unlikely. While a non-logic table could be used, the syntax of the logic relation is convenient for the purposes of a transition table.

23.11 Sample Code

class TCPConnection:
    def __init__(self):

189
state = logic.Relation()
state.assert(TCPEstablished, self.establishedOpen, TCPEstablished)
state.assert(TCPEstablished, self.establishedClose, TCPClosed)
# etc.

def open(self):
    state.query(self.currentState, stateFunction, nextState)
    stateFunction()
    self.myState = nextState

23.12 Related Patterns

The Observer - DescriptiveHub is designed in a similar manner. The difference is that descriptiveHib acts as a communication center while Relational Table handles state transitions.
24  OO++ TEMPLATE METHOD EXTENSIONS

24.1 Base Pattern - Extension Name

Template Method - Dynamic Hooks

24.2 Motivation

The Template Method pattern is used extensively in object-oriented programming. It allows subclasses to change the behavior of an operation without changing its algorithm by overriding primitive operations. However, the template method cannot respond to dynamic changes that require new primitive operations to be defined on the fly.

A simple OO++ extension to the Template Method pattern allows the object instance to overwrite the functionality of primitive operations at run-time through a known interface. In this manner, multiple clients could contribute to the algorithm dynamically or a function might be defined to meet specific needs and passed in after creation.

24.3 Forces

- Not all operations for a template method may be known during the definition of a class.

24.4 Applicability

Use the Dynamic Hooks extension

- When subclasses cannot fully anticipate their primitive operations
- When subclasses may wish to change their primitive operations
24.5 OO++ Interactions

This pattern extension requires the following language features:

- First class functions

24.6 Structure

Figure 24.1 captures the structure of this extension in UML.

24.7 Extended Participants

- ConcreteClass

  - provides an interface for overwriting a function

24.8 Collaborations

A client can pass a function to a ConcreteClass instance to overwrite a specific method. After this configuration, the ConcreteClass instances is accessed normally.

24.9 Consequences

The Dynamic Hooks extension has 2 significant consequences.

1. Dynamic Template Implementation. The template can be implemented with functions defined on the fly based on the environment and/or parameter configuration.
2. *Hook Connections can be Lost.* In a subclass, all of the hook methods that are overwritten are tied together through the subclass. With Dynamic Hooks, replacement functions may not be tied together at all. This can generate confusing and difficult to maintain code.

24.10 Implementation

In using Dynamic Hooks, the major implementation issue is how will methods be overwritten? In a very dynamic language, methods might be overwritten by direct assignment. But in less dynamic languages and even in dynamic languages where private methods must be modified, this will not work. The better solution is to have functions (stored as data) called within the methods and provide a single method to replace these functions to Client classes.

24.11 Sample Code

```python
class ConcreteClass(object):
    def __defaultOperation(self):
        ## Default

    def __init__(self, primitiveOp=None):
        self.__primitiveOpMod = primitiveOp

    def __primitiveOp(self):
        if self.primitiveOpMod == None:
            self.__defaultOperation()
        else:
            self.__primitiveOpMod()
```

24.12 Related Patterns

The *Iterator - Dynamic Interface* is a narrow example of this extension. The functions that can be passed in as filter functions are the dynamic primitives and filtration is the pre-existing algorithm.
25.1 **OO++ Visitor Discussion**

Upon first examination of the Visitor pattern, it seems that logic programming might provide a solution to some of the problems of the Visitor pattern. Using logic programming to design the double-dispatch, it appears that you could simplify the interface of the Visitor and make it easier to add concrete nodes without propagating all the changes to various Visitor subclasses.

However on closer inspection, the logic programming solution has the very same weaknesses as the traditional Visitor pattern. First, even though the double-dispatch system is simplified, multiple visitor subclasses are still required. This is because methods need to be registered with the logic-style visitor and all of those methods have to be defined somewhere. Additionally, many times the Visitor is used to accumulate state. Unlike methods (which can accumulate state by sending messages to their bound receiver) functions cannot accumulate state without accessing global variables. Finally, adding a new concrete node still requires a new method be defined for that node type for every visitor type.

Functional programming also provides no aid to the weaknesses of the Visitor pattern. The nature of the Visitor pattern is very static invalidating the dynamic power of functional programming.

The Visitor pattern is very object-oriented in nature. It performs no searching and has no need for dynamic functions. The classes and instances that compose this pattern seemingly cannot be simplified or replaced. Despite the drawbacks of the Visitor pattern, it does not seem improvable by OO++ constructs. Perhaps it
would be more effective to evaluate redesigning the complex object structure to more easily support new operations.
In this thesis, I have investigated a subtle paradigm shift from strict object-orientation to multiparadigm in various popular contemporary languages. Because these languages are still centered on object-oriented theory and language structure, I have designated them as the “object-oriented plus plus” (OO++) subset of multiparadigm languages. Additionally, I have proposed that paradigm shifts in languages, like imperative to object-oriented and object-oriented to OO++, may fracture the interface between design methods and language implementation. When these shifts happen, existing design methods cannot fully utilize, and may impede the correct use of, evolved language features. I have termed this design-induced disability in using programming language “design dysphasia”.

A possible solution to design dysphasia is to adapt existing design methods to match evolved language elements. The effectiveness of these adapted methods is not measured entirely by the solutions they enable but rather by their ability to empower engineers to utilize and interface with the evolved features correctly. Practitioners of effectively evolved design methods will better understand the languages as well as the language features they employ.

As a concrete illustration of adapting existing design methods, I have chosen to extend the twenty-three software design patterns proposed by Gamma, Helm, Johnson and Vlissides to OO++ variants. The GOF patterns are a well-known object-oriented design tool used in both academic and industrial circles. I have first proposed a design pattern maintenance cycle and then applied this cycle to the design patterns. I have provided a detailed analysis of all twenty-three patterns and successfully extended
seventeen. For the remaining five patterns, I have provided a short analysis of the forces that prevented extension.

In the sections below, I present a brief analysis and conclusions for the major contributions of the thesis.

26.1 **OO++**

Within a multiparadigm environment, the cross-paradigm interaction can be partitioned into combination and hybridization. Combination is the easiest form of paradigm interaction because it allows the constructs of each paradigm to operate individually, communicating with each other through a similar or common interface. Hybridization is much more challenging because it requires a profound understanding of the fabric of the component paradigms (or a very serendipitous accident) to generate the new synthesis. It may also require extensions to the existing language syntax and semantics.

A cursory examination of OO++ languages reveals that very few hybrid constructs exist. The preponderance of OO++ combinations relative to the scant number of OO++ hybrids is partially due to the early age of the OO++ paradigm but may also be an indictment of the relatively scant research in multiparadigm languages and design. The difficulty in extending language syntax and semantics also inhibits the development of OO++ hybrids.

Throughout this thesis, I provide OO++ implementations in Python (with extensions). The bulk of the multiparadigm constructs within these implementations are combinations with a few hybrid constructs sprinkled throughout. However, Python is a very dynamic language with relatively extensible semantics and a great deal of hybridization is possible. In a number of the implementations within this thesis, I considered creating and using a hybrid construct, but chose not to for fear of clouding
the purpose of the illustration. Nevertheless, dynamic languages (like Python), are perfect for prototyping and testing hybrid OO++ constructs.

26.2 Design Dysphasia

It is impossible to know the full impact and extent of design dysphasia. The hypothesis of design dysphasia is based on observations drawn from human nature, literature, and the history of computing. The research presented in this thesis asserts the existence of design dysphasia and claims that its impact is negative but does not provide any quantification.

Interestingly, I am a data point for design dysphasia. I found that as I investigated the extending of object-oriented patterns to OO++ variants, I struggled not to think about solutions to problems in anything other than a strongly object-oriented/imperative fashion. It required significant work to think within an OO++ paradigm. I still feel that some of the OO++ solutions I propose are short of truly innovative and elegant design that lies just beyond my mental reach. I have been conditioned to design in an object-oriented fashion, and this has impacted my ability to effectively use the features of an OO++ language.

26.3 Design Patterns

I present a very thorough analysis of the GOF patterns in the appendices. The analysis describes the forces that impact each GOF pattern as well as its strengths and weaknesses. More importantly, the analysis also breaks the object-oriented structure of the GOF pattern down into structure, data, behavior, and interaction components. This breakdown provides insights into why the pattern works and the fundamental principles of the design pattern.

For example, a common construction in the GOF patterns is to define how the solution interacts through a parent class (in many cases, an abstract class or, in other words, an interface). But the behavior of the solution is captured in subclasses. This
The separation of interactions from behavior is a common object-oriented design concept and is a component in many of the patterns.

The breakdown and analysis of the GOF patterns was essential to adapting the patterns to OO++ languages. By splitting the patterns into data, structure, behavior, and interaction components, I was able to experiment with swapping a part of the pattern with an OO++ equivalent.

There is no question that the OO++ extensions to the GOF patterns are not mature design solutions. In other words, the extensions are not patterns in the traditional sense; rather, they are extensions to existing patterns. Pattern extensions combine prototypical OO++ explorations with stable, seasoned object-oriented solutions. Observe that OO++ languages are very similar in that they have mature object-oriented syntax and semantics with somewhat prototypical multiparadigm constructs. It is possible that the relationship between extended object-oriented design and extended object-oriented language is not coincidental.

The primary benefit of the OO++ pattern extensions are not the actual design solutions, though some of them seem effective. The true contribution of these extensions is the OO++ relationships they embody. These relationships allow practitioners to understand and manipulate the multiparadigm features of an OO++ language in concert enabling further experimentation and design maturation.

26.4 Guiding Principles and Meta Patterns

In extending the object-oriented patterns to OO++, I used a set of “guiding principles” for functional and logic design. This set of principles is defined in 3 As I applied these guiding principles to individual components of GOF patterns, I was able to better grasp the way in which the object-oriented components of the original design could interface with non-object-oriented features. These experiments not only produced OO++ pattern extensions, but they also allowed me to understand the
original paradigm at a deeper level and better understand these functional and logic programming principles.

As I evaluated various patterns against various principles, I began to see additional principles that also guide functional and logic thinking. Below, I present an enhanced set of “guiding principles” with my observations. New principles discovered in the course of this research and presented here for the first time are italicized.

• Logic Analysis and Design
  – Data Refinement
    * Fact Refinement - See “Rule Refinement”
    * Rule Refinement - Both fact and rule refinement define data from the logic perspective that relationships are the data. Logic data (relationships) can be integrated into object-oriented components when the solution requires facts and rules.
  – Query Refinement
    * Think Non-Deterministically - Algorithms are based on searching for a solution to constraints. This means that query operations might return zero, one, or many solutions. Object-oriented operations that represent queries (the next() operation for an iterator, for example) can be replaced with a logic construct for non-deterministic querying.
    * Think Descriptively - The twin sibling of “Think Non-Deterministically”, Query definitions are inherently descriptive. Some object-oriented operations can be replaced with logic components for a more descriptive solution.
  – Control Design - This principle is primarily associated with the non-
logical components of logic programming. At present, it appears to have very little impact in OO++ design.

• Functional Analysis and Design

  – General Design Approaches

    * Adapting Similar Problems - This principle is the functional perspective on code-reuse. In the functional perspective, designs modify existing functions to operate on new problems. In OO++, functions can be similarly modified.

    * Stepwise Refinement - Also a principle in imperative and object-oriented design, stepwise refinement is also known as “top-down design” where the problem is broken down in stages of finer and finer granularity. In functional programming, this is partially achieved by passing one function as an argument to another to assist in the breakdown of the problem.

  – Functional Data Design

    * Recursive Refinement - Functional programming lends itself to recursive design. Functions operate on themselves, continually reducing the problem until a final solution is reached.

    * Type Refinement - Because functional programming avoids explicit typing, the purpose of functions is often to mold the type into something that another function expects. In OO++, even with explicit typing, functions can be used for translating types.

    * Delayed (Lazy) Evaluation - Some operations or pieces of data do not require full evaluation until directly called for or manipulated.
Lazy Evaluation means that data (including functions) should not be fully evaluated, defined, or computed until directly requested. This principle can be used to decouple definition and evaluation in an OO++ solution.

- **Functional Design Perspectives**

  * Pipes and Filters - Functional data tends to be constant. Variables are generally defined once and cannot be modified afterwards. With this “side effect free” data perspective, designs often pass information from one stage to another, creating entire solutions from a chain of operations.

  * Functional Composition - New functions can be created by combining existing functions with one another. This can be applied directly to OO++ solutions.

  * Function Interchange - Similar to Functional Composition, function interchange achieves dynamic designs by swapping out one function with another in a component (generally another function). This can be applied directly to OO++ solutions.

During the extension of the GOF patterns, I was not surprised to see that the extension process itself produced repetitive patterns. In other words, pattern extensions similar in form and function could be applied to multiple GOF patterns. I have catalogued these patterns of pattern extension (or meta-patterns) below. It should be noted that they bear a strong resemblance to many of the “guiding principles” defined previously. It is not coincidental that the principles that guide pattern redesign should be strongly correlated to the patterns of redesign. In the descriptions of the meta-patterns below, I also identify related guiding principles.
• **Constraint-Based Operation** (Think Non-Deterministically) - This meta-pattern extends an object-oriented pattern with logic programming constructs to support constraint-based operations. The basic concept is to replace an explicitly defined operation with one that follows some form of constraint.

• **Descriptive Based Operation** (Think Descriptively) - This meta-pattern also extends an existing pattern with logic constructs. Although the implementation might be similar to patterns with “Constraint Based Operations,” the fundamental issue is one of interface. This (meta-) pattern is used when an interface needs to replace some operation (including parameter passing) with a descriptive variant.

• **Dynamic Interface** (Functional Composition, Function Interchange, Delayed Evaluation) - Sometimes an interface cannot be (easily) defined statically. Functional programming can be used to generate dynamic interfaces to solve this problem by creating the interface at run-time when additional information is known.

• **Lazy Operations** (Delayed Evaluation) - It may be advantageous to separate the definition of an operation and the execution of that operation. Using functions defined at run-time (a functional concept), we can define an operation and then store it for later execution.

• **Simplified Callbacks** (Function Interchange) - Some object-oriented solutions create an object to be passed as a parameter simply for wrapping a function inside the class. Functional programming allows the function to be passed directly to the objects that need to access it.
26.5 Future Work

There are a vast number of avenues open for further research into the topics touched on in this thesis. Out of necessity, this thesis investigated four topics jointly: the OO++ paradigm, design dysphasia, pattern maintenance, and GOF pattern extensions. These four topics are tightly coupled and required some joint investigation for the results obtained. However, each topic is independently researchable.

First, the OO++ paradigm requires further investigation on many fronts. I identified a number of languages that support multiple paradigms in addition to object-oriented. These languages all have a number of points in common in the way OO++ combinations are created. What is unknown is how much commonality they share in OO++ hybridization. The dynamic features of Python that enable some hybrid constructs are not available in C++. What hybrid features can they both support? Are there any hybrid features that C++ can support that Python cannot? Can we conceive of hybrid operations independent of an implementation?

Another OO++ issue is that of readability. With more than a decade of extensive use in academic and industrial circles, object-oriented programming has matured considerably. The software development community has catalogued hundreds of different ways that object-oriented programs should not be constructed as well as a few suggestions for how they should. That kind of “shake-down” does not exist for functional or logic programs, and it certainly does not exist for hybrid combinations in multiparadigm programming. The multiparadigm community is still learning what they can do. It will be a long time before enough evidence exists to determine what they should do.

The second topic that is open to considerable research is design dysphasia. In this thesis, I propose the existence of design dysphasia. Additional work is required to understand design dysphasia in greater depth and to quantify its impact. This research
might include cognitive science in an interdisciplinary approach to understanding how design knowledge affects an individual’s view and ability to learn concepts from different design approaches.

Pattern maintenance, the third topic, is also a novel concept presented in this thesis that is largely unexplored. One of the most challenging problems in developing better pattern maintenance is its rejection from the patterns community. The traditional approach within this community to “discovering” a pattern is that it must have occurred independently in three separate software projects before it is a “real” pattern. Because of this, the patterns community is hesitant to embrace pattern extensions. An avenue of research might be to attempt validation of pattern extensions by looking through publicly available (open source, etc.) “real world” designs for code that could be rewritten using one or more of the extensions. Such examples would help to mature the extension as well as demonstrate viability.

Finally, the GOF pattern extensions provide a number of research opportunities. Certainly there are more possible extensions that could be applied to the GOF patterns than have been captured in this thesis. Perhaps more importantly, there are more meta-patterns for extending object-oriented patterns that need to be discovered and refined. It should be noted that there are a number of “guiding principles” mentioned previously that were not used to create the current extensions. These include:

- Logic Analysis and Design: *Control Design*

- Functional Analysis and Design
  - General Design Approaches: *Stepwise Refinement*
  - Functional Data Design: *Recursive Refinement*
– Functional Design Perspectives: *Pipes and Filters*

These guiding principles may lead to additional pattern extensions as well as additional meta-patterns for pattern extension.
LIST OF REFERENCES


209


A PATTERN REDESIGN DOCUMENTS

A.1 Analysis of the GOF Abstract Factory Pattern

The purpose of the Abstract Factory Pattern is to “Provide an interface for creating families of related or dependent objects without specifying their concrete classes”

A.1.1 Metadata

• Forces

  – Interchangeable components are an important component in many software systems
  
  – Many times, interchangeable components are constructed from a family of subcomponents; each subcomponent is instantiated independently
  
  – The construction process is often limited to use subcomponents from a single family
  
  – And yet, these families must be interchangeable
  
  – These families may need to be extended to include additional subcomponents later [Unresolved]
  
  – These families may not have perfect subcomponent/interface isomorphism [Unresolved]

• Strengths

  – It isolates concrete classes
  
  – It makes exchanging product families easy
– It promotes consistency among products

• Weaknesses

– Supporting new kinds of products is difficult

A.1.2 Concrete Components

• Data

– ConcreteFactory instance

– ConcreteProduct instances

• Structure

– ConcreteFactory classes implement the AbstractFactory interface

– ConcreteProduct classes implement an AbstractProduct interface

– ConcreteFactory instances create ConcreteProduct instances

• Interaction

– AbstractFactory interface publicly provide createProduct() methods to the Client

– AbstractProduct interfaces publicly provide access methods for a given product to the Client

• Behavior

– ConcreteFactory classes define the correct products to be returned given a createProduct() method call

– ConcreteProduct classes define concrete implementations for a given product
A.1.3 Metadata/Concrete Component Mapping

- **(DATA) ConcreteFactory instance** – The Client must know explicitly which ConcreteFactory it needs. The Client instantiates the correct ConcreteFactory and then uses the ConcreteFactory to instantiate ConcreteProduct instances.

- **(DATA) ConcreteProduct instances** – Product instances are not instantiated directly by the Client. They are created by method calls to the ConcreteFactory. This forces the Client to use the correct products (that belong to the appropriate family) and also allows the product families to be interchangeable by changing the ConcreteFactory. Each ConcreteFactory must know the products it instantiates explicitly.

- **(STRUCTURE) ConcreteFactory classes implement the AbstractFactory interface** – This allows multiple ConcreteFactory instances to be interchangeable. But it also makes it challenging add new products because it requires a new method to be added to the AbstractFactory interface (and, subsequently, all descendant subclasses) to create the appropriate product.

- **(STRUCTURE) ConcreteProduct classes implement an AbstractProduct interface** – This allows products of different families to be interchangeable. It should be noted that the Client only accesses the AbstractProduct interface. In other words, if ConcreteProduct classes extend the interface, the Client will generally not have access to these.

- **(STRUCTURE) ConcreteFactory instances create ConcreteProduct instances** – Each ConcreteFactory is required to know the appropriate ConcreteProduct classes to instantiate.
(INTERACTION) AbstractFactory classes publicly provide createProduct() methods to the Client – The Client does not directly create instances of ConcreteProduct classes. Instead, it is aware of the AbstractProduct interface and knows that it can create an appropriate subclass through a ConcreteFactory instance. This promotes consistency among products.

(INTERACTION) AbstractProduct classes publicly provide an interface for a given product to the Client – Generally, the Client is intentionally designed without knowledge of ConcreteProduct classes. This isolates the Client from these concrete classes.

(BEHAVIOR) ConcreteFactory classes define the correct products to be returned given a createProduct() method call – Methods in ConcreteFactory classes have a very limited behavior: return products. The sole purpose of a ConcreteFactory is to be a structural container for related products.

(BEHAVIOR) ConcreteProduct classes define concrete implementations for a given product – As we observed previously, the ConcreteProduct classes generally should not extend their interfaces. This constrains the implementation to work within the existing AbstractProduct interface.

A.2 Abstract Factory Pattern Redesign

A.2.1 Component Modification Possibilities

• Modify createProduct() methods to be based on logical rules defined within the corresponding ConcreteFactory.

  – Description – Each createProduct() method is optionally powered by a logical relation defined within the ConcreteFactory. Each method has
access to all subclasses of a given AbstractProduct and qualitative information about these subclasses. The class-wide logical relation is used to determine which subclass is appropriate. ConcreteFactory classes can be further subclassed to narrow the rules. This converts a ConcreteFactory from being a container for related products to a virtual container for qualitatively related products.

- Guiding Principle – Rule Refinement
- Guiding Principle – Think Descriptively
- Benefit – Groups products together qualitatively
- Limitation – Default products must be available if the query fails
- Limitation – Queries can return multiple results that may be undesirable in this design solution

A.2.2 New Component Possibilities

- Single “Kit” class with dynamic functions

  - Description – This component replaces the AbstractFactory class and all ConcreteFactory classes. This single class defines all createProduct() methods on the fly (but within the class definition). These methods are created with all versions of the product as parameters. When the class is instantiated, it requires a type parameters that determines which product will be returned when the a createProduct() method is called.

  - Guiding Principle – Functional Composition

  - Benefit – Resolves the difficult extendibility issue.
A.3 Analysis of the GOF Builder Pattern

The purpose of the Builder Pattern is to “separate the construction of a complex object from its representation so that the same construction process can create different representations”

A.3.1 Metadata

- Forces

  - Constructing a complex object from different source types OR from a single source into multiple output types
  
  - In either case, the construction process is divided into parts that can vary and parts that are static
  
  - Clients generally require independent interaction with the static construction components

- Strengths

  - It lets you vary a product’s internal representation
  
  - It isolates code for construction and representation
  
  - It gives you finer control over the construction process

- Weaknesses

  - None noted

A.3.2 Concrete Components

- Data

  - director instance
  
  - ConcreteBuilder instances
- **Product instance**

- **Structure**
  - `ConcreteBuilder` classes implement the `Builder` interface
  - `Client` configures `director` instances with an appropriate `ConcreteBuilder` instance
  - `ConcreteBuilder` instance creates the `Product` instance

- **Interaction**
  - `Builder` interface publicly provides `buildPart` methods to the `director` class
  - `ConcreteBuilder` classes publicly provide `getResult()` methods to the `Client`

- **Behavior**
  - `ConcreteBuilder` classes implement concrete construction behavior for each `buildPart` method
  - `director` class defines static construction process using `Builder` interface methods

### A.3.3 Metadata/Concrete Component Mapping

- **(DATA) `director` instance** – The `Client` creates an instance of `director` to control the complex object creation.

- **(DATA) `ConcreteBuilder` instances** – The `Client` creates an instance from a `ConcreteBuilder` class. The `Client` must know explicitly which `ConcreteBuilder` is required.
• (DATA) **Product instance** – The Product instance is a complex object constructed by the ConcreteBuilder under the direction of the Director. The ConcreteBuilder maintains the Product instance as it is constructed and provides the Client access to it. Because the director never has to know about the actual Product instance, the Product does not have to be subclassed (and in fact, often cannot be).

• (STRUCTURE) **ConcreteBuilder classes implement the Builder interface** – This allows the various ConcreteBuilder classes interchangeable. The director class is dependent on the Builder interface and not the concrete subclasses.

• (STRUCTURE) **Client configures a director instance with an appropriate ConcreteBuilder instance** – This allows the Client to control what is built while the director instance controls how construction proceeds.

• (STRUCTURE) **ConcreteBuilder instance creates the Product instance** – This frees the director from having to know about or understand the Product instance.

• (INTERACTION) **Builder interface publicly provides buildPart methods to the director class** – The director controls the construction of the Product instance through the buildPart methods provided by the Builder interface. The buildPart methods provide stepwise construction of the complex object (the Product instance). The buildPart methods return nothing intentionally making the director blind to the nature of the Product instance.

• (INTERACTION) **ConcreteBuilder classes publicly provide getResult() methods to the Client** – The Client uses the director instance to guide the
construction of the Product using the ConcreteBuilder. When the director is finished, the ConcreteBuilder provides the constructed Product instance to the Client via this getResult() method. The getResult() method is not defined in the Builder interface and is, therefore, inaccessible to the director instance.

- **(BEHAVIOR)** ConcreteBuilder classes implement concrete construction behavior for each buildPart method – Each ConcreteBuilder class knows what type of complex object it should construct and how to handle each buildPart message. Each method stepwise refines complex object.

- **(BEHAVIOR)** director class defines static construction process using Builder interface methods – The director encapsulates a static construction algorithm using the interface methods of the Builder. The behavior of the director is hard-coded.

## A.4 Builder Pattern Redesign

### A.4.1 Component Modification Possibilities

- Modify director class to contain a build template that produces a function that can be executed later with a ConcreteBuilder.

  - **Description** – Instead of directly building, the director class now produces a function with build instructions. This function is configured by options and/or can be designed to react to the environment. Alternatively, the function can be representative of build instructions for a single Product instance. The function then can be executed with a ConcreteBuilder to produce the Product. The director now serves as a build template producing concrete construction functions.
– **Guiding Principle** – Delayed (Lazy) Evaluation

– **Guiding Principle** – Type Refinement

– **Benefit** – Instructions for building an instance of a complex object can be encapsulated into a single function

– **Benefit** – Instructions for building certain types of complex objects can be manipulated, altered, and modified before storage into a single function

### A.4.2 New Component Possibilities

- Combine `Director.construct()` and `ConcreteBuilder.getResult()` into a single function.

  – **Description** – A function can be defined that both constructs and returns the resulting complex object (the `Product`). This function will behave like a constructor for the Complex Object.

  – **Guiding Principle** – Function Composition

  – **Benefit** – Abstracts the creation of a Complex Object to a constructor-like interface

  – **Limitation** – Dynamic function definition is sometimes unreadable and/or difficult to maintain

### A.5 Analysis of the GOF Factory Method Pattern

The purpose of the **Factory Method** Pattern is to “Define an interface for creating an object, but let subclasses decide which class to instantiate. **Factory Method** lets a class defer instantiation to subclasses”

### A.5.1 Metadata

- Forces
– Sometimes only the abstract type of an object is known in the construction template
– Abstract classes cannot (and should not!) be instantiated

• Strengths

– Provides hooks for subclasses
– Connects parallel class hierarchies

• Weaknesses

– The Factory Method only returns an instance of the abstract class

A.5.2 Concrete Components

• Data

– ConcreteCreator instance
– ConcreteProduct instance

• Structure

– ConcreteCreator classes implement the Creator interface
– The factoryMethod() signature is defined in the Creator interface
– ConcreteProduct classes implement the Product interface
– The factoryMethod() creates the ConcreteProduct instance

• Interaction

– The Creator interface provides (possibly privately) the factoryMethod() method to itself and subclasses; alternatively, the factoryMethod() can be publicly provided to a Client
• Behavior

  – Each ConcreteCreator class defines which ConcreteProduct its
  factoryMethod() should create and return

A.5.3 Metadata/Concrete Component Mapping

• (DATA) ConcreteCreator instance – Provides a factoryMethod() either internally (to other methods) or externally (to client instances). In the latter case, the ConcreteCreator may or may not serve any other purpose.

• (DATA) ConcreteProduct instance – This object is created by the factoryMethod(). If the factoryMethod() is internal, this instance is passed to other (probably inherited) methods. If it is external, the instance is returned to a Client

• (STRUCTURE) ConcreteCreator classes implement the Creator interface
  – While internal methods or Client instances cannot know the exact class that will be instantiated, they rely on every ConcreteCreator class supporting the creation interface.

• (STRUCTURE) The factoryMethod() signature is defined in the Creator interface – This allows the Client or the internal methods an interface for creating an object without knowing the exact (sub)class of the object.

• (STRUCTURE) ConcreteProduct classes implement the Product interface
  – All the objects that can be created by a factoryMethod() must inherit from the same superclass. This is necessary in statically typed languages or in dynamically typed languages when the type of the product will be checked. If typing will not be enforced, this requirement can be dropped in the dynamically typed languages. However, in this latter case, even though the Product
instances need not be specifically tied together, they must generally support the same interface. In other words, they will in all likelihood implicitly implement the same interface.

- **(STRUCTURE)** *The factoryMethod() creates the ConcreteProduct instance* – This allows the creating entities to not know the ConcreteProduct class.

- **(INTERACTION)** *The Creator interface provides (possibly privately) the factoryMethod() method to itself and subclasses; alternatively, the factoryMethod() can be publicly provided to a Client* – The interaction depends on the nature of the solution. In private scenario is generally used when the Creator class provides a method that encapsulates an object creation template (e.g., the object is instantiated and configured). If this is the only purpose for the factoryMethod(), there is no need to make it public. On the other hand, if the factoryMethod() is needed for external client instances, then it must obviously be public. In either case, the only knowledge needed for creating the object is the signature of the factoryMethod().

- **(BEHAVIOR)** *Each ConcreteCreator class defines which ConcreteProduct its factoryMethod() should create and return* – This forces Client instances or internal methods to use the object the subclass designates but it also frees them from having to know exact types. This allows parallel hierarchies to be tied together, and template construction methods defined in the Creator class to work in ConcreteCreator classes.
A.6 Factory Method Pattern Redesign

A.6.1 Component Modification Possibilities

- Convert `factoryMethod()` into a non-deterministic variant.
  - Description – Instead of being hard coded to produce a single object, `factoryMethod()` is now search based (logic programming style) and configured by logical rules that describe the sort of object that should be created. `factoryMethod()` for a given class returns zero to many objects that meet that classes defined logical rules.
  - Guiding Principle – Think Non-Deterministically
  - Guiding Principle – Think Descriptively
  - Benefit – Objects can be created by description
  - Limitation – No guarantee of object creation

A.7 Analysis of the GOF Prototype Pattern Pattern

The purpose of the Prototype Pattern is to “Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype”

A.7.1 Metadata

- Forces
  - Some applications require a large number of types (classes) of objects
  - Defining a separate class for each type of object may be wasteful
  - Or which subclass needs to be instantiated in a class hierarchy may also be unknown
  - Often, instances of the objects already exist or are available

- Strengths
– May reduce the number of classes needed
– Reduces subclassing
– Allows adding and removing products at run-time
– Creation of new objects (types) by varying structure

• Weaknesses

– “The main liability of the Prototype pattern is that each subclass of Prototype must implement the \[\text{clone}\()\] operation, which may be difficult. For example, adding \[\text{clone}\()\] is difficult when the classes under consideration already exist. Implementing \[\text{clone}\()\] can be difficult, when their internals include objects that don’t support copying or have circular references.” [GOF]

A.7.2 Concrete Components

• Data

– Instances of \texttt{ConcretePrototype} classes.

• Structure

– \texttt{ConcretePrototype} classes implement the \texttt{Prototype} interface
– \texttt{ConcretePrototype} instances can reproduce themselves

• Interaction

– \texttt{Prototype} interface publicly provides the \texttt{clone()} method to Clients

• Behavior

– Each \texttt{ConcretePrototype} class defines the \texttt{clone()} operation; this operation applied to an instance returns a replica of that instance
A.7.3 Metadata/Concrete Component Mapping

• (DATA) Instances of ConcretePrototype classes – Rather than proliferating classes and/or subclasses, these instances (when serving as a prototypical instance) serve as the type definitions. Because the copies will be replicas of the prototypes, the line between data and data definitions are blurred.

• (STRUCTURE) ConcretePrototype classes implement the Prototype interface – This requirement is essential in statically typed languages. Under these circumstances, the clone() operation generally is designed to return an instance of Prototype and the exact subclass is unknown. In a dynamically typed language, the ConcretePrototype classes must provide a clone() operation and implicitly implement a Prototype interface.

• (STRUCTURE) ConcretePrototype instances can reproduce themselves – The clone() operation is a pseudo-constructor. Clients create other instances from the prototypical instance. This eliminates the need of Clients to explicitly know ConcretePrototype constructors.

• (INTERACTION) Prototype interface publicly provides the clone() method to Clients – This allows Clients to know that the objects they are working with are cloneable. This interaction replaces the need for additional classes (if the instances replace subclassing) or for factory methods (if the clone() method is to create other objects when the type is unknown).

• (BEHAVIOR) Each ConcretePrototype class defines the clone() operation; this operation applied to an instance returns a replica of that instance – Each class must know how to reproduce itself. The significant weakness of the
pattern is that this operation may be very difficult depending on the structure of the class.

A.8 Prototype Pattern Redesign Pattern Redesign

No effective redesign possibilities were found.

A.9 Analysis of the GOF Singleton Pattern Pattern

The purpose of the Singleton Pattern is to “ensure a class only has one instance, and provide a global point of access to it”

A.9.1 Metadata

• Forces

  – A class is required to encapsulate some functionality that is widely accessed among various clients
  – The class should only have a single point of access; in other words, there should only be one object (with state) that represents this system
  – The class may need to be subclassed

• Strengths

  – Controlled access to the sole instance
  – Reduced name space
  – Permits refinement of operations and representations
  – Easily extended to support a variable number of instances
  – More flexible than class operations

• Weaknesses

  – Subclassing is difficult
A.9.2 Concrete Components

- Data
  - The sole Singleton class instance

- Structure
  - The Singleton class creates and controls access to the sole Singleton instance

- Interaction
  - The Singleton class provides instance() to grant access to the sole instance

- Behavior
  - instance() controls creation and access to the sole instance

A.9.3 Metadata/Concrete Component Mapping

- (DATA) The sole Singleton class instance – This is that actual instance that will be accessed by clients. By limiting the number of instances to one, whatever functionality is encapsulated by the class is limited to a single state - that of the sole instance.

- (STRUCTURE) The Singleton class creates and controls access to the sole Singleton instance – The class itself defines functionality that limits the number of instances to one. Additionally, the class controls the creation of the instance; clients are denied the ability to construct it.

- (INTERACTION) The Singleton class provides instance() as a class method to grant access to the sole instance – clients do not know about the
construction of the instance or even when it happens. Access to it is always through the \texttt{instance()} method which controls instantiation. This is the global point of access.

- (BEHAVIOR) \texttt{instance()} controls creation and access to the sole instance
  - This method returns the sole instance but first creates it if it does not exist.

While this is advantageous to clients of \texttt{Singleton} as a base class (from the client’s perspective, the instance has always existed and is always available) it makes subclassing challenging.

A.10 Singleton Pattern Redesign

No effective redesign possibilities were found.

A.11 Analysis of the GOF Adapter (Object Adapter) Pattern

The purpose of the Adapter Pattern is to “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.”

\textbf{NOTE:} We ignore the Class Adapter pattern and focus only on the Object Adapter pattern.

A.11.1 Metadata

- Forces
  - Classes are often created by different entities, with different interfaces for different purposes
  - Two classes may conceptually be compatible, but not correctly interface
  - Classes designed for general (and multiple) purposes cannot possibly anticipate the needs of all classes that may wish to use them

- Strengths
• Lets a single Adapter work with many adaptees

• Weaknesses

  • Makes it harder to override Adaptee behavior
  • Cannot be used in place of Adaptee (because it does not inherit from adaptee)
  • Introduces an additional pointer and pointer indirection

A.11.2 Concrete Components

• Data

  • Adaptee instance
  • Adapter instance

• Structure

  • Adapter classes implement the Target interface
  • Client instances configure Adapter instances with an appropriate Adaptee instance

• Interaction

  • The Target interface publicly provides request() methods to the Client
  • The Adaptee class publicly provides specificRequest() methods to the ConcreteAdapter class

• Behavior

  • Each Adapter class wraps the behavior of the Adaptee’s specificRequest() methods to Client class’s needs within its request() methods
A.11.3 Metadata/Concrete Component Mapping

- **(DATA) Adaptee instance** – The Adaptee exists independently from the other components of this design pattern. Its instantiation and history are unknown (and unimportant) and it is invariant at this level of the design solution. In fact, the other participants are designed to wrap around this piece of data.

- **(DATA) Adapter instance** – Serves as an intermediate interface between the Adaptee instance and the Client. This instance is NOT of type Adaptee and cannot be used interchangeably.

- **(STRUCTURE) Adapter classes implement the Target interface** – This allows the Adapter instance to work with the Client but prevents it from being used in place of the Adaptee. In other words, this structure is like a degenerate Facade instance. This structure also prevents the Adaptee operations from being overwritten directly.

- **(STRUCTURE) Client instances configure Adapter instances with an appropriate Adaptee instance** – The Adapter knows nothing about the specific Adaptee instance it will wrap. This allows the Adapter to adapt all instances of Adaptee and subclasses. The Client is responsible for understanding how the Adaptee instance should be wrapped.

- **(INTERACTION) The Target interface publicly provides request() methods to the Client** – The Target interface is designed with explicit knowledge of the Client class’s needs. From this perspective, the Target class hierarchy is not designed to be a general, re-usable component. Its sole purpose is that of an interface.

- **(INTERACTION) The Adaptee class publicly provides specificRequest()**
methods to the ConcreteAdapter class — The specificRequest() methods are actually provided to Client classes, but within the context of this design solution, these Client classes do not know how to work with these methods. Instead, they are provided to the ConcreteAdapter and translated into methods the Client classes can understand.

• (BEHAVIOR) Each Adapter class wraps the behavior of the Adaptee’s specificRequest() methods to Client classes’ needs within its request() methods — Most of the real work being done is performed by the Adaptee instance. The purpose of these methods is to translate interfaces. The behavior of the Adapter’s methods should be minimal in terms of actual functionality.

A.12 Adapter (Object Adapter Pattern Redesign)

A.12.1 Component Modification Possibilities

• Adapter modifying Adaptee interface

  – Description — Modify Adapter to accept an Adaptee instance as input and return it with an extended interface of dynamically created functions (methods). Adapter includes an unadapt(Adaptee) method which removes the extended interface.

  – Guiding Principle — Adapting Similar Problems

  – Guiding Principle — Type Refinement

  – Benefit — Combines the benefits of Class Adapter and Object Adapter.

  – Limitation — Can increase difficulty of readability and maintainability.

  – Limitation — Requires dynamic language semantics.
A.13 Analysis of the GOF Bridge Pattern

The purpose of the Bridge Pattern is to “Decouple an abstraction from its implementation so that the two can vary independently”

A.13.1 Metadata

• Forces

  – Within a given software system are both abstractions and implementations

  – Within the same system, a given abstraction may be implemented multiple ways (generally through subclassing); during maintenance, extension, or third-party use, implementations proliferate

  – Subclassing allows variations in implementation but not in abstraction

  – Subclassing can introduce inter-class dependencies which proliferate the number of subclasses, reduces cohesion, and increases inappropriate coupling (between the subclasses)

• Strengths

  – The interface and implementation are decoupled

  – Both the interface and the implementation can be extended

  – Implementation details are hidden from clients

• Weaknesses

  – Unknown

A.13.2 Concrete Components

• Data
Abstraction instance and/or instances of RefinedAbstraction classes (Abstraction may or may not be an instantiateable); for simplicity, we refer to the concrete instance as being of type RefinedAbstraction.

Instances of ConcreteImplementation classes

- Structure
  - RefinedAbstraction classes implement the Abstraction interface
  - ConcreteImplementation classes implement the Implementor interface
  - RefinedAbstraction instances reference an object of type Implementor

- Interaction
  - Implementor publicly provides implementation methods to the Abstraction class
  - Abstraction publicly provides operation methods to Client classes

- Behavior
  - ConcreteImplementation classes define behavior for the implementation methods they inherit
  - RefinedAbstraction classes define operations using the implementation methods of the ConcreteImplementation object it references

A.13.3 Metadata/Concrete Component Mapping

- (DATA) Abstraction instance and/or instances of RefinedAbstraction classes (Abstraction may or may not be an instantiateable); for simplicity, we refer to the concrete instance as being of type RefinedAbstraction – Client classes interact with the RefinedAbstraction instance.
• (DATA) *Instances of ConcreteImplementation classes* – The Client does not use the implementation classes at all, but it does have to create them and pass them to the RefinedAbstraction instance.

• (STRUCTURE) *RefinedAbstraction classes implement the Abstraction interface* – This allows the abstraction to vary.

• (STRUCTURE) *ConcreteImplementation classes implement the Implementor interface* – This allows the implementation to vary.

• (STRUCTURE) *RefinedAbstraction instances reference an object of type Implementor* – The RefinedAbstraction instance requires the Implementor instance for configuration. Changing the Implementor instance changes the nature of the product produced by the RefinedAbstraction instance’s operation.

• (INTERACTION) *Implementor publicly provides implementation methods to the Abstraction class* – RefinedAbstraction instances perform their operations using the implementation interface defined by Implementor. Client classes do not need to know this interface (and, in fact, may be denied from accessing it). By denying Client instances interaction with the implementation, they are relieved from knowing about or understanding implementation details.

• (INTERACTION) *Abstraction publicly provides operation methods to Client classes* – Client classes understand the operations of the combined design solution only through the interfaces provided by Abstraction and its subclasses.

• (BEHAVIOR) *ConcreteImplementation classes define behavior for the implementation methods they inherit* – These classes do not have to worry about algorithmic details, and instead focus on defining the behavior of primitives.
• (BEHAVIOR) RefinedAbstraction classes define operations using the implementation methods of the ConcreteImplementation object it references – In this manner, the RefinedAbstraction defines algorithms without worrying about implementation details.

A.14 Bridge Pattern Redesign

A.14.1 New Component Possibilities

• Rule Based Variable Bridge Point (Floating Bridge)

  – Description – Rather than specifically tying an abstraction and an implementation together, create a logic-style junction between the two. Force implementation classes to describe themselves and use logical rules in the abstraction classes to determine which one or which ones would be best.

  – Guiding Principle – Think Descriptively

  – Guiding Principle – Think Non-Deterministically

  – Benefit – Clients no longer need to know which implementation is best; each RefinedAbstraction class (or even each Abstraction instance) determines that at run-time.

  – Benefit – Abstraction classes can be designed to operate on multiple implementations if appropriate.

  – Limitation – Junction rules may not return any implementation.

A.15 Analysis of the GOF Composite Pattern

   The purpose of the Composite Pattern is to “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly”
A.15.1 Metadata

- Forces
  - Some solutions tend (conceptually and practically) towards object composition
  - Often, these composite objects should support the same operations and behavior of their components

- Strengths
  - Simplifies client code
  - Facilitates addition of new types of components
  - Allows interchangeability between simple and composite objects

- Weaknesses
  - Can make the solution’s design overly general

A.15.2 Concrete Components

- Data
  - Leaf instance
  - Composite instance

- Structure
  - Both Leaf and Composite implement the Component interface
  - Composite classes maintain references to other (child) Component instances

- Interaction

239
- Component publicly provides operations common to both Leaf and Composite instances to the Client

- Either the Component class or the Composite class publicly provide child management operations to the Client

• Behavior

  - Leaf classes define the correct behavior for their operations

  - Composite classes define the correct behavior for their operations, which involves determining how to invoke the appropriate operations on child classes

  - If the Component class provides child management methods, the Leaf class must define behavior to deal with this (as an error case, generally)

  - The Composite class defines the correct behavior for child management

A.15.3 Metadata/Concrete Component Mapping

• (DATA) Leaf instance – The key to this pattern is that Leaf instances and Composite instances appear the same. They are both of type Component.

• (DATA) Composite instance – The weakness to this pattern is the same as the benefit: both leaf and composite instances (should) look the same. The problem is that Composite instances require additional functionality.

• (STRUCTURE) Both Leaf and Composite implement the Component interface – This structure allows both Leaf and Composite instances to be treated as the same type. However, the difficult question is where the Composite-only interface should be defined.
• (STRUCTURE) **Composite** classes maintain references to other (child) **Component** instances – This creates a recursive tree structure. Generally **Composite** classes achieve some or all of their functionality through operations on their children.

• (INTERACTION) **Component** publicly provides operations common to both **Leaf** and **Composite** instances to the **Client** – The interaction with **Composite** and **Leaf** structures for these operations should be the same.

• (INTERACTION) Either the **Component** class or the **Composite** class publicly provide child management operations to the **Client** – If the child management features (which are specific to the **Composite** class) are defined in the **Component** interface, the **Leaf** class will have to define behavior for those methods (which is generally errant behavior). On the other hand, if only the **Composite** class defines the operations, it must be downcast from a **Component** to use these features and transparency is not preserved.

• (BEHAVIOR) **Leaf** classes define the correct behavior for their operations – From a recursive perspective, these are the “base case” operations.

• (BEHAVIOR) **Composite** classes define the correct behavior for their operations, which involves determining how to invoke the appropriate operations on child classes – These operations call leaf operations (base case) and other composite operations which recurse.

• (BEHAVIOR) If the **Component** class provides child management methods, the **Leaf** class must define behavior to deal with this (as an error case, generally) – How the errant behavior will work depends on the context and the language features (most languages support throwing exceptions).
• (BEHAVIOR) *The Composite class defines the correct behavior for child management* – This behavior depends on the nature of the composite. In the simple case, this is simply adding or deleting a child from a list. Other circumstances may necessitate more complicated solutions (like ordering, etc.).

A.16 Composite Pattern Redesign

A.16.1 Component Modification Possibilities

• Method Composition

  – *Description* – Instead of composing whole objects, simply compose functions or methods

  – *Guiding Principle* – Function Composition

  – *Benefit* – Objects of any type may participate

  – *Limitation* – Decomposability is not preserved.

A.16.2 New Component Possibilities

• Descriptive Composite Creation

  – *Description* – Use logical rules to describe a composite structure to be instantiated

  – *Guiding Principle* – Think Descriptively

  – *Benefit* – Simplifies the creation of complex objects

A.17 Analysis of the GOF Decorator Pattern Pattern

The purpose of the *Decorator* Pattern is to “Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality”
A.17.1 Metadata

- Forces
  
  - Many solutions require a large number of objects of similar intent and purpose but with slightly different responsibilities
  
  - Using subclasses to add responsibilities is inefficient, especially if the responsibilities may need to be shared across several classes (subclass explosion)
  
  - Or subclasses are not flexible enough for run-time configuration
  
  - Also, some classes are hidden and not available for subclassing

- Strengths
  
  - More flexible than static inheritance
  
  - Avoids feature-laden classes high up in the hierarchy

- Weaknesses
  
  - Loss of object identity
  
  - Can result in a large number of general objects that are difficult to maintain

A.17.2 Concrete Components

- Data
  
  - Instances of ConcreteComponent classes
  
  - Instances of ConcreteDecorator classes

- Structure
  
  - ConcreteComponent classes implement the Component interface
- **Decorator** interface inherits the **Component** interface

- **ConcreteDecorator** classes implement the **Decorator** interface

- **ConcreteDecorator** instances each maintain a reference to a **Component** instance

- **Interaction**

  - **Component** publicly provides **operation()** methods to **Client** classes

  - **Decorator** provides the same **operation()** methods as **Component** (and must because it inherits from **Component**) and primarily serves as a subtype

  - **ConcreteDecorator** classes may provide additional operation methods to **Client** classes or privately to themselves

- **Behavior**

  - **ConcreteComponent** classes define the correct behavior for the operation methods defined in **Component**

  - **ConcreteDecorator** classes modify the behavior of the **ConcreteComponent** that it maintains a reference to

### A.17.3 Metadata/Concrete Component Mapping

- **(DATA)** **Instances of ConcreteComponent classes** – These represent the “real” data. These objects are extended by decorators with additional responsibilities.

- **(DATA)** **Instances of ConcreteDecorator classes** – These objects, though of the same type as **ConcreteComponent**, are not designed to function independently. They add responsibilities to **ConcreteComponent** instances. This
increases flexibility but incurs a penalty of additional numbers of objects and a loss of object identity.

- **(STRUCTURE)** *ConcreteComponent* classes implement the *Component* interface – *ConcreteComponent* and *Decorator* have the same parent.

- **(STRUCTURE)** *Decorator* interface inherits the *Component* interface – It is critical to the design solution that the *ConcreteComponent* classes and *Decorator* and its subclasses share the same parent class. The *Decorator* must be interchangeable with the *ConcreteComponent* it decorates.

- **(STRUCTURE)** *ConcreteDecorator* classes implement the *Decorator* interface – This partitions the *ConcreteDecorator* classes from the *ConcreteComponent* classes. The *Decorator* interface does not necessarily add any additional methods to the *Component* interface it inherits. (In other words, the *Decorator* class exists for establishing a separate subtype.)

- **(STRUCTURE)** *ConcreteDecorator* instances each maintain a reference to a *Component* instance – The *ConcreteDecorator* is a container for a single responsibility. It adds functionality to the *Component* it references. *ConcreteDecorator* instances are used in place of the *Component* they decorate.

- **(INTERACTION)** *Component* publicly provides *operation()* methods to *Client* classes – This is the interface that *Client* classes will use. The *Client* classes can interact with decorated *Component* instances identically to how they would interact with *ConcreteComponents*.

- **(INTERACTION)** *Decorator* provides the same operation methods as *Component* (and must because it inherits from *Component*) and primarily
serves as a sub-type – Decorator creates a new subtype to distinguish ConcreteDecorators from ConcreteComponents.

- **(INTERACTION)** *ConcreteDecorator* classes may provide additional operation methods to *Client* classes or privately to themselves – Generally, the *Client* classes work with the *ConcreteDecorators* as if they were *Component* instances and will not access additional operations. However, the added methods might be used by themselves in decorating the *Component* they reference.

- **(BEHAVIOR)** *ConcreteComponent* classes define the correct behavior for the operation methods defined in *Component* – Because the Decorators can add additional functionality, these operation methods can be designed cohesively and simply.

- **(BEHAVIOR)** *ConcreteDecorator* classes modify the behavior of the *ConcreteComponent* that it maintains a reference to – This modification is generally an extension (hence, decoration). In fact, the GOF pattern enforces this by having the component’s operation always called first.

A.18 Decorator Pattern Redesign

A.18.1 Component Modification Possibilities

- Functional Decorator
  
  - *Description* – Instead of defining other classes, for *Component* classes where decorators modify a single function, use functional composition to decorate the operation.
  
  - *Guiding Principle* – Functional Composition
  
  - *Benefit* – No loss of object identity
- *Benefit* – Reduction of *Decorator* subclasses and object instantiations.

- *Limitation* – Become impractical for use on classes that decorate more than one function.

### A.19 Analysis of the GOF Facade Pattern Pattern

The purpose of the *Facade* Pattern is to “Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use”

#### A.19.1 Metadata

- **Forces**
  - Object-oriented design and programming facilitate massive decomposition into smaller and smaller classes
  - Within any reasonably sized system, the number of classes can be very large
  - Third-party libraries increases the number of classes in use significantly
  - Some *Client* instances within the software solution may require a simplified interface to the extensive number of classes available

- **Strengths**
  - It shields *Client* classes from subsystem components
  - It promotes weak coupling between the subsystem and its clients
  - It doesn’t prevent applications from using subsystem classes if they need to

- **Weaknesses**
  - Unknown
A.19.2 Concrete Components

- Data
  - Facade instance

- Structure
  - Facade instance maintains references to subsystems

- Interaction
  - Facade publicly provides simplified interface methods to access critical features of the subsystems

- Behavior
  - Facade defines behavior for its methods that rely on the subsystems for the actual processing; this behavior could be emulated by the client (as the subsystems are generally public) but is encapsulated within the Facade class to promote high cohesion and loose coupling

A.19.3 Metadata/Concrete Component Mapping

- (DATA) Facade instance – Client classes interact with this data instead of multiple subsystems.

- (STRUCTURE) Facade instance maintains references to subsystems – This eliminates the need of Client classes to access these systems BUT it does not prevent it. Client classes can still access the subsystems if needed.

- (INTERACTION) Facade publicly provides simplified interface methods to access critical features of the subsystems – This greatly reduces the amount of knowledge Client classes must have about the combined interfaces of the
subsystems. Many of the subsystems’ methods are not relevant to the Client instance’s needs.

- **(BEHAVIOR)** Facade defines behavior for its methods that rely on the subsystems for the actual processing; this behavior could be emulated by the client (as the subsystems are generally public) but is encapsulated within the Facade class to promote high cohesion and loose coupling – Capturing this information in a single class improves readability and maintenance.

A.20 Facade Pattern Redesign

A.20.1 Component Modification Possibilities

- Rule Based Access

  - **Description** – During definition, the Facade is configured with multiple possibilities for a single subsystem. Relational rules defined by Client classes determine which subsystem is used.

  - **Guiding Principle** – Think Non-Deterministically

  - **Guiding Principle** – Think Descriptively

  - **Benefit** – Allows the Client classes to have some level of control by manipulating logical rules without having to understand implementation details of the subsystems.

A.21 Analysis of the GOF Flyweight Pattern

The purpose of the Flyweight Pattern is to “Use sharing to support large numbers of fine-grained objects efficiently”

A.21.1 Metadata

- Forces
– Fine grained objects can support exceptional flexibility and functionality
– The creation of a large number of small objects consumes a large amount of memory resources; this may be unacceptable

• Strengths
– Considerable space savings

• Weaknesses
– Increased computational costs

A.21.2 Concrete Components

• Data
– FlyweightFactory instance
– ConcreteFlyweight instances
– UnsharedConcreteFlyweight instances
– intrinsicState
– extrinsicState
– allState

• Structure
– ConcreteFlyweight implements the Flyweight interface
– UnsharedConcreteFlyweight implements the Flyweight interface
– ConcreteFlyweight instances require externalState from the Client to operate correctly
– **FlyweightFactory** manages creation of all **Flyweight** instances and the sharing of **ConcreteFlyweight** instances

- **Interaction**
  - **Flyweight** publicly provides `operation(extrinsicState)` to the **Client**
  - **FlyweightFactory** publicly provides `getflyweight(key)` to the **Client**

- **Behavior**
  - **ConcreteFlyweight** defines behavior for `operation(extrinsicState)` and is dependent on `extrinsicState` to work correctly
  - **UnsharedConcreteFlyweight** defines behavior for `operation(extrinsicState)` but is actually independent of `extrinsicState` - the interface is kept the same for transparency
  - **FlyweightFactory** defines the creation and sharing behavior for the **ConcreteFlyweight** instances in the `getFlyweight(key)` method; it also provides creation behavior for the **UnsharedConcreteFlyweight** instances

### A.21.3 Metadata/Concrete Component Mapping

- (DATA) **FlyweightFactory instance** – This is the **Client** class’s access point for obtaining flyweights.

- (DATA) **ConcreteFlyweight instances** – These instances of shared objects are limited in number and require extrinsic state to operate correctly

- (DATA) **UnsharedConcreteFlyweight instances** – On the other hand, these objects are fully self contained. However, they are treated the same as the **ConcreteFlyweights** to maintain transparency.
• (DATA) **intrinsicState** – The intrinsicState within a shared object must be context independent. The intrinsicState can be large without incurring memory penalties because it is the intrinsicState that is shared. This is the source of the memory savings.

• (DATA) **extrinsicState** – This information must be provided externally to shared objects for correct operation. This part of a shared object’s state can be computed or stored by the Client. If the extrinsicState is computed, computation time is increased for every call to a shared operation. Even without a computed extrinsicState, the look-up for the correct shared object, and calling it with the stored extrinsicState increases computation time.

• (DATA) **allState** – This is the combination of intrinsic and extrinsic state within an unshared object. Unshared objects do not require the extrinsic state to be provided to them by the Client.

• (STRUCTURE) **ConcreteFlyweight** and UnsharedConcreteFlyweight implement the Flyweight interface – Even though the UnsharedConcreteFlyweight instances do not require extrinsicState to be passed to them (as is defined in the Flyweight interface), they provide the same interface to do so in order to maintain transparency.

• (STRUCTURE) **ConcreteFlyweight instances require externalState from the Client to operate correctly** – These instances must be configured with extrinsic state to operate correctly. The Client must be capable of providing this information by storing it or by computing it on the fly.

• (STRUCTURE) **FlyweightFactory manages creation of all Flyweight instances and the sharing of ConcreteFlyweight instances** – The Client is de-
nied creation and access to both shared and unshared flyweights (in fact, sharing is completely transparent to the Client) and so must access them through this object.

- **(INTERACTION)**  
  
  Flyweight publicly provides operation(extrinsicState) to the Client – The Client works with flyweights through the Flyweight interface. It is completely unaware if it is using a shared or unshared object.

- **(INTERACTION)**  
  
  FlyweightFactory publicly provides getflyweight(key) to the Client – This is the interface for obtaining Flyweight instances.

- **(BEHAVIOR)**  
  
  ConcreteFlyweight defines behavior for operation(extrinsicState) and is dependent on extrinsicState to work correctly – The ConcreteFlyweight class uses intrinsic (context independent) information in conjunction with extrinsic (context sensitive) information to operate.

- **(BEHAVIOR)**  
  
  UnsharedConcreteFlyweight defines behavior for operation(extrinsicState) but is actually independent of extrinsicState – the interface is kept the same for transparency UnsharedConcreteFlyweight’s behavior in the operation(extrinsicState) is independent of what is passed as a parameter. Its behavior is determined without regard to the extrinsicState parameter.

- **(BEHAVIOR)**  
  
  FlyweightFactory defines the creation and sharing behavior for the ConcreteFlyweight instances in the getFlyweight(key) method; it also provides creation behavior for the UnsharedConcreteFlyweight instances
This behavior depends on the FlyweightFactory knowing if the shared flyweight is created or not. If not, it creates it, stores it, and returns it to the Client. If it already exists, it looks up the instance and returns it to the Client.

A.22 Flyweight Pattern Redesign

A.22.1 New Component Possibilities

- Pseudo Flyweight Instances

  - Description – Create a dynamic proxy to shared flyweights using composition to create new methods that tie extrinsicState to shared operations. The methods are bound to the flyweight proxy.
  
  - Guiding Principle – Adapting Similar Problems
  
  - Guiding Principle – Type Refinement
  
  - Benefit – If intrinsic state is very large, space is still conserved without the hassle of a lookup of the shared object.

  - Limitation – The creation of many little objects that causes considerable overhead.

A.23 Analysis of the GOF Proxy Pattern

The purpose of the Proxy Pattern is to “Provide a surrogate or placeholder for another object to control access to it”

A.23.1 Metadata

- Forces

  - Some components within a system cannot or should not be directly accessed by the rest of the system some or all of the time
– The system, however, may need to have some kind of interaction with these components at all times

– Additionally, the restrictions on the components should be transparent to the system

• Strengths

– Provides an always-available, unrestricted interface to clients regardless of the nature of the restrictions on the actual component

– Proxy is transparent to the system simplifying client code

• Weaknesses

– Unknown

A.23.2 Concrete Components

• Data

  – RealSubject instance

  – Proxy instance

• Structure

  – RealSubject implements Subject interface

  – Proxy implements Subject interface

• Interaction

  – Subject publicly provides request() methods to the Client

• Behavior
RealSubject defines the appropriate behavior for request().

Proxy defines an indirection behavior for each method defined in the Subject interface; each method of Proxy responds in place of the RealSubject method it proxies for.

### A.23.3 Metadata/Concrete Component Mapping

- **(DATA)** RealSubject instance – This object represents the “real data”. The purpose of this design pattern is to shield this data in some way.

- **(DATA)** Proxy instance – This is the object that is accessed in place of the shielded RealSubject instance. From the Client instance’s perspective, this is the data.

- **(STRUCTURE)** RealSubject and Proxy both implement the Subject interface – It is essential that the Proxy be of the same type as RealSubject so that the Proxy can be used in place of the RealSubject.

- **(INTERACTION)** Subject publicly provides request() methods to the Client – Client classess are designed to work with the interface. By limiting their knowledge to the Subject interface, they do not need to know the difference between the RealSubject and Proxy instances.

- **(BEHAVIOR)** RealSubject defines the appropriate behavior for request() – The behavior defined by RealSubject is what the Client really wants to access.

- **(BEHAVIOR)** Proxy defines an indirection behavior for each method defined in the Subject interface; each method of Proxy responds in place of the RealSubject method it proxies for – The behavior defined by the Proxy is what
the Client actually gets. Different types of proxies have vastly different behavior ranging from guarding requests to the real object or passing it remotely over a communication channel.

A.24 Proxy Pattern Redesign

A.24.1 Component Modification Possibilities

- Dynamic Interface
  - Description – Allow the Proxy instance’s interface to be defined dynamically. Allow methods of the RealSubject to be “copied” directly into the Proxy instance or as data for the Client. Special methods are created on the fly.
  - Guiding Principle – Adapting Similar Problems
  - Guiding Principle – Function Interchange
  - Benefit – Proxy methods can be defined on the fly if implementation details are not known ahead of time.
  - Benefit – Proxy and RealSubject do not need to share a base class

A.25 Analysis of the GOF Chain of Responsibility Pattern

The purpose of the Chain of Responsibility Pattern is to “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it”

A.25.1 Metadata

- Forces
  - Messages passed in an object-oriented system do not always (and/or should not) have a known receiver when issued
– Sometimes, the receiver may be assigned dynamically

– Alternatively, multiple receivers are, or can be, desired

• Strengths

  – Reduced coupling

  – Added flexibility in assigning responsibilities to objects

• Weaknesses

  – Receipt isn’t guaranteed

A.25.2 Concrete Components

• Data

  – Instances of ConcreteHandler classes

• Structure

  – ConcreteHandler classes implement the Handler interface

• Interaction

  – Handler publicly provides handleRequest() methods to the Client

• Behavior

  – Each ConcreteHandler class defines the appropriate behavior for each handleRequest() method including the manner and timing of passing the request to a successor
A.25.3 Metadata/Concrete Component Mapping

- **(DATA)** Instances of *ConcreteHandler* classes – Each instance is tasked with handling one or more types of requests. The chain of these instances can be viewed as data as well.

- **(STRUCTURE)** *ConcreteHandler* classes implement the *Handler* interface – This structure is necessary for the design pattern to function in strongly-typed languages. But even in weakly-typed languages, the *ConcreteHandler* classes implicitly implement a *Handler* interface. This allows the chain of handlers to be created by linking their common interface.

- **(INTERACTION)** *Handler* publicly provides *handleRequest()* methods to the *Client* – To decouple the sender from the receiver, the sender is only aware of the interface, not the actual subclass that will eventually handle the request. From the *Handler* instance’s perspective, each element in the chain knows that the next element in the chain will provided the same interface and can safely pass the request on to the successor.

- **(BEHAVIOR)** Each *ConcreteHandler* class defines the appropriate behavior for each *handleRequest()* method including the manner and timing of passing the request to a successor – The actual behavior for the majority of the handlers in the chain is to do nothing except pass the request along. If a handler receives a request that it is actually responsible for, it responds to the request and then terminates or passes it on for further handling.

A.26 Chain of Responsibility Pattern Redesign

A.26.1 Component Modification Possibilities

- Compose Handle Methods
– *Description* – Instead of linking objects, compose the `handleRequest()` methods into a single function.

– *Guiding Principle* – Function Composition

– *Benefit* – Unknown

– *Limitation* – Who should maintain the composed function?

### A.26.2 New Component Possibilities

- Relational Store for `Handler` instances

  – *Description* – Eliminate the entire chain. Create a relational store that handlers can be passed to, possibly with descriptive information.

  – *Guiding Principle* – Think Non-Deterministically

  – *Guiding Principle* – Rule Refinement

  – *Benefit* – With descriptive information, the relational store can search for the right handler without the handler becoming active (avoids context switching).

  – *Benefit* – Handlers can be organized and partitioned by increasingly complex logical rules.

  – *Limitation* – Handlers must provide descriptive information to the relational store.

  – *Limitation* – Ordering the handlers can be difficult

### A.27 Analysis of the GOF Command Pattern

The purpose of the Command Pattern is to “Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations”
A.27.1 Metadata

- Forces

  - Sometimes requests must be issued without the issuer knowing anything about the request
  - Or, there may be times where a design solution may require manipulating the request like data (storing the request or passing it like a parameter)
  - Finally, it is desirable to couple the request with ancillary information about the request (e.g., undo)

- Strengths

  - Decouples between the request invocation and request execution
  - Supports easy addition of new commands

- Weaknesses

  - Unknown

A.27.2 Concrete Components

- Data

  - Invoker instance
  - ConcreteCommand instance
  - Receiver instance

- Structure

  - ConcreteCommand implements the Command interface
  - ConcreteCommand is instantiated with a reference to a Receiver instance
• Interaction
  
  – Command publicly provides the `execute()` method to the Invoker

• Behavior
  
  – `ConcreteCommand` overrides the behavior of the `execute()` method to bind a execute an action on the Receiver instance it points to

A.27.3 Metadata/Concrete Component Mapping

• (DATA) `Invoker instance` – This object calls on the `ConcreteCommand` instance to carry out the request decoupling the sender and the receiver of the request.

• (DATA) `ConcreteCommand instance` – Stores the request and the receiver of that request.

• (DATA) `Receiver instance` – Can be of any class and is probably unaware of the Command pattern entirely.

• (STRUCTURE) `ConcreteCommand implements the Command interface` – Client classes understand the Command interface allowing them to work with many different types of commands and allows easy addition of more commands.

• (STRUCTURE) `ConcreteCommand is instantiated with a reference to a Receiver instance` – Each command has to know the receiver of the request. The type of the receiver is generally hard-coded within the `ConcreteCommand` class, but the exact instance is set by the Client.

• (INTERACTION) `Command publicly provides the execute() method to the Invoker` – The Invoker doesn’t understand what the command does. It simply uses the `execute()` method leaving the behavior to the `ConcreteCommand`. 

262
• (BEHAVIOR) *ConcreteCommand* overrides the behavior of the *execute()* method to bind a *execute* an action on the *Receiver* instance it points to – This behavior can be simple (e.g., receiver.request()) or more algorithmically complicated.

A.28 Command Pattern Redesign

A.28.1 Component Modification Possibilities

• Command Function
  
  – *Description* – Replace the Command/*ConcreteCommand* class hierarchy with simple command functions.
  
  – *Guiding Principle* – Lazy Evaluation
  
  – *Guiding Principle* – Function Interchange
  
  – *Benefit* – Reduces class count and object number.
  
  – *Limitation* – It is difficult to restrict the functions to specific types.
  
  – *Limitation* – May be more difficult to read and maintain.

A.29 Analysis of the GOF Interpreter Pattern

The purpose of the *Interpreter* Pattern is to “Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language”

A.29.1 Metadata

• Forces
  
  – Given a small language, classes easily represent the grammar and can contain an interpreter for the sentences of the language

• Strengths
- It’s easy to change and extend the grammar
- Implementing the grammar is easy
- Makes it easier to evaluate an expression in a new way

• Weaknesses

- Complex grammars are hard to maintain

A.29.2 Concrete Components

• Data

  - Context instance
  - Instances of NonterminalExpression classes
  - Instances of TerminalExpression classes

• Structure

  - NonterminalExpression classes implement the AbstractExpression interface
  - TerminalExpression classes implement the AbstractExpression interface
  - The Client provides a Context instance to AbstractExpression instances

• Interaction

  - AbstractExpression publicly provides interpret(Context) to Client classes

• Behavior
NonterminalExpression classes define behavior for interpret(Context) that subsequently calls interpret for other NonterminalExpression and TterminalExpression instances.

TterminalExpression classes define behavior for interpret(Context)

A.29.3 Metadata/Concrete Component Mapping

- (DATA) Context instance – Contains information that is global to the interpreter and should be shared among the components.
- (DATA) Instances of NonterminalExpression classes – Data with pointers to other AbstractExpression instances.
- (DATA) Instances of TterminalExpression classes – Data representing the actual words in the sentence.
- (STRUCTURE) TterminalExpression and NonterminalExpression classes implement the AbstractExpression interface – These classes are used interchangeably within the abstract syntax tree.
- (STRUCTURE) The Client provides a Context instance to AbstractExpression instances – The Context is used to manage data that is the same across all AbstractExpression instances.
- (INTERACTION) AbstractExpression publicly provides interpret(Context) to Client classes – The AbstractSyntaxTree is a type of Composite pattern, the composite structure and terminal elements can be viewed as the same. The interpret(Context) method works at any point in the recursive tree.
• (BEHAVIOR) *NonterminalExpression* classes define behavior for *interpret*(Context) that subsequently calls *interpret* for other *NonterminalExpression* and *TerminalExpression* instances – These classes pass on interpret requests (with context) to other *AbstractExpression* instances that compose it.

• (BEHAVIOR) *TerminalExpression* classes define behavior for *interpret*(Context) – The final behavior of the tree is determined by these nodes.

### A.30 Interpreter Pattern Redesign

#### A.30.1 New Component Possibilities

• Relational Tree Definition
  
  – *Description* – Use relational rules to create the Abstract Syntax Tree. Relational rules easily define (and interpret!) the grammar.

  – *Guiding Principle* – Fact Refinement

  – *Guiding Principle* – Rule Refinement

  – *Benefit* – The creation of the grammar is very simple and easy to understand.

  – *Benefit* – Can be used to create expressions as well as interpret them.

The purpose of the **Iterator** Pattern is to “access the elements of an aggregate object sequentially without exposing its underlying implementation”

#### A.30.2 Metadata

• Forces

  – Information hiding is valued
- High cohesion is valued
- Loose coupling is valued
- Client expects interface for sequential access
- Client may require multiple iteration policies

• Strengths

  - Supports variations in aggregate traversal
  - Simplifies the aggregate interface
  - Allows more than one pending traversal
  - Abstracts internal structure of aggregate

• Weaknesses

  - May require aggregate to export methods (high coupling)
  - May require access to private aggregate data (violates encapsulation)
  - May be difficult to traverse recursive composite structures

A.30.3 Concrete Components

• Data

  - ConcreteAggregate reference (object)
  - Aggregated elements references (objects)
  - Traversal State
  - ConcreteIterator Reference

• Structure

  - ConcreteAggregate implements the Aggregate interface
ConcreteIterator implements the Iterator interface
ConcreteAggregate creates ConcreteIterator
ConcreteIterator operates on ConcreteAggregate

• Interaction

  - Aggregate publicly provides createIterator() to the Client
  - Iterator publicly provides first(), next(), isDone(), and currentItem() to the Client (additional methods are optional)

• Behavior

  - ConcreteAggregate defines the correct iterator to be returned from createIterator()
  - ConcreteIterator defines a traversal policy
    * The first() message sets the current internal state to point to the first aggregate element according to policy
    * The next() message set the current internal state to point to the next element from the current element according to policy
    * The isDone() message requests a Boolean determination of complete transversal of all elements that meet policy requirements
    * The currentItem() message requests the current aggregate element pointed to by the current internal state

A.30.4 Metadata/Concrete Component Mapping

• (ConcreteAggregate Reference) In implementation, instances of the ConcreteIterator class are explicitly tied to the ConcreteAggregate they traverse. Internally, they generally access the ConcreteAggregate through a pointer.
• (References to Aggregated Elements) *The traversing operations require access to the elements within the aggregate structure. This data is the source of the three weaknesses of the Iterator pattern that we have identified.*

• (Privileged Access Violates Encapsulation) *If the ConcreteIterator has access to the private data of the ConcreteAggregate, then modifying the ConcreteAggregate produces cascading side effects in the Iterator and its subclasses.*

• (Special Methods Create a Fragile Interface) *If the ConcreteAggregate exports access methods, then defining new traversals (through new ConcreteIterator classes) may require new methods to be exported in the ConcreteAggregate. While it is expected that the ConcreteIterator depends on the ConcreteAggregate, making the ConcreteAggregate dependent on the ConcreteIterator introduces an awkward coupling.*

• (Recursive Composite Traversal Complicates Implementation) *If the ConcreteIterator is to traverse a recursive composite structure, it must store a traversal path through the composite making the implementation challenging.*

• (Traversal State) *Multiple traversals over the aggregate data is possible because each instance of ConcreteIterator maintains a separate traversal state. However, aggregated data that is recursive may complicate storing the traversal state if storing a path through the recursive data is necessary.*
• (ConcreteIterator Reference) The client accesses the ConcreteIterator directly to control iteration through the data of the ConcreteAggregate. –

• (ConcreteAggregate Implements the Aggregate Interface) This enables the use of the createIterator() factory method. –

• (ConcreteIterator Implements the Iterator Interface) This enables interchangeable iterators. –

• (ConcreteAggregate Creates ConcreteIterator) This frees the Client from having to know which ConcreteIterator corresponds to a given ConcreteAggregate. –

• (ConcreteIterator Operates on ConcreteAggregate) This structure is the heart of the pattern and allows the traversing of the aggregate to be separated from the structure of the aggregate. This reinforces high cohesion by allowing the ConcreteAggregate class to be responsible for the elements and the ConcreteIterator class to be responsible for traversal. –

• (Aggregate publicly provides createIterator() to a client) createIterator() is a factory method. The client calls the createIterator() of a specific ConcreteAggregate to instantiate the appropriate ConcreteIterator. This factory method can be parameterized allowing ConcreteAggregate classes the ability to create instances from different ConcreteIterator classes each with different traversal policies. –

• (Iterator publicly provides first(), next(), isDone(), and currentItem() to a client) The Iterator interface is inherited by all ConcreteIterator classes. These methods abstractly define sequential iteration without reference to any specific concrete implementation. –

270
• (ConcreteAggregate defines the correct iterator to be returned from createIterator()) Each ConcreteAggregate overrides the createIterator() Factory Method to return an appropriate ConcreteIterator instance.

• (ConcreteIterator defines a traversal policy) The details of the traversal policy are hidden from the Client but define a concrete implementation for the abstract methods that are defined by the Iterator interface.

A.31 Iterator Pattern Redesign

A.31.1 Component Modification Possibilities

• Use a relation for aggregated elements references (Logic)
  
  – Description – The ConcreteAggregate provides a logical relation that can query all of the aggregated elements. This relation can operate as a ConcreteIterator but can also be easily extended by new relations created by the Client. This kind of iterator cannot traverse in a directed fashion (e.g., sequential, pre-order, etc.).
  
  – Guiding Principle – Think Non-Deterministically
  
  – Benefit – Extendable traversal policies without subclasses
  
  – Limitation – Cannot direct traversal policies

• Store recursive traversal state in lazy recursive function (Functional)
  
  – Description – Normal recursive functions have been used to support internal iterators for recursive composite structures. Lazy recursive functions allow recursion to be delayed and can be used to provide an external iterator for these structures.
  
  – Guiding Principle – Adapting Similar Problems
• **Guiding Principle** – Recursive Refinement

• **Benefit** – Improved traversal of recursive composite structures

- **Description** – The next(), first(), and isDone() methods are modified to accept a function as an optional argument. This function serves as a filter and can modify the traversal policy.

- **Guiding Principle** – Adapting Similar Problems

- **Guiding Principle** – Pipes and Filters

- **Guiding Principle** – Type Refinement

- **Benefit** – Extendable traversal policies without subclasses

### A.31.2 New Component Possibilities

• Extend ConcreteIterator with logical relation (Logic, Functional)

- **Description** – Additional iterators can be created that are logical relations. The ConcreteIterator can be wrapped into the relations even if the ConcreteIterator is not a relation itself.

- **Guiding Principle** – Think Non-Deterministically

- **Guiding Principle** – Rule Refinement

- **Guiding Principle** – Adapting Similar Problems

- **Guiding Principle** – Type Refinement

- **Guiding Principle** – Functional Composition
Benefit – If the ConcreteIterator is not a logical relation, then both directed iteration and non-deterministic iteration could be applied to the aggregated structure.

Benefit – The Client can extend the ConcreteIterator without understanding the aggregated data structure or the internals of the ConcreteIterator.

- Dynamically create additional iteration methods (Functional)

Description – The Client can dynamically create a new function through currying, composition or anonymous functions.

Guiding Principle – Adapting Similar Problems

Guiding Principle – Type Refinement

Guiding Principle – Functional Composition

Benefit – New functionality within a limited scope (e.g., an individual method) without modifying interfaces or creating new subclasses.

A.32 Analysis of the GOF Mediator Pattern

The purpose of the Mediator Pattern is to “Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.”

A.32.1 Metadata

- Forces

  - Communication (message passing) is an integral concept in object-oriented programming

  - In complex systems, communication between objects can be complex, difficult to maintain, and inhibiting to appropriate generalization
– The communication itself should be captured in a cohesive, self-contained manner

• Strengths
  – It limits subclassing
  – It decouples colleagues
  – It simplifies object protocols
  – It abstracts how objects cooperate

• Weaknesses
  – It can become a monolith that’s hard to maintain

A.32.2 Concrete Components

• Data
  – ConcreteMediator instance
  – Instances of ConcreteColleague classes

• Structure
  – ConcreteColleague classes implement the Colleague interface
  – ConcreteMediator implements the Mediator interface

• Interaction
  – Mediator publicly provides a communication interface to Colleague
  – ConcreteColleague classes publicly provide request operations to Mediator
• Behavior

  – ConcreteMediator defines the actual behavior for the communication system

  – ConcreteColleague classes define appropriate behavior for the requests routed to them by the ConcreteMediator

A.32.3 Metadata/Concrete Component Mapping

• (DATA) ConcreteMediator instance – manages communication between the colleagues.

• (DATA) Instances of ConcreteColleague classes – Communicate with the ConcreteMediator instead of with each other (i.e., it decouples them from each other).

• (STRUCTURE) ConcreteColleague classes implement the Colleague interface – In strongly-typed languages, the Colleague classes must share the same parent class so that they can identify themselves to the Mediator. In weakly-typed languages this is not needed at all.

• (STRUCTURE) ConcreteMediator implements the Mediator interface – Also for use in strongly-typed languages, this allows Colleague classes to know the Mediator interface. In weakly-typed languages the ConcreteMediator must implicitly support the Mediator interface.

• (INTERACTION) Mediator publicly provides a communication interface to Colleague – The Colleague classes only have to understand communication with the Mediator through this interface. This simplifies the communication protocol from many to many into many to one.
• (INTERACTION) *ConcreteColleague* classes publicly provide request operations to *Mediator* – These requests are specific to the *Colleague* class. The *ConcreteColleague* classes are explicitly tied to the *ConcreteMediator*.

• (BEHAVIOR) *ConcreteMediator* defines the actual behavior for the communication system – Unfortunately, this definition can become complex and difficult to change or extend. While it simplifies the *Colleague* classes, it becomes unwieldy in itself.

• (BEHAVIOR) *ConcreteColleague* classes define appropriate behavior for the requests routed to them by the *ConcreteMediator* – These requests are not actually specified by the *ConcreteMediator*. Generally, they already exist as part of the *Colleague* interface. The *Mediator* must know these very specific interfaces and is, therefore, tightly coupled to the system of objects and cannot be generalized.

A.33 Mediator Pattern Redesign

See Observer pattern redesign

A.34 Analysis of the GOF Memento Pattern

The purpose of the Memento Pattern is to “Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later”

A.34.1 Metadata

• Forces

  – Objects generally need to keep their state encapsulated (and hidden)
Yet, without exposing the state publicly, it can never be recorded (or captured), which is sometimes needed.

It is generally burdensome for Objects to be required to store their state internally.

Classes (conceptually and in some languages) can expose their state narrowly to specific classes while keeping it closed to the rest of the System.

- **Strengths**
  - Preserves encapsulation boundaries
  - It alleviates the burden of an object having to store its own previous states

- **Weaknesses**
  - Mementos can be expensive
  - Some languages do not support “narrow” and “wide” interfaces
  - There are hidden costs in maintaining and storing mementos

### A.34.2 Concrete Components

- **Data**
  - *Originator* instance
  - *Memento* instances
  - *Caretaker* instance
  - *Originator* state snapshot

- **Structure**
  - *Memento* instances are created by the *Originator* instance
– **Memento** instance maintains reference to snapshot of **Originator** state
– **Caretaker** instance maintains reference(s) to **Memento** instance(s)

**Interaction**

– **Originator** publicly provides `setMemento(Memento)` and `createMemento()` methods to **Caretaker**
– **Memento** publicly provides `getState()` and `setState()` methods to **Originator**

**Behavior**

– **Caretaker** defines behavior for **Memento** management including requests for memento creation, storage and usage
– **Originator** defines how to create **Memento** instances and how restoration of state behavior when **Memento** instances are passed back to it
– **Memento** defines how **Originator** state snapshot is captured and restored

A.34.3 Metadata/Concrete Component Mapping

• **(DATA) Originator instance** – The purpose of this pattern is to allow this instance to be restored to previous states.

• **(DATA) Memento instances** – These objects hold the state of the **Originator** at some point in time. By storing the **Originator** instance’s state in these objects, the **Originator** is not forced to make public the internal mechanisms of its state and encapsulation is preserved.

• **(DATA) Caretaker instance** – The **Memento** instances produced by the **Originator** are stored for later access within this instance.
• (DATA) **Originator state snapshot** – This data is encapsulated within Memento instances.

• (STRUCTURE) **Memento instances are created by the Originator instance** - Memento instances are generated from the Originator instance. This can incur a significant penalty if a lot of copying is involved. –

• (STRUCTURE) **Memento instance maintains reference to snapshot of Originator state** – The state of the Originator is encapsulated within the Memento object and restored the Originator upon request of the Caretaker instance.

• (STRUCTURE) **Caretaker instance maintains reference(s) to Memento instance(s)** – A simple Caretaker may only keep a single Memento for a single backstep for the originator. More complicated Caretaker may have theoretically unlimited Memento storage. Of course, the Caretaker may have no idea how memory expensive the Memento instances are to store and maintain.

• (INTERACTION) **Originator publicly provides setMemento(Memento) and createMemento() methods to Caretaker** – The Caretaker signals creation of the Memento instances and restoration of them through this interface.

• (INTERACTION) **Memento publicly provides getState() and setState() methods to Originator** – Through this interface the Originator saves and reloads state.

• (BEHAVIOR) **Caretaker defines behavior for Memento management including requests for memento creation, storage and usage** – The Caretaker knows when to create the Memento instances, how they should be maintained, and when they should be restored.
• (BEHAVIOR) **Originator** defines how to create **Memento** instances and how restoration of state behavior when **Memento** instances are passed back to it
  – While the Caretaker knows when the **Memento** instances are created, the **Originator** defines how the state should be saved and how state should be restored from the **Originator**’s perspective. Even though the **Memento** class actually saves the state, the **Originator** is responsible for actually making the request. This allows **Originator** to make sure it is in a ’saveable’ configuration before attempting to save.

• (BEHAVIOR) **Memento** defines how **Originator** state snapshot is captured and restored – Using a special interface, the **Memento** can peer into the **Originator** and snapshot the interface. The restoration process is generally a simple process of returning values to the **Originator** and the **Originator** actually changes it state back.

A.35 Memento Pattern Redesign

A.35.1 New Component Possibilities

• Memento Function

  – *Description* – Create a function dynamically that will restore the **Originator** to a certain state. The function is tied to the **Originator** that created it and can be executed by the Caretaker.

  – *Guiding Principle* – Delayed (Lazy) Evaluation

  – *Benefit* – Eliminates the **Memento** class

  – *Benefit* – Can be explicitly tied to a specific **Originator** class
A.36 Analysis of the GOF Observer Pattern

The purpose of the Observer Pattern is to “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically”

A.36.1 Metadata

- Forces
  - Object-oriented design and implementation naturally leads to system-partitioning and decomposition
  - Related objects are often separated across partition boundaries
  - Synchronization and Communication are often perpendicular to class hierarchies

- Strengths
  - Reduces coupling between communicating objects
  - Creates support for broadcast communication

- Weaknesses
  - Update protocol has no prediction of update cost, is not easily traced, and cannot determine what changed (or how or when)

A.36.2 Concrete Components

- Data
  - ConcreteSubject instance
  - Instances of ConcreteObserver classes
  - subjectState
- observerState

• Structure

- `ConcreteSubject` implements `Subject` interface
- `ConcreteObserver` classes implement `Observer` interface
- `ConcreteSubject` instance maintains references to `Observer` instances
- `ConcreteObserver` instances maintain references to the `ConcreteSubject` instance

• Interaction

- `Subject` publicly provides `attach(Observer)`, `detach(Observer)`, and `notify()` methods to `Client` classes (which may include `Observer` classes)
- `ConcreteSubject` publicly provides `getState()` and `setState()` methods to `Client` classes (including instances of `ConcreteObserver` classes)
- `Observer` publicly provides `update()` to `Subject`

• Behavior

- `Subject` defines the behavior for attaching and detaching `Observer` instances and notifying them of state changes
- `ConcreteSubject` defines behavior for allowing `Client` classes (including instances of `ConcreteObserver` classes) to get and set its internal state (`subjectState`)
- `ConcreteObserver` classes define appropriate behavior for `update()` that will reconcile its state (`observerState`) with that of the `ConcreteSubject` instance
A.36.3 Metadata/Concrete Component Mapping

- (DATA) **ConcreteSubject instance** – From the perspective of the design pattern, this is the data that is directly changeable. However, a **ConcreteObserver** instance may change the state of the **ConcreteSubject**.

- (DATA) **Instances of ConcreteObserver classes** – On the other hand, these instances observe the changes in the **ConcreteSubject** instance.

- (DATA) **subjectState** – Stored in the **ConcreteSubject** instance, the **subjectState** can be altered from many sources: **Client** instances, **ConcreteObserver** instances, or even the **ConcreteSubject** itself.

- (DATA) **observerState** – This data is dependent on the **subjectState** in some way, though it is conceivable that direct changes to the **observerState** are acceptable and may trigger changes to **subjectState**.

- (STRUCTURE) **ConcreteSubject implements Subject interface** – **Observer** instances know how to interface with the **Subject** interface allowing them to work with various **ConcreteSubject** instances.

- (STRUCTURE) **ConcreteObserver classes implement Observer interface** – **ConcreteSubject** instances operate on **ConcreteObserver** classes through the **Observer** interface, thus allowing them to update various **ConcreteObservers**.

- (STRUCTURE) **ConcreteSubject instance maintains references to Observer instances** – The **ConcreteSubject** instance is tasked with maintaining connections to observing objects.

- (STRUCTURE) **ConcreteObserver instances maintain references to the**
ConcreteSubject instance – ConcreteObserver instances must have access to the ConcreteSubject as well in order to check for changes in subject state.

• (INTERACTION) Subject publicly provides attach(observer), detach(observer), and notify() methods to Client classes (which may include Observer classes) – In the GOF pattern, their default interaction shows a ConcreteObserver instance setting state on the ConcreteSubject and then be notified by that object of the changes. However, GOF also points out that notify may be called by any object.

• (INTERACTION) ConcreteSubject publicly provides getState() and setState() methods to Client classes (including instances of ConcreteObserver classes) – In the GOF pattern, a ConcreteObserver instance sets the state on the ConcreteSubject and then is notified of the changes. The ConcreteObserver uses the setState() method to change the ConcreteSubject and then uses the getState() to check for changes.

• (INTERACTION) Observer publicly provides update() to Subject – Through this interface, the ConcreteSubject can update all associated Observer instances.

• (BEHAVIOR) Subject defines the behavior for attaching and detaching Observer instances and notifying them of state changes – Attaching and detaching behavior is generally straightforward, but notification of changes varies based on the complexity of the solution as a whole, the communication protocol between subject and observer, and other related factors. In the simple case, the ConcreteSubject iterates through the list of associated Observer instances and calls their update() method with no parameters.
• **(BEHAVIOR)** `ConcreteSubject` defines behavior for allowing `Client` instances (including instances of `ConcreteObserver` classes) to get and set its internal state (`subjectState`) – In some circumstances, `ConcreteObserver` instances will set the state of the `ConcreteSubject`. In other cases, the `update()` method called by the `ConcreteSubject` passes them no information and they turn around and request `getState()` from the `ConcreteSubject`. More complex communication protocols move away from this approach, but the basic concept is the same.

• **(BEHAVIOR)** `ConcreteObserver` classes define appropriate behavior for `update()` that will reconcile its state (`observerState`) with that of the `ConcreteSubject` instance – The default behavior is to query the `ConcreteSubject` for its current state (`getState()`) and update itself accordingly.

## A.37 Observer Pattern Redesign

### A.37.1 Component Modification Possibilities

- Relational Object Communication System
  
  - *Description* – Connect objects through relational rules and allow them to pass messages to associated listeners.

  - *Guiding Principle* – Fact Refinement

  - *Guiding Principle* – Rule Refinement

  - *Benefit* – This general system can work either as a Mediator or a Change-Manager for the Subject/Observer

  - *Benefit* – The connections between objects can be made by description

  - *Limitation* – Sender has no guarantee of the receipt of the message
A.38 Analysis of the GOF State Pattern

The purpose of the State Pattern is to “Allow and object to alter its behavior when its internal state changes. The object will appear to change its class”

A.38.1 Metadata

• Forces
  
  – Conceptually, some software solutions naturally suggest dynamic object reclassification
  
  – Object-Oriented design supports this
  
  – Many Object-Oriented languages do not

• Strengths
  
  – It localizes state-specific behavior and partitions behavior for different states
  
  – It makes state transitions explicit
  
  – State objects can be shared

• Weaknesses
  
  – It introduces a (possibly) large number of additional classes, pointer indirection, and overhead
  
  – Tightly couples State subclasses
  
  – It is difficult to introduce additional operations to the context (changes to the interface must be propagated do all State subclasses)

A.38.2 Concrete Components

• Data
Context instance

ConcreteState instances

• Structure

  ConcreteState classes implement the State interface

  Context is responsible for managing State instances

• Interaction

  Context publicly provides request() methods to Client classes

  State publicly provides handle() methods to Context

• Behavior

  Context defines a simple behavior for request() methods which generally
  reduces to passing the request off to some handle() method of the State
  instance

  ConcreteState classes define state appropriate behavior for handle() methods

  Context or the ConcreteState classes (or both) must define the state-
  changing behavior

A.38.3 Metadata/Concrete Component Mapping

• (DATA) Context instance – Client classes interact with the Context. In fact,
  as far as the system is concerned, the Context is all that exists.

• (DATA) ConcreteState instances – These instances, hidden within the
  Context, alter how the Context interacts with Client classes.

287
• (STRUCTURE) *ConcreteState* classes implement the *State* interface – The *State* interface mirrors closely or exactly the *Context* interface because the *Context* forwards requests on to the *ConcreteState* instances through the *State* interface.

• (STRUCTURE) *Context* is responsible for managing *State* instances – Client classes will not see, create, or manage the instances of *State*. Instead, the *Context* instance access the *State* instances as needed. Creation is either handled by the *Context* instance or a flyweight or other management object known to the *Context*.

• (INTERACTION) *Context* publicly provides *request()* methods to *Client* classes – This is the only interface *Client* classes are aware of. They do not interact with the *State* instances.

• (INTERACTION) *State* publicly provides *handle()* methods to *Context* – The *Context* works with the *handle()* methods interface. The *ConcreteState* instance determines the appropriate behavior.

• (BEHAVIOR) *Context* defines a simple behavior for *request()* methods which generally reduces to passing the request off to some *handle()* method of the *State* instance – These methods generally have little complexity or functionality other than to hand off the request to *ConcreteState* instances. Even if the *request()* behavior is the same for all states, it is better to hand it off to a *ConcreteState* instance for easy maintenance (such as adding new states).

• (BEHAVIOR) *ConcreteState* classes define state appropriate behavior for *handle()* methods – This behavior eliminates the need for complex conditional statements in a monolithic object.
• **(BEHAVIOR) Context or the ConcreteState classes (or both) must define the state-changing behavior** – In the GOF example, the ConcreteState classes know when and how to change state. This functionality could also be recorded in the Context.

A.39 Observer Pattern Redesign

A.39.1 New Component Possibilities

• Relational State

  – *Description* – Replace ConcreteState system with a single relational table that stores appropriate functions for appropriate state. Each method of the Context is stored with additional methods organized by state.

  – *Guiding Principle* – Rule Refinement

  – *Guiding Principle* – Function Interchange

  – *Benefit* – Eliminates a complicated tree hierarchy

  – *Benefit* – Ties state operations and state transitions together

  – *Limitation* – It is very similar to table-driven transitions. See GOF page 308 for more.

A.40 Analysis of the GOF Strategy Pattern

The purpose of the Strategy Pattern is to “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it”

A.40.1 Metadata

• Forces

  – Some dynamic systems require the same operation to use different algorithms at different times and places

289
- Imperative control statements (if, case, etc.) create large, monolithic structures that are difficult to understand and maintain
- Conceptually, the algorithm and the context are separate entities

- **Strengths**
  - Creates families of related algorithms
  - Supports dynamic algorithm switching
  - Eliminates conditional statements

- **Weaknesses**
  - **Client** classes must understand strategy implementation
  - Not all algorithms require the same interface/communication complexity
  - Increased overhead

**A.40.2 Concrete Components**

- **Data**
  - **Context** instance
  - Instances of **ConcreteStrategy** classes

- **Structure**
  - **ConcreteStrategy** classes implement the **Strategy** interface
  - **Context** instance maintains a reference to the **ConcreteStrategy** instance it has been configured with

- **Interaction**
– Context publicly provides `contextInterface()` methods to Client classes

– Strategy publicly provides `algorithmInterface()` methods to Context

• Behavior

– Context defines a behavior for each `contextInterface()` method that access `algorithmInterface()` methods of the ConcreteStrategy instance

– Each ConcreteStrategy class defines algorithmically appropriate behavior for each `algorithmInterface()` method

A.40.3 Metadata/Concrete Component Mapping

• (DATA) Context instance – Client classes interact with the Context instance only.

• (DATA) Instances of ConcreteStrategy classes – Used by the Context instance for full functionality.

• (STRUCTURE) ConcreteStrategy classes implement the Strategy interface – The Context instance will use the Strategy interface unaware of the ConcreteStrategy instance it is using.

• (STRUCTURE) Context instance maintains a reference to the ConcreteStrategy instance it has been configured with – It uses the ConcreteStrategy instance to fulfill its functionality. A weakness of this pattern is in this structure. The configuration of the Context instance requires an understanding of how the ConcreteStrategy instance works.

• (INTERACTION) Context publicly provides `contextInterface()` methods
to Client classes – The Client classes use this interface unaware that the algorithms underneath them are changing.

- (INTERACTION) Strategy publicly provides algorithmInterface() methods to Context – The Context class defines its behavior using the algorithmInterface methods defined by Strategy.

- (BEHAVIOR) Context defines a behavior for each contextInterface() method that access algorithmInterface() methods of the ConcreteStrategy instance – The Behavior of Context shouldn’t appear to change (that would be State). Rather, how the behavior is achieved changes.

- (BEHAVIOR) Each ConcreteStrategy class defines algorithmically appropriate behavior for each algorithmInterface() method – Another weakness is found in this behavior. Within the various ConcreteStrategy subclasses, one may need to define behavior for an interface method that the other does not. Worse yet, a new ConcreteStrategy may require an interface method that is not available.

A.41 Strategy Pattern Redesign

No acceptable general modifications.

A.42 Analysis of the GOF Template Method Pattern

The purpose of the Template Method Pattern is to “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure”

A.42.1 Metadata

- Forces
It is convenient to capture behavior for some operations in the parent class.

Sometimes, it is desirable to enforce behavior constraints on subclasses at the parent class.

However, some steps (or component behaviors) of the operation are unknown to the parent class.

- **Strengths**
  
  - Promotes code reuse
  
  - Enables invariant algorithms passed from parent classes to subclasses

- **Weaknesses**
  
  - Locks behavior high in the class hierarchy

### A.42.2 Concrete Components

- **Data**
  
  - `ConcreteClass` instance

- **Structure**
  
  - `ConcreteClass` implements `AbstractClass` interface

- **Interaction**
  
  - `AbstractClass` privately provides `primitiveOperation()` methods to subclasses
  
  - `AbstractClass` publicly provides `templateMethod()` to `Client` classes (and `ConcreteClass` should NOT override or modify this method)

- **Behavior**
AbstractClass defines the behavior of the templateMethod() using
primitiveOperation() methods

ConcreteClass defines the behavior of the primitiveOperation() methods

A.42.3 Metadata/Concrete Component Mapping

- **(DATA) ConcreteClass instance** – The template method will be accessed
  through this instance. Client classes will have no knowledge of the nature of
  the template method.

- **(STRUCTURE) ConcreteClass implements AbstractClass interface** –
  This allows the ConcreteClass to take advantage of the template method de-
  fined in AbstractClass.

- **(INTERACTION) AbstractClass privately provides
  primitiveOperation() methods to subclasses** – These methods are de-
  signed as hooks; they allow the subclasses to hook into the template method.

- **(INTERACTION) AbstractClass publicly provides templateMethod() to
  Client classes (and ConcreteClass should NOT override or modify this
  method)** – Client instances access the templateMethod() without concern for
  the ConcreteClass specifically.

- **(BEHAVIOR) AbstractClass defines the behavior of the templateMethod()
  using primitiveOperation() methods** – This behavior gives the pattern its
  name. The templateMethod() is defined using operations that are not yet
defined.

- **(BEHAVIOR) ConcreteClass defines the behavior of the
  primitiveOperation() methods** – Through these hook methods, the
ConcreteClass controls aspects of the templateMethod() without changing the template.

A.43 Template Method Pattern Redesign

A.43.1 Component Modification Possibilities

- Functional Template
  - Description – Allow functions to be passed in rather than overridden by subclasses.
  - Guiding Principle – Adapting Similar Problems
  - Guiding Principle – Functional Composition
  - Guiding Principle – Functional Interchange
  - Benefit – No need for subclasses
  - Benefit – Dynamic functions can be used to alter the templateMethod()

A.44 Analysis of the GOF Visitor Pattern Pattern

The purpose of the Visitor Pattern is to “Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates”

A.44.1 Metadata

- Forces
  - Large object structures can be complex in form and function
  - Adding new operations to apply to the entire structure require modification of most or all of the objects within the structure, which is difficult to implement and likely to produce error
  - Some new operations may be unrelated to the objects individually
- Some object structures change very little during the life of the product

- **Strengths**
  - Adding new operations is easy
  - Gathers related operations and separates unrelated ones
  - Can travel across class hierarchies
  - Can accumulate state

- **Weaknesses**
  - Adding new ConcreteElement classes is hard
  - Weakens encapsulation

### A.44.2 Concrete Components

- **Data**
  - * Object Structure composed of instances of ConcreteElement classes
  - * Instances of ConcreteVisitor classes

- **Structure**
  - * ConcreteElement classes implement the Element interface
  - * ConcreteVisitor classes implement the Visitor interface
  - * Element instances are configured with Visitor instances

- **Interaction**
  - * Element publicly provides the accept(Visitor) method to the Client
  - * ConcreteElement classes publicly provide operation() methods to Client classes and ConcreteVisitor classes
* Visitor publicly provides \texttt{visitConcreteElement()} elements to \texttt{Element} and subclasses

  - Behavior

  * Each \texttt{ConcreteElement} class defines behavior for the \texttt{accept(Visitor)} method that calls the appropriate \texttt{Visitor.visitConcreteElement()} method

  * Each \texttt{ConcreteElement} class defines the behavior of the \texttt{operation()} methods that the \texttt{Visitor} will call

  * Each \texttt{ConcreteVisitor} subclass defines the correct behavior for each \texttt{visitConcreteElement} method

A.44.3 Metadata/Concrete Component Mapping

- (DATA) \textit{Object Structure composed of instances of \texttt{ConcreteElement} classes}
  - This structure is generally a type of Composite. The visitor pattern makes less sense for non-composite structures.

- (DATA) \textit{Instances of \texttt{ConcreteVisitor} classes} – These instances each represent a “new” operation that can be performed on the Object Structure.

- (STRUCTURE) \textit{ConcreteElement classes implement the \texttt{Element} interface}
  - This interface allows the \texttt{Visitor} instances to operate on the Object Structure.

- (STRUCTURE) \textit{ConcreteVisitor classes implement the \texttt{Visitor} interface}
  - This interface allows the \texttt{Element} instances to support the \texttt{Visitor} instances operation. This structure allows for “double dispatch” (see GOF 338).

- (STRUCTURE) \textit{Element instances are configured with \texttt{Visitor} instances}
- This configuration is the calling of the new operation represented by the ConcreteVisitor.

- **(INTERACTION)** *Element* publicly provides the *accept(Visitor)* method to the *Client* – This is the interface that allows a *Visitor* to visit an *Element* instance.

- **(INTERACTION)** *ConcreteElement* classes publicly provide *operation()* methods to *Client* classes and *ConcreteVisitor* classes – The operations are generally provided for *Client* classes and then subsequently accessed by *ConcreteVisitor* classes. One of the weaknesses of this pattern is that sometimes operations are required by *ConcreteVisitor* classes that are not required by *Client* classes in general. This breaks encapsulation.

- **(INTERACTION)** *Visitor* publicly provides *visitConcreteElement()* elements to *Element* and subclasses – This allows the *Element* instance, once it is accepts the *Visitor* instance to do the correct call back (double dispatch).

- **(BEHAVIOR)** Each *ConcreteElement* class defines behavior for the *accept(Visitor)* method that calls the appropriate *Visitor.visitConcreteElement()* method – This is generally the sole purpose of the *accept(Visitor)* method.

- **(BEHAVIOR)** Each *ConcreteElement* class defines the behavior of the *operation()* methods that the *Visitor* will call – These operations are generally defined without reference to the *Visitor* instances. In other words, their general behavior is driven by *Client* (not *Visitor*) needs.

- **(BEHAVIOR)** Each *ConcreteVisitor* subclass defines the correct behavior for each *visitConcreteElement()* method – Each one of these methods rep-
represents a certain aspect of the new operation. Generally, the new operation is over the entire composite structure, and so, each method operates on a different component within that object structure.

A.45 Visitor Pattern Redesign

No effective redesign possibilities were found.