An Object-Oriented Methodology and Supporting Framework for
Creating Engineering Software by Dynamic Integration

by
Scott A. Woyak

dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Mechanical Engineering

APPROVED:

Dr. Arvid Myklebust, Chairman

Dr. W. F. O’Brien

Dr. M. P. Deisenroth

Dr. J. H. Bøhn

Dr. J. R. Mahan

January 18, 1995
Blacksburg, Virginia
AN OBJECT-ORIENTED METHODOLOGY AND SUPPORTING FRAMEWORK
FOR CREATING ENGINEERING SOFTWARE BY DYNAMIC INTEGRATION

by
Scott A. Woyak

Dr. Arvid Myklebust, Chairman

Mechanical Engineering

Software design within the engineering community has generally been relegated to encoding algorithms for the purpose of executing them very rapidly. This is a very important purpose, however substantially more is required to build an entire CAD application. Structure must be provided to the data maintained in the application. Various analyses must be integrated and coordinated in an orderly fashion. Interaction with the user must be managed. These topics have traditionally received secondary attention. The result has been engineering applications that are difficult to use, costly to create, and expensive to maintain or modify. The system created in this dissertation, the Dynamic Integration System, addresses these issues with respect to engineering-related software. Code constructed with Dynamic Integration System techniques anticipate future needs, such as integration, before those needs explicitly arise. This greatly reduces downstream costs and facilitates the development of engineering-related software. The Dynamic Integration System consists of two primary constructs: Dynamic Variables and dependency hierarchies. Dynamic Variables are used to model the key parameters in an application while a dependency hierarchy is built from the relationships between Dynamic Variables. Using these constructs, issues such as integration and analysis coordination are automated by the underlying Dynamic Integration System facilities.
Acknowledgments

The last several years have been hectic and strenuous, but an enjoyable time for me. In that time, a lot has been accomplished to further the techniques used to develop engineering software. There have been many people who have helped me along the way.

Dr. Myklebust, my advisor, has consistently provided me with a place to work and equipment to use. And while I did not always know how he did it, I always had a source of funding. I thank Dr. Myklebust for providing me the opportunity to showcase my talents as a graduate student at Virginia Tech.

I thank my committee, consisting of Dr. Bohn (who replaced Dr. Jayaram when he left for a position at the Washington State University), Dr. Disenroth, Dr. O’Brien, and Dr. Mahan, for showing interest in the CAD side of mechanical engineering. While some people are resistant to accepting computer programming into engineering, my committee has approached it with an inquisitive eye.

Another Ph.D. student, Brett Malone, has been very instrumental in the development and acceptance of the Dynamic Integration System. Brett guided me as to the needs of the aircraft industry and was the first person to start using my code. His efforts have enabled me to really demonstrate the full capabilities of the Dynamic Integration System. I thank Brett for having faith in me and for helping me get my work recognized by the aircraft industry.
For the duration of my dissertation work I have been funded by the ACSYNT Institute, which is primarily funded by the NASA Ames Research Center. I owe a lot of gratitude to NASA for keeping the Institute going so that I could complete my research. I also thank the various members of the Institute, including Northrop, Lockheed, and GE, who have contributed funding throughout the years.

Perhaps a reason for the continued funding from NASA has been that we keeping sending employees their way. Such is the case with Shahab Hasan. Shahab was only here for the initial planning stages of the Dynamic Integration System, but he has contributed a great deal. I thank Shahab for keeping NASA up-to-date with my work and for keeping me up-to-date with their needs. The dialogue provided by Shahab has been very valuable. Shahab has also served as my personal proof-reader for the past several years. If you do not notice any mistakes in this dissertation, it can be directly attributed to his attention to detail when proof-reading.

I had the misfortune of being diagnosed with Burkitt’s lymphoma midway through my graduate studies. Fortunately, I am now able to look upon that as part of my past. I have many people to thank for my recovery. I thank my oncologists Dr. Pickens and Dr. McCoy for providing me with treatment. I also thank the many nurses at Abington Memorial and Montgomery Regional hospitals. The nursing profession is a very under-appreciated profession.

Lastly, but certainly not least, I thank my wife Laura. It was initially a rough road for her finding employment in Blacksburg, but things worked out in the end. She is now the Marketing Manager for EnviroTech Mid-Atlantic. I thank Laura for being patient while I completed my degree. It’s tough to be a graduate student, but perhaps even more difficult to be the spouse of a graduate student. I also thank Tammi Decatur for being such a good friend to Laura while I was busy many-a-night working on my computer.

Acknowledgments
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>DISSECTATION ORGANIZATION</td>
<td>1</td>
</tr>
<tr>
<td><strong>CHAPTER 1 - INTRODUCTION</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>CHAPTER 2 - MOTIVATION</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 ACSYNT HISTORY</td>
<td>5</td>
</tr>
<tr>
<td>2.2 THE CURRENT STRUCTURE OF ACSYNT AND ITS ASSOCIATED PROBLEMS</td>
<td>6</td>
</tr>
<tr>
<td>2.3 REMODELING ACSYNT</td>
<td>9</td>
</tr>
<tr>
<td>2.4 SUMMARY</td>
<td>12</td>
</tr>
<tr>
<td><strong>CHAPTER 3 - PROBLEM STATEMENT AND GOALS</strong></td>
<td>13</td>
</tr>
<tr>
<td>3.1 PROBLEM STATEMENT</td>
<td>13</td>
</tr>
<tr>
<td>3.2 DETAILED PROBLEM STATEMENT</td>
<td>14</td>
</tr>
<tr>
<td>3.2.1 Providing Interfaces to Analyses</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1.1 Data Files</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1.2 Function Procedures</td>
<td>18</td>
</tr>
<tr>
<td>3.2.1.3 Global Data</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1.4 Centralized Database</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2 Managing Integration Problems</td>
<td>23</td>
</tr>
<tr>
<td>3.2.2.1 Maintaining Consistency Among Modules</td>
<td>24</td>
</tr>
<tr>
<td>3.2.2.2 Responding to Changes</td>
<td>26</td>
</tr>
<tr>
<td>3.3 GOALS</td>
<td>26</td>
</tr>
<tr>
<td>3.4 APPROACH</td>
<td>27</td>
</tr>
<tr>
<td>3.5 SUMMARY</td>
<td>28</td>
</tr>
<tr>
<td><strong>CHAPTER 4 - INFLUENCES AND RELATED LITERATURE</strong></td>
<td>30</td>
</tr>
<tr>
<td>4.1 OBJECT-ORIENTED DESIGN</td>
<td>31</td>
</tr>
<tr>
<td>4.1.1 What is Object-Oriented Design?</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2 Why Use the Object-Oriented Approach</td>
<td>32</td>
</tr>
<tr>
<td>4.1.3 Example Object-Oriented Applications</td>
<td>33</td>
</tr>
<tr>
<td>4.1.4 Object-Oriented Numerical Research</td>
<td>36</td>
</tr>
<tr>
<td>4.2 DESIGN METHODS</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1 Example Methodologies</td>
<td>39</td>
</tr>
<tr>
<td>4.2.1.1 Booch</td>
<td>39</td>
</tr>
<tr>
<td>4.2.1.2 OMT - Object Modeling Technique</td>
<td>40</td>
</tr>
</tbody>
</table>
5.4.4.5 Dfloat Dependency Add .......................................................... 114
5.4.4.6 Dfloat Dependency Remove .................................................. 115

5.5 PATTERNS ................................................................................. 116
5.5.1 Creating and using a Dynamic Variable .................................... 116
5.5.2 Monitoring the value of a Dynamic Variable .............................. 117
5.5.3 Monitoring the value of multiple Dynamic Variables ................. 120
5.5.4 Connecting a user interface element to a Dynamic Variable ....... 122
5.5.5 Creating an analysis using Dynamic Variables ......................... 124
  5.5.5.1 Modeling an equation ....................................................... 125
5.5.6 Connect existing C or FORTRAN code to Dynamic Variables .... 133
  5.5.6.1 Eager-evaluation ............................................................. 133
  5.5.6.2 Lazy-evaluation ............................................................. 136
5.5.7 Performing iteration ............................................................. 139
5.5.8 Obtaining parameters related to a Dynamic Variable ................. 149
5.5.9 Monitoring the state of a Dynamic Variable ............................ 150
5.5.10 Monitoring actions occurring to a Dynamic Entity ............... 152
5.5.11 Monitoring action occurring in the Dynamic Registry ............ 154
5.5.12 How to model a bi-directional constraint .................................. 155
5.5.13 How to create a new Dynamic Data Type .............................. 161

CHAPTER 6 - A DESIGN METHODOLOGY .............................................. 170
6.1 APPLICATION STRUCTURE ......................................................... 170
6.2 METHODOLOGY DOCUMENTATION ............................................ 175

CHAPTER 7 - SUPPORT TOOLS ......................................................... 180
7.1 DEPENDENCY HIERARCHY VIEWER ........................................... 181
7.2 MONITOR .................................................................................. 182
7.3 OPTIMIZER .............................................................................. 184
7.4 GRAPHER .................................................................................. 186
7.5 DYNAMIC REGISTRY VIEWER .................................................... 187
7.6 DYNAMIC REGISTRY MONITOR ............................................... 188
7.7 TYPE-AHEAD ........................................................................... 189
7.8 EXAMINER .............................................................................. 189

CHAPTER 8 - EXAMPLE APPLICATIONS .............................................. 191
8.1 MASS-SPRING-DAMPER ANALYSIS PROGRAM ......................... 191
  8.1.1 Modules ............................................................................. 192
  8.1.2 Module Assembly ............................................................... 195
  8.1.3 Dynamic Module Specification Diagrams ............................. 197
  8.1.4 Alternate Implementation .................................................... 199
8.2 AIRCRAFT ANALYSIS EXAMPLE ............................................... 201
  8.2.1 Analysis Controller ............................................................. 202
  8.2.2 Wing Geometry Analysis ..................................................... 202
  8.2.3 Weights Analysis ............................................................... 203
  8.2.4 Propulsion Analyses .......................................................... 206
  8.2.5 Lift Surface Analysis .......................................................... 209
  8.2.6 Aerodynamic Forces Analysis ............................................. 210
  8.2.7 Drag Buildup Analysis ......................................................... 210
  8.2.8 Dynamic Pressure Analysis ................................................ 211

CHAPTER 9 - CONCLUSIONS ............................................................. 212
# REFERENCES

215

# GLOSSARY

232

# APPENDIX A - MASS-SPRING-DAMPER ANALYSIS

240

A.1 PARAMETERS .................................................................................. 240
A.2 ANALYSIS ..................................................................................... 240
   A.2.1 Steady-State Solution ..................................................... 241
   A.2.2 Transient Solution .......................................................... 242

# APPENDIX B - OBJECT-ORIENTED DESIGN WITH C++ EXAMPLES

243

B.1 ABSTRACTION ............................................................................. 243
B.2 ENCAPSULATION ....................................................................... 244
   B.2.1 Objects .............................................................................. 244
   B.2.2 Classes ............................................................................. 244
B.3 INHERITANCE ............................................................................ 246
B.4 POLYMORPHISM ...................................................................... 250

# APPENDIX C - RESPONSIBILITY-DRIVEN DESIGN

254

C.1 EXPLORATORY PHASE ............................................................... 254
   C.1.1 Classes .............................................................................. 255
   C.1.2 Responsibilities ............................................................. 256
   C.1.3 Collaborations ................................................................. 238
C.2 ANALYSIS PHASE .................................................................... 259
   C.2.1 Hierarchies ....................................................................... 259
   C.2.2 Contracts ......................................................................... 262

# VITA

267

Table of Contents
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACSYNT STRUCTURE</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>ACSYNT GLOBAL DATA MANAGEMENT</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>MASS-SPRING-DAMPER SYSTEM</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>EXAMPLE DEPENDENCY HIERARCHY</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>PARTIAL MASS-SPRING-DAMPER DEPENDENCY HIERARCHY</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>INVALIDATION PROCESS</td>
<td>59</td>
</tr>
<tr>
<td>7</td>
<td>EVENT-RESPONSE PAIR</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>APPLICATION OF EVENT-RESPONSE PAIRS</td>
<td>65</td>
</tr>
<tr>
<td>9</td>
<td>EXAMPLE DYNAMIC VARIABLE USAGE</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>DYNAMIC REGISTRY</td>
<td>75</td>
</tr>
<tr>
<td>11</td>
<td>DYNAMIC ENTITY</td>
<td>76</td>
</tr>
<tr>
<td>12</td>
<td>DYNAMIC VARIABLE</td>
<td>78</td>
</tr>
<tr>
<td>13</td>
<td>EXAMPLE APPLICATION</td>
<td>79</td>
</tr>
<tr>
<td>14</td>
<td>DYNAMIC VARIABLE CLASS HIERARCHY</td>
<td>81</td>
</tr>
<tr>
<td>15</td>
<td>LINKED LIST CLASS HIERARCHY</td>
<td>82</td>
</tr>
<tr>
<td>16</td>
<td>DYNAMIC INTEGRATION SYSTEM COLLABORATIONS GRAPH</td>
<td>104</td>
</tr>
<tr>
<td>17</td>
<td>ENTITY SUBSYSTEM COLLABORATIONS GRAPH</td>
<td>105</td>
</tr>
<tr>
<td>18</td>
<td>DYNAMIC COLLABORATIONS GRAPH</td>
<td>106</td>
</tr>
<tr>
<td>19</td>
<td>DREGISTRY COLLABORATIONS GRAPH</td>
<td>106</td>
</tr>
<tr>
<td>20</td>
<td>SET SUBSYSTEM COLLABORATIONS GRAPH</td>
<td>107</td>
</tr>
<tr>
<td>21</td>
<td>DRESPONDER COLLABORATIONS GRAPH</td>
<td>107</td>
</tr>
<tr>
<td>22</td>
<td>EXAMPLE DYNAMIC VARIABLE USAGE</td>
<td>116</td>
</tr>
<tr>
<td>23</td>
<td>EXAMPLE DYNAMIC VARIABLE MONITORING CLASS DECLARATION</td>
<td>118</td>
</tr>
<tr>
<td>24</td>
<td>EXAMPLE DYNAMIC VARIABLE MONITORING CLASS DEFINITION</td>
<td>118</td>
</tr>
<tr>
<td>25</td>
<td>EXAMPLE MULTIPLE DYNAMIC VARIABLE MONITORING CLASS DECLARATION</td>
<td>120</td>
</tr>
<tr>
<td>26</td>
<td>EXAMPLE MULTIPLE DYNAMIC VARIABLE MONITORING CLASS DEFINITION</td>
<td>121</td>
</tr>
<tr>
<td>27</td>
<td>EXAMPLE USER-INTERFACE ELEMENT CLASS DECLARATION</td>
<td>123</td>
</tr>
<tr>
<td>28</td>
<td>EXAMPLE USER-INTERFACE ELEMENT CLASS DEFINITION</td>
<td>123</td>
</tr>
<tr>
<td>29</td>
<td>MODULE INTEGRATION EXAMPLE</td>
<td>125</td>
</tr>
<tr>
<td>30</td>
<td>EXAMPLE DYNAMIC INTEGRATION SYSTEM ANALYSIS USAGE</td>
<td>126</td>
</tr>
<tr>
<td>31</td>
<td>EAGER-EVALUATION EXECUTION DIAGRAM</td>
<td>127</td>
</tr>
<tr>
<td>32</td>
<td>EXAMPLE EAGER-EVALUATION ANALYSIS CLASS DECLARATION</td>
<td>127</td>
</tr>
<tr>
<td>33</td>
<td>EXAMPLE EAGER-EVALUATION ANALYSIS CLASS DEFINITION</td>
<td>128</td>
</tr>
<tr>
<td>34</td>
<td>LAZY-EVALUATION EXECUTION DIAGRAM</td>
<td>130</td>
</tr>
<tr>
<td>35</td>
<td>EXAMPLE LAZY-EVALUATION ANALYSIS CLASS DECLARATION</td>
<td>130</td>
</tr>
<tr>
<td>36</td>
<td>EXAMPLE LAZY-EVALUATION ANALYSIS CLASS DEFINITION</td>
<td>131</td>
</tr>
<tr>
<td>37</td>
<td>EXAMPLE EAGER-EVALUATION Legacy ANALYSIS CLASS DECLARATION</td>
<td>134</td>
</tr>
<tr>
<td>38</td>
<td>EXAMPLE EAGER-EVALUATION Legacy ANALYSIS CLASS DEFINITION</td>
<td>134</td>
</tr>
<tr>
<td>39</td>
<td>EXAMPLE notify of change) ALTERNATE IMPLEMENTATION #1</td>
<td>135</td>
</tr>
<tr>
<td>40</td>
<td>EXAMPLE notify of change) ALTERNATE IMPLEMENTATION #2</td>
<td>135</td>
</tr>
<tr>
<td>41</td>
<td>EXAMPLE LAZY-EVALUATION Legacy ANALYSIS CLASS DECLARATION</td>
<td>136</td>
</tr>
<tr>
<td>42</td>
<td>EXAMPLE LAZY-EVALUATION Legacy ANALYSIS CLASS DEFINITION</td>
<td>137</td>
</tr>
<tr>
<td>43</td>
<td>EXAMPLE request validation) ALTERNATE IMPLEMENTATION</td>
<td>139</td>
</tr>
<tr>
<td>44</td>
<td>CYCLIC DEPENDENCY</td>
<td>140</td>
</tr>
</tbody>
</table>
List of Figures

FIGURE 45 - SIMULATED CYCLIC DEPENDENCY ................................................................. 141
FIGURE 46 - EXAMPLE ITERATIVE ANALYSIS CLASS DECLARATION ........................................ 143
FIGURE 47 - EXAMPLE ITERATIVE ANALYSIS CLASS DEFINITION ........................................ 144
FIGURE 48 - SAMPLE STEPPING ALGORITHM ........................................................................ 145
FIGURE 49 - EXAMPLE ITERATIVE ANALYSIS SAMPLE PROGRAM .......................................... 146
FIGURE 50 - EXAMPLE ITERATIVE ANALYSIS SAMPLE PROGRAM OUTPUT ............................ 146
FIGURE 51 - ITERATION FROM THE DYNAMIC INTEGRATION SYSTEM VIEWPOINT .................. 148
FIGURE 52 - EXAMPLE DYNAMIC INTEGRATION SYSTEM ANALYSIS USAGE ......................... 149
FIGURE 53 - EXAMPLE DYNAMIC VARIABLE STATE MONITORING CLASS DECLARATION ....... 151
FIGURE 54 - EXAMPLE DYNAMIC VARIABLE STATE MONITORING CLASS DEFINITION .......... 151
FIGURE 55 - EXAMPLE DYNAMIC ENTITY MONITOR CLASS DECLARATION ........................... 152
FIGURE 56 - EXAMPLE DYNAMIC ENTITY MONITOR MONITORING CLASS DEFINITION ........... 153
FIGURE 57 - EXAMPLE DYNAMIC REGISTRY MONITOR CLASS DECLARATION .................... 154
FIGURE 58 - EXAMPLE DYNAMIC REGISTRY MONITOR MONITORING CLASS DEFINITION .... 154
FIGURE 59 - SAMPLE CONSTRAINT CLASS .......................................................................... 157
FIGURE 60 - BI-DIRECTIONAL CONSTRAINT CLASS DECLARATION ...................................... 158
FIGURE 61 - BI-DIRECTIONAL CONSTRAINT CLASS DEFINITION ........................................ 159
FIGURE 62 - BI-DIRECTIONAL CONSTRAINT SAMPLE PROGRAM .......................................... 160
FIGURE 63 - SAMPLE PROGRAM OUTPUT ........................................................................... 160
FIGURE 64 - DPointData CLASS ......................................................................................... 161
FIGURE 65 - DPoint CLASS DECLARATION ......................................................................... 162
FIGURE 66 - DPoint CLASS CONSTRUCTOR DEFINITION ................................................... 163
FIGURE 67 - DPoint CLASS DEFINITION ............................................................................ 165
FIGURE 68 - SAMPLE CLASS USING DPoint ........................................................................ 167
FIGURE 69 - DPoint SAMPLE PROGRAM ........................................................................... 168
FIGURE 70 - DPoint SAMPLE PROGRAM OUTPUT .............................................................. 169
FIGURE 71 - DEPENDENCY HIERARCHY FORMATION .............................................................. 171
FIGURE 72 - MODULE CONTENT ......................................................................................... 173
FIGURE 73 - APPLICATION STRUCTURE ............................................................................. 174
FIGURE 74 - CURRENT RESPONSIBILITY-DRIVEN DESIGN MODULE SPECIFICATION ......... 176
FIGURE 75 - DYNAMIC MODULE SPECIFICATION DIAGRAM ........................................... 177
FIGURE 76 - EXAMPLE DMS DIAGRAM: L kidnapping EVALUATION ADDER MODULE ....... 178
FIGURE 77 - EXAMPLE DMS DIAGRAM: EAGER-EVALUATION ADDER MODULE ............. 179
FIGURE 78 - DEPENDENCY HIERARCHY VIEWER MODULE ................................................... 182
FIGURE 79 - MONITOR MODULE ......................................................................................... 184
FIGURE 80 - OPTIMIZER MODULE ......................................................................................... 185
FIGURE 81 - GRAPHER MODULE ........................................................................................ 187
FIGURE 82 - DYNAMIC REGISTRY VIEWER MODULE ............................................................ 188
FIGURE 83 - DYNAMIC REGISTRY MONITOR MODULE ......................................................... 189
FIGURE 84 - TYPE-AHEAD MODULE ................................................................................... 189
FIGURE 85 - EXAMINER MODULE ......................................................................................... 190
FIGURE 86 - MASS-SPRING-DAMPER ENVIRONMENT ......................................................... 192
FIGURE 87 - FLOW OF CONTROL FOR MASS-SPRING-DAMPER PROGRAM ....................... 195
FIGURE 88 - TIMER MODULE DYNAMIC MODULE SPECIFICATION .................................... 197
FIGURE 89 - GEOMETRY MODULE DYNAMIC MODULE SPECIFICATION ............................. 197
FIGURE 90 - ADDER MODULE DYNAMIC MODULE SPECIFICATION .................................... 198
FIGURE 91 - TRANSIENT ANALYSIS MODULE DYNAMIC MODULE SPECIFICATION ........ 198
FIGURE 92 - STEADY-STATE ANALYSIS MODULE DYNAMIC MODULE SPECIFICATION .... 198
FIGURE 93 - USER-INTERFACE MODULES DYNAMIC MODULE SPECIFICATIONS .............. 199
FIGURE 94 - UNIX MASS-SPRING-DAMPER APPLICATION ................................................. 200
FIGURE 95 - ANALYSIS CONTROLLER .................................................................................. 202
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>Wing Geometry Analysis DMS</td>
<td>203</td>
</tr>
<tr>
<td>97</td>
<td>Weights Analysis DMS</td>
<td>205</td>
</tr>
<tr>
<td>98</td>
<td>Propulsion Analysis Controller</td>
<td>206</td>
</tr>
<tr>
<td>99</td>
<td>DIS - FORTRAN Integration</td>
<td>207</td>
</tr>
<tr>
<td>100</td>
<td>Cycle Propulsion Analysis DMS</td>
<td>208</td>
</tr>
<tr>
<td>101</td>
<td>Lewis Table-Lookup Propulsion Analysis DMS</td>
<td>209</td>
</tr>
<tr>
<td>102</td>
<td>Lift Surface Analysis DMS</td>
<td>209</td>
</tr>
<tr>
<td>103</td>
<td>Aerodynamic Forces Analysis DMS</td>
<td>210</td>
</tr>
<tr>
<td>104</td>
<td>Drag Buildup Analysis DMS</td>
<td>210</td>
</tr>
<tr>
<td>105</td>
<td>Dynamic Pressure Analysis DMS</td>
<td>211</td>
</tr>
<tr>
<td>106</td>
<td>Mass-Spring-Damper System</td>
<td>240</td>
</tr>
<tr>
<td>107</td>
<td>Block Class Declaration</td>
<td>245</td>
</tr>
<tr>
<td>108</td>
<td>Block Class Definition</td>
<td>245</td>
</tr>
<tr>
<td>109</td>
<td>Cylinder Class Declaration</td>
<td>247</td>
</tr>
<tr>
<td>110</td>
<td>Cylinder Class Definition</td>
<td>247</td>
</tr>
<tr>
<td>111</td>
<td>Solid Class Declaration</td>
<td>248</td>
</tr>
<tr>
<td>112</td>
<td>Solid Class Definition</td>
<td>249</td>
</tr>
<tr>
<td>113</td>
<td>Cylinder Class Declaration Using Inheritance</td>
<td>249</td>
</tr>
<tr>
<td>114</td>
<td>Cylinder Class Definition Using Inheritance</td>
<td>250</td>
</tr>
<tr>
<td>115</td>
<td>Solid Class Declaration Using Virtual Functions</td>
<td>251</td>
</tr>
<tr>
<td>116</td>
<td>Solid Class Definition Using Virtual Functions</td>
<td>252</td>
</tr>
<tr>
<td>117</td>
<td>Cylinder Class Declaration Using Inheritance and Virtual Functions</td>
<td>252</td>
</tr>
<tr>
<td>118</td>
<td>Cylinder Class Definition Using Inheritance and Virtual Functions</td>
<td>253</td>
</tr>
<tr>
<td>119</td>
<td>Block Class Declaration Using Inheritance and Virtual Functions</td>
<td>253</td>
</tr>
<tr>
<td>120</td>
<td>Block Class Declaration Using Inheritance and Virtual Functions</td>
<td>253</td>
</tr>
<tr>
<td>121</td>
<td>Example Class Hierarchy</td>
<td>260</td>
</tr>
<tr>
<td>122</td>
<td>Example Venn Diagram</td>
<td>261</td>
</tr>
<tr>
<td>123</td>
<td>Example Class and Contract</td>
<td>264</td>
</tr>
<tr>
<td>124</td>
<td>Collaborations Graph Inheritance Representation</td>
<td>264</td>
</tr>
<tr>
<td>125</td>
<td>Example Collaboration</td>
<td>265</td>
</tr>
<tr>
<td>126</td>
<td>Example Collaborations Graph</td>
<td>265</td>
</tr>
</tbody>
</table>
This dissertation is organized into the following sections:

- **Introduction** - The general state of engineering software development is described.
- **Motivation** - The ACSYNT project, as the direct inspiration for this research, is described. Problems associated with ACSYNT are also discussed.
- **Problem Statement and Goals** - The problems associated with ACSYNT are first generalized to the engineering community and then specialized. A set of goals for the research addressed by this dissertation is then stated.
- **Influences and Related Literature** - Relevant information is described.
- **Dynamic Integration System** - The Dynamic Integration System is described, including design documents, comparisons to existing systems, and design patterns.
- **A Design Methodology** - Guidelines for creating applications using the Dynamic Integration System are described with methods for documenting such applications.
- **Support Tools** - A set of tools for examining and interacting with codes developed using Dynamic Integration System techniques is described.
- **Example Applications** - Two applications implementing Dynamic Integration System techniques are documented and described.
- **Conclusions** - The dissertation is summarized and conclusions are drawn.
Chapter 1  \hspace{0.1cm} \textit{Introduction}

The development of engineering software has traditionally not been governed by any specific guidelines. New applications are created on an as-needed basis. Writing new code is often avoided, and existing applications are retrofitted to meet new needs as much as possible. This process has been repeated so many times that the core routines in some applications are now over 30 years old.

On the one hand, it is good that software is reused within the engineering community. There is usually no need to rewrite a body of code if the desired functions are already correctly performed. On the other hand, too much reuse is bad because some older codes do not take advantage of new techniques. It is important to properly distinguish between proper reuse and "clinging" to existing code. If excessive work is required to integrate a code into an application, is it really better to reuse the code than to just rewrite it?

The predominant programming language in the engineering community is FORTRAN. Engineers adopted FORTRAN when it was first created because there was no alternative. It was by no means a bad choice and at the time FORTRAN was something to be excited about. Many quality codes were developed and engineers got a lot of mileage out of FORTRAN. Since then, however, new languages have been introduced with more sophisticated features that assist in the management and maintenance of applications throughout the software lifecycle.

It is true that FORTRAN is an excellent symbolic language for performing numerical analyses, but this is where the benefits end. Today’s sophisticated applications often
exceed 1,000,000 lines of code. FORTRAN was not meant for these types of applications. Today we need more structure from languages so that complicated domains such as finite-element analyses, graphical user-interfaces, or CAD geometry management can be adequately modeled and manipulated.

The engineering community has begun to recognize the need for new programming techniques [Ross92a][Ross92b][Keff92][Wald93][Buzz93]. There is a slow but steady shift to new languages as the deficiencies of the older ones become apparent. Companies such as Rogue Wave have appeared to serve the object-oriented engineering community. Support for new technologies is growing, but there are still problems associated with developing engineering software, even when using advanced techniques such as object-oriented programming.

There are several factors hindering the development of engineering software. The first is the mistaken assumption that developing a large computer application is simply \( n \)-times more difficult than developing a small one. It is in reality a much larger problem. Doubling the size of an application can easily quadruple the effort required to create it. This effort can expand, and often does, if it is not properly focused. It is already widely documented that most software projects are usually over-budget, late, and do not meet initial requirements.

One way to curb this problem is to formulate a well-defined plan of attack for implementing an application before starting work on it. This may sound like common sense, but most software developers start a computer project by coding the areas they already understand. The project is completed by building on the code already developed. This is a costly process because there is little chance that the rest of the application will agree with what was already developed. As a project grows, existing code must be
continually modified. By the end of the project, the application is a patch-work of fixes before it has ever been used.

Within the computer science community, this problem is approached by applying a methodology to the project. The methodology instructs the designer on how to structure the application. A set of design documents describing the application for future developers is also produced. The methodology and design documents combine to provide developers with a sense of direction when creating code. A unified game plan among developers results in a well integrated and sound application.

So why do not engineers just adopt an existing methodology and use it? One reason is that most engineers are not even aware that they exist. Another more important reason, however, is that existing methodologies are primarily directed at guiding the computer science community. Existing methodologies are directed at applications such as database development, user-interface design, or office-related software development. The needs of the engineering community are distinctly different. Existing methodologies provide a sound starting point, but must be “tuned” for the engineering community.

This dissertation represents the first attempt to develop a methodology that will guide engineers in the software development process. The work is designed specifically to address the problems exposed by ACSYNT – a conceptual-level aircraft design code. The Responsibility-Driven Design approach for object-oriented programming is extended to incorporate the features of Dynamic Integration. This methodology and the tools developed for it form the basis for future software development at the Virginia Tech CAD lab.
Chapter 2 | Motivation

This chapter introduces the project that was the impetus of this dissertation work: the ACSYNT project. ACSYNT is an aircraft conceptual-design code exhibiting many features common to engineering-type applications including the integration of several analysis modules with a geometry interface. A brief history of ACSYNT is first presented. The modules composing ACSYNT are then described along with the techniques used to integrate them. A discussion of some of the problems associated with the current integration schemes are next presented. The initial approach to solving these problems, the object-oriented approach, is discussed with reasons why it alone is not the solution. Finally, a simplified description of the solution taken by this dissertation is introduced.

2.1 ACSYNT History

ACSYNT (Aircraft Synthesis) is a feature-based, parametric, computer-aided aircraft conceptual design code [ACSY93][Jaya92][Mykl94]. The program is based on an analysis code developed at NASA in the 1970’s [Greg73][Vand76]. In 1986 Virginia Tech first entered the picture by providing a graphical front-end to the analysis that enabled the user to interactively prepare input for the analysis and interpret output [Wamp88a][Wamp88b][Mykl88]. Since then many enhancements have occurred ranging from work on the CAD system [Steu93][Schr91][Rive93][Tayl88] to work on the geometry structure [Jone91][Marc91][Jaya91][Kell93][Hasa93] to work on the analysis [Mala89][Mala90][Arle93][Grie88][Grie89][Warr93][Squi92] to work on integrating new modules and frameworks [Woya92][Woya93][Uhor93][Mykl93].
2.2 The Current Structure of ACSYNT and its Associated Problems

ACSYNT is currently composed of many C, FORTRAN, and C++ modules. The core ACSYNT modules are:

- Geometry
- Aerodynamics
- Propulsion
- Weights
- Trajectory
- Takeoff and Landing

Along with the core modules, the following modules have since been integrated:

- Economics
- Stability and Control
- Navy requirements
- Powered-Lift effects
- Sonic Boom
- Quick IR

The core modules of ACSYNT are integrated using COPES (Control Program for Engineering Synthesis) [Vand76]. The CONMIN (Constrained Minimization) [Vand73] program is utilized from within COPES to perform optimization based on the method of Feasible Directions [Vand84]. Other modules, such as the Quick IR module, are integrated directly through the CAD system. A simplified diagram showing the structure of ACSYNT is shown in Fig. 1.
Figure 1 - ACSYNT Structure

Some modules, such as the Quick IR module, are integrated with ACSYNT by exchanging data files. For other modules the process is more complicated. A simple run-through of an optimization cycle demonstrates the process:

1. The ACSYNT CAD System code first requests that COPES optimize some parameter or set of parameters. These parameters are identified as locations in a global array.

2. COPES, using control routines provided by the ACSYNT CAD System, first initializes the global space. This is performed by reading in a data file created by the ACSYNT CAD System code describing the aircraft and other parameters.

3. COPES next initiates a design iteration. This involves using the ACSYNT control routines to sequentially execute the analysis modules. The analysis modules read and write parameters to and from the global space. Once the analysis modules have been executed, COPES examines the parameters in the global space. If they are not acceptable, another design iteration is performed. If the parameters are acceptable, a data file describing the optimized structure is created and control is passed back to the ACSYNT CAD System which may read in and interpret the data files.

Chapter 2 - Motivation
Each analysis module does not actually utilize the common global space directly. Instead, in an attempt to modularize the design, each analysis module operates using its own global space. This global space is used to integrate individual routines within each analysis. This structure is shown in Fig. 2. The local global parameters and common global parameters are translated back and forth at the beginning and end of execution of each analysis module. This effort alone has been estimated to be as much as 40% of the total analysis time.

![Diagram of ACSYNT Global Data Management](image)

**Figure 2 - ACSYNT Global Data Management**

As shown, ACSYNT primarily uses two forms of integration:

- **data files.** To perform an analysis, ACSYNT first prepares a data file and then executes the analysis. The analysis reads the data file, performs calculations, and writes another data file. Control then passes back to ACSYNT which reads the output data file from the analysis and interprets the results.

- **global data.** Analysis modules freely read and write parameters to and from global space. The global space acts as a giant dictionary of design variables from which modules may connect.
There are several deficiencies in ACSYNT’s approach to integration. The first problem is performance. Execution speed is affected by two factors:

- **overhead associated with data files.** The time associated with the preparation, creation, reading, and interpretation of a data file is very costly with respect to a simple function call. A direct memory transfer between modules through function calls is much more efficient.

- **translation of global spaces.** The execution of each analysis module requires that parameters stored in the common global space be translated to a local global space. Upon completion of the analysis, parameters must be translated back from the local space to the global space. This pre- and post-processing time can be costly.

The second problem associated with ACSYNT’s integration schemes is the limited amount of interaction available. Each module is integrated in an all-or-nothing fashion. Executing an integrated module involves executing the complete module when perhaps only part of it is required. The ideal form of integration would permit modules to be executed in a more fine-grained manner. Redundant calculations performed by executing an entire module instead of a subset only add to overhead.

It is apparent from the current structure of ACSYNT that better forms of integration need to be defined. To date, more effort has been applied to working around problems associated with integration rather than creating new methods that will avoid the problems in the future. For examples, see [McC94][Hwan94][Hale94][West94]. This dissertation takes the first step in creating new design methods for engineering software.

### 2.3 Remodeling ACSYNT

In an attempt to remedy some of ACSYNT’s problems, a completely new system was proposed. The initial strategy was to use the object-oriented approach. The idea was that an airplane is an object. Performing an analysis on the airplane would involve flying it
through a mission, collecting data, and refining the design. The process would continue until an acceptable solution was found.

In terms of objects, an airplane could be defined as some sort of assembly. An assembly consists of a model and its related components. Each component could also have subcomponents. An aircraft would therefore be defined as a subclass of a model class with the wings, fuselage, tail, etc., as subclasses of a component class.

Flying the aircraft would be simulated by specifying an altitude, Mach number, and other environmental conditions. Key parameters such as lift, drag, or weight would then be retrieved by querying the aircraft object. The aircraft would answer queries by collecting information from each of its components.

The object-oriented paradigm appears to provide an elegant solution; however, a closer look at the design reveals problems. The natural division of an aircraft using object-oriented design concepts is along geometric borders, that is the wing, the fuselage, etc. This breakdown assumes that components may be packaged as separate entities. This assumption is incorrect. Problems associated with modeling an aircraft with components include but are not limited to:

- **how to account for analyses that cross component boundaries** - Some analyses such as the weight analysis are easy to implement - the weight of an aircraft is simply the sum of the weights of its components. Other analyses such as calculating the drag are not so simple. The drag on an airplane is not simply the sum of the drag of each component. The drag on the tail of an airplane depends on what is upstream, e.g., the wing and fuselage. The calculation of the drag on the tail is therefore not self-contained. The only component that knows what other components are upstream is the aircraft itself. Moving the drag calculation to the aircraft would violate the encapsulation of the tail component.
• **how to make design modular** - The idea of letting vendors supply specific components from which an airplane can be built sounds good, but is not practicable. The process of analyzing an aircraft has traditionally been broken down by discipline (for example weights analysis, aerodynamic analysis, cost analysis, etc.). To break down an aircraft in terms of objects requires that each object know how to analyze itself in terms of each discipline. This is required so that when an aircraft is assembled a complete analysis can be performed. If one component cannot perform an analysis from a discipline, then that discipline cannot be applied to the airplane as a whole. Requiring each component to be able to perform analyses from each discipline is an unreasonable goal and a managerial and validation nightmare. Therefore, it is best to breakdown the analysis by discipline.

• **how to not constrain users by our system** - In the object-oriented approach, the natural method for extension or customization of a system is through inheritance and polymorphism - inherit from classes in a framework and provide the extra functionality required. A logical way to do this in an analysis is to provide virtual functions that perform some calculation. These virtual functions can be implemented with any specific calculation the user wants. The problem with inheritance is that new classes must conform to the existing hierarchy structure. This often involves viewing (and perhaps modifying) source code which is undesirable [Taen89]. In addition, most existing analyses are procedural and would probably require considerable modification to fit into a specific object-oriented framework.

After initial testing, it became clear that while the object-oriented approach was well suited to modeling the physical structure of an aircraft, it did not sufficiently capture the details of certain analysis approaches. It was clear that an object-oriented structure for ACSYNT would prohibit extensibility and discourage reuse. A better suited approach would be to divide ACSYNT into modules similar to the way it is now. Each module, if desired, could model an aircraft structure in a way that was best suited for it. The
geometry side could use an object-oriented approach. An analysis module may prefer a simpler hierarchical procedural structure. Utilizing a different structure in each module introduces extra overhead, but increases encapsulation at the module level improving the plug-n-play possibilities. It may even be possible to use existing modules as part of the new design.

This modular structure is the foundation for the design techniques developed in this dissertation. The key to building engineering software in this format is to provide a means for communication and consistency management between modules. The solution to this problem is a primary focus of this dissertation - *Dynamic Integration*.

### 2.4 Summary

This chapter has introduced the ACSYNT aircraft conceptual design program and its problems as the motivation for this Dissertation. A brief history of ACSYNT was presented followed by a discussion of the current structure of ACSYNT. The problems of that structure were discussed along with several reasons why just applying object-oriented design to the project does not help. This leads into the next chapter - a more specific discussion of the problems associated with engineering software including an introduction to the methods used to address these problems.
Chapter 3 | Problem Statement and Goals

This chapter extracts from the previous chapter a more concrete definition of the problems associated with developing engineering software. A high-level problem statement is first presented. This statement approaches software development from an abstract point of view. In an attempt to point out specific issues causing these high-level problems, a specific, low-level point of view is then presented. The approaches pursued to address these problems are then briefly described. Finally, a set of goals is presented.

3.1 Problem Statement

Stated simply, today’s engineering software is too difficult to create, use and maintain. Reuse does occur within the engineering community\(^1\), but not without great effort. New techniques need to be developed that guide the engineer in developing reusable software.

From the high-level point of view, the goal of this dissertation is to provide a method for developing engineering software that is:

- easier to create
- easier to use
- easier to reuse
- easier to maintain

\(^1\) Software reuse is probably the number one research topic in the computer science community. Engineers, for various reasons, put a premium on reusing existing code. The effort put into integrating some older codes, however, does not always justify the benefits received by reusing the code.
Making the software easier to use involves providing better methods to interact with the software. Making software easier to reuse involves making the software "integration aware" before it ever needs to be integrated. This implies providing built-in mechanisms that manage the interactions between modules. The following section examines these issues from a more specific, implementation point of view.

3.2 Detailed Problem Statement

This section pinpoints some of the problems associated with trying to create an engineering analysis code. Typically, the development process for an analysis code focuses on choosing proper algorithms and encoding them efficiently. While this is very important, there are other issues that are often overlooked.

An analysis code, no matter how significant the calculations it performs, is not very useful if it cannot be used within the framework of the application it is to be integrated. There are several factors that affect this integration. As noted by Ramamoorthy, research into the following integration factors in particular is needed [Rama92]:

1. a common data representation
2. compatible interface structures
3. consistency of logically inter-linked data

As discussed with respect to ACSYNT in Chapter 2, representing a problem domain distributed among several modules using a common structure is almost an impossibility².

---

² This of course falls under the assumption that modules are being developed by different factions with minimal communication between them. If an application is developed completely in-house, then it is entirely possible that an application can share the same structures in each of its modules.
The approach taken in this dissertation is to allow each module to model the domain in whatever manner it desires. Each module is treated as independently as possible in hopes of decreasing coupling and increasing reuse capabilities. By permitting dissimilar data representations, the issue becomes how to integrate modules. This deals with the second and third factors pointed out by Ramamoorthy. These two problems can be redefined as: 1) how to put an interface onto an analysis, and 2) how to manage the integration of multiple analyses. The research performed in this dissertation will focus on these two issues in particular.

For purposes of demonstration, an example analysis will be used throughout the remainder of this chapter and dissertation. The analysis described is based on a simple mass-spring-damper system such as the one shown in Fig. 3. The mathematics of the system is described in Appendix A.

![Mass-Spring-Damper System](image)

**Figure 3 - Mass-Spring-Damper System**

Issues regarding how to create an analysis describing such a system will be discussed in terms of current techniques and new techniques proposed here.
3.2.1 Providing Interfaces to Analyses

The first problem presented to the analysis programmer is how to define an interface to the system. The interface is the interaction mechanism with the analysis, i.e. the interface the analysis presents to the rest of the application. The interface to an analysis is usually composed of one of or a combination of the following methods [Penn91]:

- **data files** - execution of the analysis involves only one function. Calling this function results in the reading and writing of two data files. First a data file of input values is read. After performing the analysis, results are then written to another data file.

- **function procedures** - execution of the analysis involves calling one or more functions. Each function provides a set of input and output arguments.

- **global data** - execution of the analysis involves calling one or more functions. Each of these functions reads input values from and writes output values to the global space.

- **centralized database** - execution of the analysis involves calling one or more functions. Each of these functions reads input values from and writes output values to a centralized database.

The following sections will discuss the advantages and disadvantages of each of the described methods.

3.2.1.1 Data Files

While rather primitive and inefficient, using data files as a mechanism for information exchange is in fact quite common. As noted by Khedro et al. [Khed93], "...most current types of software have limited ability to communicate their data outside their boundaries by simply reading from or writing to data files in specified formats.” One reason is that
most older analyses are often formatted to work this way. Many analysis programs were originally designed to run in a batch-mode format. Modernization of the analysis usually only involved providing a way to call the analysis from within a program. Rather than passing arguments to the analysis, the data file remains.

This method has two primary drawbacks:

- inefficiency in terms of speed
- inefficiency in terms of interaction.

The primary advantage of this method is:

- good encapsulation

This method is, as expected, extremely inefficient in terms of speed and interaction. The overhead associated with reading and writing to a data file is large in comparison to passing arguments to functions. In terms of efficiency, using data files for input and output is only appropriate for analyses that take a long time to execute or for analyses that are designed to be run in a batch-mode format.

The second, and perhaps more prohibitive drawback to using data files as a method of information exchange is the limited potential for interaction. To obtain results from the analysis, a complete execution of the analysis must be performed. An input data file must be created and formatted, the analysis must be executed, and the resulting output data file must be interpreted. This is a burdensome process.

One advantage to using data files is that the analysis is well encapsulated. Inputs and outputs are clearly defined and there is no need to understand the execution of the
module. The code remains clearly separated from the rest of an application. The advantage to this is that the analysis can be modified or replaced with minimal effects to the rest of the application. The only difficulty associated with replacing an analysis is to rewrite the translators that read and write the output and input files. This task non-trivial.

3.2.1.2 Function Procedures

The next step up from using data files to transfer information to and from an analysis module is to pass the information directly through the calling procedures. This eliminates the need to write data file I/O routines and greatly improves efficiency.

The drawbacks of this technique are:

- potentially limited interaction
- multiple functions must be managed

Some advantages include:

- execution efficiency
- ease of creation

The first problem associated with modeling an analysis with a set of procedures is choosing the procedures to create. Consider the possibilities for the mass-spring-damper system. A complete interface involving every parameter can be created:

\[ \text{evaluate}(k, c, m, x_0, \dot{x}_0, t, F_0, \omega, x_{\text{total}}, x_{\text{transient}}, x_{\text{steady-state}}, X, \phi, \omega_n, \omega_d, \zeta) \]
This function accepts all inputs and returns all outputs. Inefficiencies are easily seen if the only output needed is $x_{\text{transient}}$. In this case, wasted effort is put into calculating $x_{\text{total}}$ and $x_{\text{steady-state}}$. Another problem occurs when an output that is calculated from a subset of the inputs is required. An example of this is $\omega_d$ which is not dependent on $x_0$, $\dot{x}_0$, $t$, or $F_0$. Is it okay to enter 0 for the inputs we do not care about? This may work, but it may also crash the program.

An alternate interface method is to provide a set of functions for calculating each of the outputs. Consider the following two examples for calculating $x_{\text{total}}$ and $\zeta$:

\begin{verbatim}
   evaluate1( k, c, m, x_0, \dot{x}_0, t, F_0, \omega, x_{\text{total}} )
   evaluate2( k, c, m, \zeta )
\end{verbatim}

Each function only uses the inputs required to calculate some specific output. The problem with this method is the redundant intermediate calculations performed, not to mention the burden on the analysis programmer for providing such a vast interface to the analysis. Redundant calculations occur when an output is retrieved which is dependent on another output. For example, the calculation of $x_{\text{total}}$ implies the calculation of $x_{\text{transient}}$ and $x_{\text{steady-state}}$. If in the use of the analysis both $x_{\text{total}}$ and $x_{\text{transient}}$ are used, $x_{\text{transient}}$ will be calculated twice: once internally for the calculation of $x_{\text{total}}$ and once for the requested calculation.

The application programmer is at the mercy of the decisions made by the analysis programmer. The interface to the analysis will only be effective and efficient if it provides exactly what is needed.
Assuming that a set of functions can be specified that provides the ideal interface, an additional problem still exists: the management of data and the coordination of routines. Data management involves knowing which routines need to be executed in order to form an output. When a single input is changed, which outputs need to be re-evaluated? Does re-evaluating these outputs affect any other outputs, i.e. are there any secondary affects associated with the change to the original input? The coordination of the execution of these routines is a very difficult task. The problem grows exponentially with the number of relations involved.

An advantage to using procedures for exchanging information is that the desired information can be precisely and efficiently specified using the arguments of the procedures. Languages that do not provide data structures, such as FORTRAN, are at a slight disadvantage in this respect since all information must be represented using the standard data types, i.e., ints, doubles, etc. This can be burdensome when large amounts of information must be exchanged.

The final advantage to this method is that it is relatively easy to implement. Most programmers learn programming using procedures so this method is usually just a continuation of past skill. This method is also easy because, as opposed to the data file method, no translators need to be written to read or write data files. The exact data needed is specified in the calling arguments of procedures. The difficulty lies in choosing which procedures to provide.

3.2.1.3 Global Data

Global data is a common method of storing design information. Large global arrays are often used as a database of design information from which all modules have equal access.
The drawbacks to using global data include:

- limited interaction
- data management
- poor encapsulation

The primary advantage is:

- quick and easy

Using global data is just a specialized form of using procedures. The parameters previously passed to the procedures now just occupy global space. The problems associated with using procedures to model analysis modules also apply to the use of global data. The difference now is that it is nearly impossible to track what is happening in the analysis - what variables are being used and which ones are being modified?

The result of not knowing where and which parameters are being modified or used is that code is almost impossible to reuse or enhance. What are the consequences of removing a particular routine from the analysis? How will the system be affected if a new routine is added? Is it okay to modify this parameter, or is another routine expecting to use its value? The result of using lots of global variables is usually the same - an untraceable code that no one will touch but the person who originally created it.

Global data often represents the sign of quick fixes. Under time constraints, the global data approach is often appealing. Changes can be made rapidly without having to significantly restructure an application. The allure of global data should be avoided because in the long run more time and effort will be devoted to maintenance of global variables than would have been devoted to properly correcting the initial error.
3.2.1.4 Centralized Database

A centralized database describes a design where each module stores information in a common database. The database is used as a sophisticated mechanism for storing global data. Modules may access any variable from anywhere. The key difference between a database and global data is that restrictions can be placed on the database. Variables can be marked as read-only, dependencies may be built between entries, etc. When using a database, all attempts to access or set the value of an entry must pass through the database. When using global data, however, there are no restrictions on what happens and thus, no management.

The disadvantages of a centralized database include:

- finding or creating a database
- extra overhead associated with the database

The primary advantage is:

- improved data management

Most existing databases are designed as libraries. Mechanisms are provided for searching the database for entries. If a database is to be used as an interactive mechanism for an analysis module, then the database needs to be specifically optimized for that purpose. This usually means creating rather than purchasing one.

No matter how efficient the database, there is always extra overhead associated with retrieving an element. The goal is to find a trade-off that outweighs this overhead such as increased reliability or increased efficiency of an analysis.
The advantage to using a database is that there is a lot more control over the application. This is especially true if the database is custom built. It is possible to know when and where a particular parameter is modified. It is also possible to implement restrictions on what parameters may be accessed or modified. As explored later, the approach taken by this dissertation is to develop a highly specialized database for the purpose of integrating engineering analyses.

3.2.2 Managing Integration Problems

Once an interface is created for an analysis, the problems associated with integrating multiple analyses begin to appear. The two primary obstacles to integrating engineering-type analysis codes are:

- **Maintaining Consistency Among Modules** - An analysis module can be defined as having a set of inputs and outputs. When several analyses are integrated, outputs of some modules become inputs to others. Maintaining consistency between all inputs and outputs is a complicated process, especially when secondary effects are taken into account.

- **Responding to Changes** - Often modules are designed to respond to changes within an application. Consider the sizing of an aircraft engine. When the engine changes, the rest of the aircraft should change to accommodate. Calling the appropriate routines to respond to this change is a difficult task.

The following two sections further describe the problems associated with integrating engineering codes.
3.2.2.1 Maintaining Consistency Among Modules

Consistency management involves maintaining a valid set of parameters in an analysis module. Any time an input to an analysis is modified, the output of the analysis must be considered invalid. When several analyses are interconnected, it is difficult to know what outputs are valid with respect to what inputs. Managing consistency varies in complexity depending on what type of interface is provided to the analysis:

- **data files** - When data files are used as interfaces to analyses, parameters are evaluated in large quantities. When one input is changed, the entire analysis is re-run to formulate new outputs. This results in extreme inefficiencies, but tends to be reliable.

- **function procedures** - Management difficulties for analyses implemented using function procedures are proportional to the number of procedures provided in the analysis interface. If few functions are provided, then management is relatively easy, but efficiency is poor. If many functions are provided, management is very difficult, but payoffs in efficiently improvement can be worthwhile.

- **global data** - Integration of analyses based on global data is nearly impossible³ unless the analyses are well documented. Problems arise from the fact that it is unclear what global variables are input and output parameter to which analyses. It is also unclear which outputs are modified (indicating the need to re-run another analysis module) when an analysis is executed. This results in the need to re-run all analyses each time new results are needed.

---

³ That is unless you take a large-grained approach like ACSYNT. ACSYNT translates the entire contents of the global space used by an analysis back and forth between its global space before and after executing the analysis. Even this method, however, requires a great deal of “expert” knowledge of parameter locations in the global space. The alternative, a fine-grained approach, is nearly impossible using global data.
• **centralized database** - Using a database, it is possible to monitor which parameters are modified, when they are modified, and where they are modified. This benefit is not received by off-the-shelf databases, but custom-built databases may be created for this specific purpose. Such a database can be made to manage dependencies among inputs and outputs of analysis routines.

Using any one of the above techniques, consistency management can be performed in several manners. These manners are briefly described below:

• **eager-evaluation** - Using this technique, outputs of an analysis are formulated whenever inputs change. This is inefficient because analyses tend to be executed multiple times before outputs are actually required. The technique is, however, easy to implement.

• **lazy-evaluation** - Using this technique, an analysis is not executed until an invalid output is needed. This is very efficient, but requires some sort of data management scheme to know when an analysis needs to be executed.

• **ad-hoc evaluation** - Using this technique, an analysis is executed whenever its outputs are used regardless of whether the analysis needs to be re-executed or not. This technique maintains consistency, but often results in analyses needlessly being executed when their outputs are already valid.

A large application usually involves some combination of the three management methods. Eager and lazy-evaluation techniques are often used within a single analysis module while ad-hoc evaluation is used for connecting multiple analyses.

The optimal method would be to use lazy-evaluation throughout the entire application. Unfortunately this is a complex task. As a result, most applications fall back on ad-hoc evaluation techniques.
3.2.2.2 Responding to Changes

Many problems in engineering codes can be traced to the problems that occur when a parameter is unexpectedly changed. This can be as simple as a geometric display not reflecting correct parameters or as disastrous as using the incorrect outputs of an analysis.

Often when parameters change, the technique used to respond to that change is to call another function. This technique is easy to manage when a module is small and self-contained, but often leads to problems when the code expands or is integrated with other modules. The problem is that the same “response” function must be called from each new place where the parameter is modified. This quickly becomes a maintenance nightmare.

3.3 Goals

As part of this dissertation, the following will be created, documented and discussed:

- **A framework for implementing engineering-type code.** A framework that engineers can use to build analyses and supporting code for the analyses will be created.

- **An object-oriented design methodology for using these tools.** A framework without a methodology describing how to use it is not very useful. A design methodology will be created that suggests design paths that should be followed to use the framework effectively. The goal of this dissertation is not to implement another object-oriented design methodology, as that field is already rather crowded, but to supplement an existing methodology.

- **Support tools for assisting in interacting with and using the framework.** A set of tools, constructed using the techniques described in this dissertation, designed to assist in the rapid prototyping, creation, and debugging of engineering codes, will be
created. These will include tools for interacting with analyses, for monitoring the execution of analyses and for viewing relationships between parameters in analyses.

- **Sample applications.** As verification of the methodologies and techniques described in this dissertation, several sample applications will be implemented featuring the techniques. These applications will try to cover a diverse range of needs, including legacy FORTRAN code compatibility and plug-n-play capabilities.

### 3.4 Approach

The approach taken to solve the problems discussed will be a highly specialized database\(^4\). This database can perhaps more appropriately be called a registry. The registry will utilize several features to optimize it for engineering-type problems. These features include:

- **optimized storage mechanisms.** The registry will only store simple parameters. Issues such as persistent data management or composite objects will not be addressed. The goal of the registry is to be extremely lightweight, providing only the minimal number of features necessary. Performance is optimized for speed so that interaction with the registry will not hinder analysis performance.

- **simplified access mechanisms.** A goal of this dissertation is to provide mechanisms familiar to engineers so that complicated interactions with the registry will not be required. Using C++, this can very easily be accomplished using overloaded operators. As will be shown later, users will read from and write to the registry without even knowing of its existence.

---

\(^4\) I hesitantly use the word database because the application that will be developed is so specialized that it only barely resembles a database. I use the word database because the system does store parameters that may be accessed from different modules. The mechanisms for storage and access are, however, very nontraditional.
• **dependencies between parameters may be specified.** The power of using a registry for storing parameters is that a centralized location exists where all parameters are accessed. This permits relationships to be built and maintained between parameters from one location. The registry created will support dependency hierarchies to be constructed from parameters and the relationships between them.

• **event-trigger pairs will be utilized.** As discussed, the ability to respond to a change to a parameter is an essential ability. The registry created for this dissertation will support event-trigger pairs. In response to the triggering of an event (such as changing the value of a parameter), functions registered by the user will be called.

The registry mechanisms will be implemented in what will be called the *Dynamic Integration System*. Further discussion of the Dynamic Integration System, its implementation and usage is presented in Chapter 5.

### 3.5 Summary

This chapter has taken the problems described in the last chapter relating to ACSYNT and concisely expressed them in terms of engineering software as a whole. The problems were first expressed from a high-level point of view. This point of view singled out the following problems:

• engineering software is difficult to create
• engineering software is difficult to use
• engineering software is difficult to reuse
• engineering software is difficult to maintain

The approach taken by this dissertation is not to directly solve these problems, but rather to develop software development tools that minimize their effects. This involves
examining engineering software problems from a closer point of view. Specifically, the following difficulties in developing engineering software can be identified:

- providing interfaces to analyses
- maintaining consistency between analyses

The approach taken in this dissertation will be to develop a registry technique that employs the following features:

- optimized storage mechanisms.
- simplified access mechanisms.
- dependencies between parameters may be specified.
- event-trigger pairs will be utilized.

Implementing the registry will only be a part of this dissertation. The complete set of goals includes:

- A framework for implementing engineering-type code, i.e. analyses, optimizers, etc.
- An object-oriented design methodology for using these tools.
- Support tools for assisting in interacting with and using the framework.
- Sample applications demonstrating the application of the above items.

The following chapter discusses some current research related to the problems discussed in this chapter. Following that chapter, the details of the solutions to these problems are presented.
Chapter 4  
Influences and Related Literature

This chapter presents research activities that are pertinent to this dissertation. The chapter focuses primarily on the following issues:

- **Object-Oriented design**: The concepts of object-oriented programming will be introduced along with a discussion of its benefits. Examples of the application of object-oriented principles to engineering research will also be presented.

- **Object-Oriented Methodologies**: The definition of a methodology as well as a summary of some of the current methodologies will be discussed. Reasons that these methodologies do not meet the needs of engineers are also presented.

- **Software Reuse**: One of the motivating factors for this dissertation is to create engineering software that can be easily integrated. This is part of a larger problem: software reuse. Research into improving software reuse will be discussed.

- **Integration**: As mentioned, improved integration capability is one of the motivational factors for this dissertation. Existing integration problems and techniques are surveyed.

In addition to the above topics, several miscellaneous research projects will be discussed. These projects, while applicable to this dissertation, do not fall under any of the above categories. In later chapters, when implementation details are discussed, more direct comparisons to existing research projects will be explored.
4.1 Object-Oriented Design

This section provides a review of object-oriented design principles. The characteristics of object-oriented design are first presented. A brief discussion of the advantages of using an object-oriented approach is then presented. Finally, a survey of existing object-oriented engineering applications/research is presented.

4.1.1 What is Object-Oriented Design?

Object-oriented design is another attempt to raise the level of abstraction of a programming language. A condensed history of programming styles can be summarized as follows:

- assembly languages
- procedural languages
- object-oriented languages

These are generally referred to as first, second, and third-generation languages. Each progression to a new language raises the level of abstraction. Procedural languages remove the burden of juggling bits and bytes from the assembly programmer. Object-oriented languages, likewise remove the burden of managing floating point and integer variables from the procedural programmer by introducing classes and objects.

The primary components of object-oriented design are:

- **classes and objects**: A class represents the encapsulation of some data and the corresponding functions used to manipulate the data. By succinctly grouping the two, a single unit of behavior, the object, is formed. An object-oriented application is composed by instantiating objects from a set of classes. These objects work together to fulfill some purpose.
• **inheritance:** A powerful form of reuse in object-oriented design is the ability to inherit one class from another. This form of specialization allows a designer to focus on what is new to his/her design rather than worrying about maintaining backward compatibility by using repetitive code.

• **polymorphism:** Polymorphism is the ability of a group of dissimilar objects to conform to a common interface. This mechanism permits an application to interact with a generically defined object. New classes conforming to the polymorphic interface may later be created without interfering with existing code.

For readers unfamiliar with object-oriented design, a more detailed review (along with C++ examples) is presented in Appendix B.

### 4.1.2 Why Use the Object-Oriented Approach

Object-oriented design is making headway into the engineering community [Ross92a][Ross92b][Keff92][Wald93][Buzz93]. While most of its benefits have not yet been fully recognized, progress is being made. As concluded by Frank McGarry (leader of the Software Engineering Labs) during the NASA-Goddard Software Engineering Workshop, object-oriented techniques may be the most influential methodology studied by the Software Engineering Labs to date [Glas93].

- The engineering community was first widely exposed to programming with the arrival of FORTRAN. Since then, new languages such as C, C++, Smalltalk, Ada, and others have appeared, yet FORTRAN has remained the dominant language. As noted by Walther and Peskin [Walt89], “Scientific computing, notably numerical modeling and simulation, is anchored in a thirty year old software tradition. It is now documented that this computing methodology is hindering computation as a tool for experimentation and discovery.” There are several basic reasons why object-oriented programming can assist the engineer:
• **analysis can benefit from custom types.** All but simple analyses manipulate more sophisticated data types than are provided by procedural languages. Examples of these data types include complex numbers, points, vectors, and matrices. Procedural languages must simulate these types with arrays, floats, and other primitive data types. Object-oriented languages can directly represent these types with the class mechanism. Even structured procedural languages such as C are inadequate due to the lack of an operator overloading mechanism such as the one found in C++.

• **data management requires organization.** One reason that computers are used to perform analyses is that they can execute them many times faster than a human. The benefit of this is that a large number of calculations can be performed where only a few could in the past. A mechanism is required for managing the results of these analyses. Object-oriented programming presents techniques for organizing and manipulating data that procedural, scientific languages do not.

• **A complete application consists of more than just the analysis.** When creating a full-fledged application, the analysis becomes only one component. Other components include the user-interface, data storage, etc. The integration of these components into a single application is well suited to an object-oriented approach.

In addition to these reasons, empirical studies have shown numerous advantages to using and object-oriented approach [Henr93][Lewi91].

### 4.1.3 Example Object-Oriented Applications

The object-oriented paradigm is just beginning to work its way into the engineering software development process. Examples are rare. Great potential does exist however, as exhibited by the projects summarized in this section.

Object-oriented design has probably received the most attention in the development of new finite-element codes. This is probably due to the need for management of large,
complex and varied data sets. Yu and Adeli have developed a general framework for creating finite-element codes [Yu93]. This framework includes classes for creating vectors, matrices, nodes, and elements among other things.

Yu and Adeli acknowledge the need for new design methods and the application of object-oriented approaches by saying, “With the increase in the size and complexity of finite element software systems the traditional waterfall algorithm-driven structured programming approach and the FORTRAN language are no longer adequate and new software architecture and programming paradigms need to be created.”

The topic of object-oriented finite-element software is also discussed by Vermeulen [Verm93]. Classes are described for vectors, matrices, and nonlinear equations. A framework is created where modules interact with a general function class. Specialization occurs by inheriting from this function class. Custom function classes may be created that are, using polymorphism, compatible with the existing modules.

Another finite-element example is implemented in the MODIFY (modeling tool for integrated finite-element analysis) project [Ohts93]. This project provides automatic mesh generation for finite-element models. Object-orientation permits simplification of complex models.

Remy et al. have also created an object-oriented finite-element system [Remy92]. Their emphasis was on creating flexible, reusable code that could be run on a variety of platforms. Remy notes that, in reference to procedural scientific programming languages, “the programming technology used hasn’t changed much since the appearance of scientific computing.” The following drawbacks to traditional programming techniques are recognized:
• The work invested in programming during the course of a scientific project is usually lost after its completion.

• Most numerical experiments described as the results of complex codes are unrepeatable, insofar as the underlying codes are badly documented, unshareable and of enormous complexity.

• The task of porting existing codes to new hardware architectures (e.g., parallel or vector computers) and/or modern software environments (e.g., windowing systems) is tedious, time-consuming and often produces poor results.

In fields other than finite-element analysis, Gabriele and Serrano has created an object-oriented tool for analyzing and designing single-stage gear boxes [Gabr91]. Object-oriented techniques were required to increase the flexibility of the design process. The authors state that, in reference to procedural languages such as FORTRAN, “the use of these languages can lead to difficulties when trying to capture the recursive or iterative nature of the design task, such as the need to sometimes start at different points, or to go back to previous steps of the design at any given time, or the need to handle a wide variety of design information that is often not easily captured in a typical computer program.”

Scott et al. have created a set of object-oriented classes for creating electronic design tools [Scot88]. The goal for the environment was to provide a common, reusable and extensible foundation for tools created at Data General. These tools take care of implementation details such as efficient storage and retrieval of design objects into/from a filing system.
Some other examples of object-oriented engineering applications include:

- a system for modeling control systems using C++ [Brüc88]
- a three-dimensional graphics system for kinematic animation with constraints [Serr90]
- a digital circuit CAD system [Char91]
- simulation of radiation therapy treatments for cancer [Jack86]
- optimized design of satellite structures [Kodi94]
- a computer-integrated manufacturing simulation environment [Newt91][Olem90]
- a scientific visualization system and tools [Schr92] [Turn90]
- multi-body simulation with symbolic evaluation [Sree89]
- a numerical modeling and analysis environment for chemists [Piel91]
- three-dimensional graphics and modeling [Egbe92a][Egbe92b][Thom92][Bahr92]
- animation systems [Bree87][Mahi90][Maio90]
- real-time applications [Ishi92a][Ishi92b]
- fluid mechanics [Angu89]

4.1.4 Object-Oriented Numerical Research

Most numerical analyses are procedural. The application of object-oriented techniques is just now being explored. Many people have recognized the benefits of modeling vectors and matrices using classes. Books have even been published on this topic alone [Buzz93]. This section briefly examines some additional research into modeling mathematical concepts using object-oriented techniques.

Budd describes an attempt to model types with classes [Budd91]. All mathematical quantities are of type `Number`. Using this technique, dividing an `Integer` object by another `Integer` object results in a `Fraction` object, but dividing a `Float` object by another `Number` object results in a `Float` object. The framework created permits the
implementation of types such as polynomials that allow mathematicians to naturally express relations between dissimilar quantities.

Jerrell has incorporated the process of automatic differentiation into C++ classes [Jerr89][Jerr90]. The author states that current methods for taking derivatives (numerical estimation and hard-coded manual evaluation) are error-prone and can be inaccurate. Using overloaded operators, Jerrell incorporates automatic differentiation into a class structure, permitting the natural expression and evaluation of derivatives.

McDonald has created the Cactus system, a system for numerical linear algebra and constrained optimization, implemented in CLOS [McDo89]. The system was implemented in an object-oriented language because existing procedural languages were not satisfactory. As the author states, “Linpack is excellent FORTRAN, but because it is FORTRAN, the representations and the rules for their use are not expressed in the code. An important consequence is that it is difficult to extend or modify Linpack routines to do something slightly different from what the original author intended.” According to the author, the advantages of using an object-oriented language for numerical representation include, “...modular handling of multiple representations for linear transformations and vectors,” and, “...that it gives concrete representations to concepts that are implicit in traditional FORTRAN libraries, e.g. vector spaces.”

4.2 Design Methods

The field of object-oriented design is still relatively modern. C++, the most common and popular object-oriented programming language, was only created in 1979 [Stro94]. Ten years later in 1989 there were still only 50,000 users world-wide. While it is true that object-oriented techniques and concepts have been around much longer than C++, widespread use of object-oriented programming did not appear until C++ was created. The
lack of maturity of the object-oriented field has led to some confusion as to what constitutes a good object design.

For procedural languages, a quality, maintainable design can be achieved by using a structured design method. This involves two primary steps: analysis and design. Structured Analysis is responsible for producing a specification from some problem domain. Structured Design converts the specification into a form that may be implemented by a programming language.

Object-oriented design approaches a problem in a considerably different manner than a procedural technique. This results in a need for new methodologies. First-generation methods were created as modifications of existing structured methods. The distinction between analysis and design is, however, not clear for object-oriented design. Some people disagree with this statement [deCh92a], while others such as Monarchi and Puhr [Mona92] state, "...the analysis and design stages in OO development are frequently commingled." For object-oriented design, the consensus is that analysis produces a specification of the problem domain with classes. Design organizes those classes into hierarchies and subsystems. This distinction is not clear because it is difficult to define a class without simultaneously identifying the hierarchy it will contribute to or is derived from. As noted by Constantine [deCh90], "...the models developed in object-oriented analysis and in object-oriented design can be expressed in the same or equivalent notations based on common principles." This has led several methodologies to simply abandon previous techniques by not separating the process into analysis and design phases. Some methods, such as Wirfs-Brock's Responsibility-Driven design [Wirf90], even cloud the process further by defining the process of creating hierarchies and subsystems as an analysis process - something that may be construed as design to someone familiar with structured methods.
Structured design methods have been in existence for some time and have generally converged on the same notation and definitions. Object-oriented methodologies are not quite as mature. As noted by Coleman [deCh91], "currently there is a deluge of object-oriented analysis and design methods and CASE tools." Several reports have been published comparing and contrasting existing analysis and design methodologies [deCh92c][Grah93b][Shar93], however there is still not a consensus among the field. Some suggest that there never will be and that it is best to just start with a general method and specialize it for one's specific needs [Mona92]. This is the approach taken in this dissertation.

The remainder of this section will review some of the more popular design methodologies. Lastly, some engineering-specific issues not addressed by most methodologies will be discussed.

4.2.1 Example Methodologies

This section will briefly introduce the Booch, OMT, OOSE, and Responsibility-Driven Design methodologies. A list of other miscellaneous methods is also presented.

4.2.1.1 Booch

The Booch method is the work of Grady Booch [Booc91]. The method consists of the following four steps:

1. Identify classes and objects at a given level of abstraction
2. Identify the semantics of these classes and objects
3. Identify the relationships among these classes and objects
4. Implement these classes and objects
The process is recursive as Booch notes, “the process of object-oriented design starts with the discovery of the classes and objects that form the vocabulary of our problem domain; it stops whenever we find that there are no new primitive abstractions and mechanisms or when the classes and objects we have already discovered may be implemented by composing them from existing reusable software components.”

Booch’s guidelines for developing classes are somewhat vague, and rely on the expertise of the designer. The concept of abstraction is thoroughly investigated and Booch’s methods are actually more of an attempt to define what an object should be. Using this general definition, the programmer is to derive the objects from his problem domain. The most reusable or practical object, however, is not necessarily the most conceptually ideal object that Booch describes.

Booch is more well known for his contributions to the object-oriented documentation process. “Booch diagrams” are often used to describe object-oriented systems regardless of how the design was derived. The primary documentation tools provided by Booch include:

- class structure diagram
- object structure diagram
- module structure diagram
- process structure diagram
- state transition diagram
- timing diagram

4.2.1.2 OMT - Object Modeling Technique

The Object Modeling Technique (OMT) is one of the most thorough object-oriented design methods [Rumb91][Brue92]. It is the work of James Rumbaugh and associates at
General Electric. OMT breaks the development process into the following phases:

- Analysis
- System Design
- Object Design

The analysis phase starts with a requirements specification and proceeds by building several models using different notations. The models are the Object Model, the Dynamic Model, and the Functional Model. System Design is responsible for organizing objects into subsystems, identifying areas of concurrency, and allocating and managing system resources. Object Design involves translating the design into actual data structures and code.

4.2.1.3 Objectory: Object-Oriented Software Engineering

Objectory is the contribution of Jacobson and Lindström [Jaco93]. This method is one of the oldest object-oriented methods, having its roots in the development of Swedish telephone systems over 20 years ago. The method is very expansive, covering the entire software development life cycle.

Like most object-oriented methods, the Object-Oriented Software Engineering (OOSE) method is iterative. The phases of analysis, construction and testing are the primary stages that are iterated on. OOSE distinguishes itself by the introduction of use cases. Use cases are descriptions of how users interact with a system. Use cases are used in conjunction with actors. An actor is an external entity with which the system interacts.

4.2.1.4 Responsibility-Driven Design

Responsibility-Driven Design is the work of Wirfs-Brock et al. [Wirf89][Wirf90]. The premise behind the methodology is that an object supports a set of cohesive
responsibilities. The basic steps in this method are broken down into an exploratory phase consisting of:

- identification of classes
- identification of responsibilities for classes
- identification of collaborations between classes

The exploratory phase is followed by a refining phase which:

- builds hierarchies between classes
- organizes classes into subsystems
- defines protocol

Recently the refining phase has been extended to include the use of stereotypes [Wirfs93]. Stereotypes are used to generalize the functionality provided by a class.

Responsibility-Driven Design (RDD) is the base method used by this dissertation. RDD is extended to fit the specialized needs of the Dynamic Integration System. A more complete review of the Responsibility-Driven Design method is presented in Appendix C.

4.2.1.5 Miscellaneous Methodologies

There are many more methods in addition to the above-mentioned techniques. Some of these methods are extensions of structured design strategies while others are second-generation methods incorporating the best features of existing first-generation techniques. Still other methods are specialized methods created to fulfill specific needs not met by broad-sweeping methods. The following is a list of some additional methods:
Coad and Yourdon [Coad91a][Coad91b]
Fusion [Jere93]
Martin and Odell [Mart92]
SOMA [Grah93a]
Edwards and Henderson-Sellers [Edwa93]
Pun and Winder [Pun89]
Gossain and Anderson [Goss90]
Contracts [Helm90]
Roles [Ande92]
layered components [Mats90]

Research has also been performed on improving existing methodologies and techniques. A sampling of this research includes:

- modeling transverse classes and objects [Kris93]
- developing rules for modeling analysis [Lang93]
- adding formalization to the design process [Haye91][Hone93]
- improving the inheritance mechanism [McGr93]

4.2.2 Problems with Existing Methodologies

While the above methodologies have served the computer science community well over the years, there are several reasons they are not commonly used by engineers\(^5\). Most of

\(^5\) These issues are based on the assumption that the engineer is aware of the concept of a methodology and desires to use one. The unfortunate truth is, however, that even when proper methodologies do exist, most engineers remain uninformed of their existence.
these reasons are based on erroneous assumptions on the behalf of the methodology. Some of these assumptions include:

- **that an application is being developed from scratch.** As stated by Adams [Gris91], "...most methodologies for the design of object-oriented systems have very little to say about the effect of reuse, and usually assume a clean slate to begin a design activity." Many engineering projects develop in a manner similar to ACSYNT – the continual improvement of or addition to some base-line code over a period of many years with a myriad of independent developers.

- **that the entire application is being developed at one location.** Most methodologies assume complete control of the design process. When development occurs at more than one site, it is not always possible to make large, broad-sweeping changes. While it may be argued that the initial design should be jointly formed before groups separate and begin development, this argument ignores the iterative nature of the design process.

- **that the final application should consist of tightly coupled objects.** The goal of most methodologies is to develop the final application. The unit of reuse is deemed the object. Higher levels of reuse, such as dividing an application into several modules, are not addressed\(^6\). Considering the previous two arguments, engineers need a higher level of reuse. Except for utilities, the class level is generally too low.

These issues are addressed in this dissertation by customizing a method to accommodate the problems. The approach is to devise a technique where independently developed components can be created that require minimal effort to be integrated. This will permit independent as well as incremental development of application modules.

\(^6\) There are exceptions to this, one being Responsibility-Driven Design which segregates objects into tightly coupled groups that are themselves loosely coupled.
4.3 Software Reuse

Broadly speaking, the problems discussed in this dissertation fall under the category of software reuse. The ability to connect new analysis modules to an existing application is a form of reuse for those modules. While the ideas of software reuse are not new [McIl68], the goal of producing easily reused software components is far from a reality\(^7\). As noted by Chen [Chen93a], "...software components designed in OOP are easier to be reused [sic] than those designed in conventional programming. But the state-of-the-art software reusability in most OOP environments is still very limited." This section surveys some of the concepts applied to software reuse and some of the attempts to improve reuse.

4.3.1 Obstacles to Reuse

There are many reasons why software is not reused as much as perhaps it should be. The reasons for this vary. Some feel that the primary obstacles are managerial [Gris93][Beck92], while others feel the problem is technical [Nova92]. According to Seidewitz et al. [Seid93], "...there seems to be a consensus that the most difficult problems in achieving reuse are, in the end, largely economic and managerial," however, according to Meyer [Meye87], "reuse is limited because designing software is hard." Most likely the cause lies on some common ground.

Some examples of managerial problems include [Meye87]:

- Economic incentives tend to work against reusability. If a contractor delivers software that is too general and too reusable, then the client will not need the contractor for the next job.

---

\(^7\) The keyword here is "easily." Engineering software is reused, but not without an expenditure of great effort. So great is the effort that, when including future maintenance issues, it perhaps exceeds that of implementing a newer more manageable design.
• The famous not-invented-here complex works against reusability.

• Reusable software must be retrievable, which means we need libraries of reusable modules and good database searching tools so client programmers can easily find appropriate modules.

Some examples of technical problems include [Chen93b]:

• the specifications of software are either nonexistent or sufficiently ambiguous so that it is not possible to determine exactly what the software does without understanding all the source code.

• software that can perform the required task is available but it is so general that it is too inefficient for the task.

• lack of a standard data interchange format limits both data sharing among applications and reusability.

This dissertation approaches reuse from the technical standpoint. The target environment for the work presented here is typically an engineering group that is involved in software development for itself. This implies a desire for reuse (because the code is their own) which eliminates many of the managerial problems.

On the technical side of the problem, various research projects have attempted to improve or identify problems associated with reusability. A sampling of these projects includes:

• the direct use of variables from within classes [Wirf89]

• events and sensors [Vite90]

• reuse using the contract construct [Holl92]

• design patterns [Gamm93]

• fine-grained versus coarse-grained reuse [John92a]
reuse by interface adaptation [Purt91][Nova92][Nord90]
the OATH library [Kenn92]
module and library interconnection languages [Prie86][Gogu86]
copying and swapping variables [Harm91]

4.3.2 Reuse Research Projects

The general goal for effective software reuse has been the development of reusable components. The concept of building software like hardware has largely been attributed to Brad Cox [Cox86]. Cox questions why software cannot be packaged and distributed the way hardware’s integrated circuit chips are. This section examines some of the issues related to software reuse in the form of component-oriented design.

Component-oriented design revolves around the creation of reusable components. Object-oriented design lends itself well here, but as Nierstrasz et al. notes [Nier92], “present object-oriented methodologies do not explicitly address the design of reusable frameworks.” One reason for this is simply that the design of reusable components is very difficult. According to Nierstrasz et al., developing components for reuse “is an order of magnitude more difficult than that of developing a single, specific application.”

The goal of component-oriented design is of course, to create environments where applications can easily and quickly be assembled by connecting parts. This permits users to construct custom applications. Corcoran quotes James Allchin, vice president of Microsoft’s Advanced Systems Division, as saying that [Corc93], “for people to become more productive in the information age, there can be no canned programs.” The need for software components is widely acknowledged. A U.S. Department of Defense analysis concluded that [Coom90], “...one issue stands out as being crucial to the development of cost-effective, timely and reliable software – software component reusability.”
A common form for packaging reusable components is through the visual metaphor. Visual scripting can be described as the visual, interactive construction of an application. Components are represented on-screen in some manner. The user visually interconnects the components so that they perform some desired function. While it is not the goal of this dissertation to create a visual environment, inspiration can be borrowed from the techniques used to connect components in such environments. The ease at which these components may be connected can and should be applied internally to code at the module level in nonvisual environments.

de Mey has created a system where components are defined as having ports [deMe94]. Each component has input and output ports and each port has a defined type. Compatible ports may be connected visually using a tool called Vista [Nier90b][Nier91]. Once a set of components is connected, the application is formed. Another example of a visual environment is the data visualization system VISAGE [Schr92]. On a more mathematical front is the Snap-Together Mathematics project [Glei90]. This application is similar to de Mey’s except that components are always mathematical functions and ports are used to transmit values and derivatives of the functions. The Fabrik environment is another example of visual programming [Inga88].

The visual environments described are generally referred to as channel-based environments. Information of some sort is passed from component to component via channels. The goal is usually to start with some input and end with some desired output. The deficiency of this approach is the simplicity at which information must flow. As noted by Lang et al. [Lang91], “... the data flow networks [channels] of current network [channel-] based packages don’t support module loops or recursive execution of module changes, nor dynamic, calculation controlled restructuring of parts of the network or dynamic generation of new networks.” In addition, channel-based environments are
primarily designed for analyzing some data set, not creating it. This limits their functionality for design environments such as CAD systems.

* Non-channel-based visual environments usually consist of static components that may be placed in other objects. This is becoming more commonplace with the advent of graphical user-interface environments. A component may be created and "pasted" into an application. Examples include Microsoft's OLE and Apple's Apple-Links. Research into this type of component development has been performed by Cowan et al. [Cowa93].

Research into component reuse has also been performed on larger scales, focusing on the entire design process. Perhaps the largest project centered on component-oriented software development is ITHACA [deMe93][Ader90][Pröf89][Fugi92]. The goal of ITHACA is to produce a complete object-oriented application development environment that can be easily adapted to various application domains.

4.4 Integration Techniques

The ability to reuse components of software implies that they can be easily integrated. This section examines some of the issues related to integrating software. This includes difficulties associated with integration as well as current techniques used to perform integration.

When speaking of integration, there are many forms that can be addressed. These can be broadly divided into the following categories:

- **process integration** - the integration of the internal organization and flow of control of modules.
- **data integration** - the integration of the information modeled by each module.
• presentation integration - the integration of user-interaction techniques and appearances of modules.

This dissertation focuses primarily on issues related to data integration. Process integration is addressed, although not directly. The methods developed in this dissertation permit dissimilarity-structured components to interact as long as they all employ some Dynamic Integration System techniques. Presentation integration is not addressed as it is more of a human-factors, case-by-case issue.

On the front of data integration, Nilsson et al. have identified several strategies [Nils90]:

• manual data transfer
• automatic data transfer
• common database
• process-to-process communication

The first two forms of integration are very primitive. A common or centralized database is a slightly higher level of integration, although for a highly interactive application such as a mathematical analysis that frequently needs access to parameters, most existing databases are not applicable. The last form of integration, process-to-process integration, involves applications or modules directly “talking” with one another. This is a difficult task but can result in a superior application.

Integration techniques are also broadly categorized by Pennington into the following forms [Penn91][Penn92]:

• rigidly-connected interfacing
• rigidly-connected coupling
- freely-connected interfacing
- freely-connected coupling

Rigidly-connected applications are those which are coupled or interfaced in a manner that is specific to the applications being joined and is not easily extended to incorporate subsequent additions or changes. Freely-connected applications are those which are coupled or interfaced in a way that is more generic in application and which has a large number of similarities between the joining of one set of programs and another. Interfacing is defined as data passing and coupling is defined as database methods.

### 4.4.1 Difficulties Associated with Integration

Integration of software is difficult for many reasons. Specific reasons related to engineering-oriented analysis software were presented in the last chapter. In general, however, integration is difficult just because the tools and techniques currently available do not adequately address the issue. As noted by Höfle [Höfle93], “current programming languages (and programming environments) are not very well equipped to handle the subtle integration problems likely to occur with pervasive reuse.”

Some general problems are also identified by Ramamoorthy et al. [Rama92]. Ramamoorthy et al. state, “when software components are developed by different groups, tracing the dependencies among the combined components is a complicated task....” and, “during the integration process, problems can arise because component softwares have not been developed with integration in mind, therefore, a lot of mismatches can exist among them.” The solution to these problems according to Ramamoorthy et al. is that, “software components should be designed to promote reusability so as to require as little re-engineering as possible in their integration. We need methodologies to guide us in producing softwares with a high potential for reusability.” The methodology developed
in this dissertation addresses the problems Ramamoorthy et al. has raised concerning engineering-related software.

4.4.2 Integration Techniques

This section examines several techniques for integration. These include:

- **standards** - standards provide a rigidly defined interface through which applications may communicate.
- **databases** - databases may be used in several ways to integrate modules/applications. This is one of the older techniques used for integration [Beeb83].
- **event-based systems** - event-based systems are a relatively unexplored design technique that de-couples components.
- **miscellaneous techniques** - techniques not classified by the previous groups.

4.4.2.1 Standards

Standards provide a common interface for separate applications. By conforming to the standard, two applications may share data. Example engineering software standards are the CAD Framework Initiative (CFI) for electronic design automation [IEEE92][Kapl92] and IGES and PDES for CAD/CAM data exchange [Smit83]. When designed properly, standards provide a convenient method for information exchange. The deficiency of standards is that they are slow to develop, often lagging years behind what is needed at the time. By the time some standards are ratified, the technology they represent is antiquated.

4.4.2.2 Databases

Using databases for integration is one of the oldest forms of computerized integration [Beeb83]. There are several ways in which a database may be used as a mechanism for
integration. They may be used as a central storage location through which each module may exchange data [Lang91] or as a repository in which applications may store and retrieve design knowledge [Park88]. This section briefly examines several approaches using databases to integrate components.

Rowell et al. have created a system called EASIE (Environment for Application Software Integration and Execution) [Rowe88]. The environment uses a relational database to store variables needed as inputs and outputs to analyses. The central database provides a common method for each analysis to communicate as well as the ability to record configurations as a design evolves. Analysis execution is controlled manually or by using scripts from within EASIE.

In terms of engineering-related databases, most research has been associated with object-oriented databases. An object-oriented database forms a method for creating persistent classes. Research into object-oriented databases for engineering design can be found in several places [Kim96][Nguy92][Venu90].

A database can also be used to integrate several smaller databases. The ZOOF project is a Heterogeneous Database System (HDBS) [Hárt93]. A HDBS is a federation of heterogeneous component database systems. The top-level database interacts with each individual database giving global users the illusions of an integrated database system. A similar approach is taken by Amin with the development of the Global Data Manager [Amin92].

4.4.2.3 Event Programming

Event programming is a relatively unexplored field. Events are a mechanism that tie together a triggering object and a receiver object. The triggering object performs some action that activates the trigger. The receiver object receives the event and performs the
necessary actions. The goal of this form of communication is to de-couple the triggering and receiving objects. By generalizing relationships, objects can be designed without specific knowledge about the objects they will communicate with and as noted by Sullivan and Notkin [Sull90], "designs that effectively separate component from relationship concerns minimize the changes that have to be made as requirements change. To achieve this separation, relationships should be designed as independent components, distinct from components they relate."

Research into events has been performed, directly or indirectly, by the following projects:

- events and sensors [Vite90]
- the Field environment [Reis90]
- dynamic variables [Turn90]

Events are explored in greater detail in Chapter 5 when the Dynamic Integration System is introduced. The existing systems will be compared and contrasted with the system developed in this dissertation.

4.4.2.4 Miscellaneous

Several research projects have addressed issues related to integration without falling into one of the previously described categories. Nanard et al. [Nana90] and Berre [Berr92] have created systems where object-type wrappers are placed around applications. This provides a common manner with which to interact with each application thus enabling them to be integrated on a high level. The concept of using agents for integration is also a topic of research [Hale94][Khed93]. Grieshaber suggests a CASE tool that assists developers implement distributed data sharing solutions in their applications [Grie91][Grie92].
Chapter 5  Dynamic Integration System

This chapter describes the framework created for this dissertation. A description of the concepts utilized is first presented. The classes in the framework are then introduced. A brief description of how these classes internally interact is next presented. This is followed by a complete set of design documents for the framework including CRC cards, inheritance hierarchies, collaboration graphs, and walk-throughs. Finally, sample code (patterns) demonstrating how to perform various tasks using the framework is presented.

5.1 Concepts

This section introduces the concepts that form the foundation for the Dynamic Integration System. The two primary concepts are:

- Dependency Hierarchies
- Event-Responder Pairs

The concepts are first introduced and explained. The application of the concepts to this dissertation is then compared and contrasted with existing techniques found in the literature.

5.1.1 Dependency Hierarchies

A dependency hierarchy is a directed acyclic graph. The graph shows dependencies between parameters. A simple dependency hierarchy for the equation \( A = B + C \) is shown in Fig. 4.
Figure 4 - Example Dependency Hierarchy

In this hierarchy, A is dependent on both B and C. Each parameter in an analysis can be modeled as a node in a dependency hierarchy tree. Arrows emanating down and out of a parameter represent dependencies of that parameter. Arrows that point to a parameter emanate from other parameters affected by the parameter.

Values that are dependent on other values are said to be \textit{constrained} by those values. Any changes to a dependency of a parameter mean that the parameter needs to be recalculated to be considered \textit{valid}. A parameter is considered \textit{invalid} if one of its dependencies has been modified since the last time the function constraining the parameter to its dependencies was evaluated.

A more complicated dependency hierarchy involving the mass-spring-damper analysis described in Appendix A is shown in Fig. 5. To simplify the diagram, arrow heads have been removed. Dependencies in this diagram always originate from the bottom of a parameter and go downward.
Figure 5 - Partial Mass-Spring-Damper Dependency Hierarchy

Modeling an analysis as a dependency hierarchy is one of the key concepts employed by the Dynamic Integration System. Once a hierarchy is constructed, interaction with an analysis is greatly simplified. For now assume that there are four primary ways to interact with nodes in the hierarchy:

1. **access the value** - the value of any parameter may be retrieved

2. **change the value** - values may be specified for any parameter that is unconstrained, i.e., any parameters that do not have dependencies.

3. **validate the value** - when a parameter below a given parameter is modified, the upper parameter is marked invalid. Functions may be registered with the parameter that inform it how to validate itself when needed.

4. **respond when the value has changed** - functions may be registered with any parameter to be called when the parameter is modified.

The last two forms of interaction are actually event-response pairs. A validate event may cause some function (response) to execute. Likewise, a change trigger may also cause
some function to execute. The actual techniques used to model nodes in the hierarchy and the mechanisms used to interact with them are discussed further when Dynamic Variables are introduced.

The dependency hierarchy manages data interaction using a lazy-evaluation technique. This process involves marking each node as either valid or invalid. Whenever a parameter is changed, parameters dependent on that parameter are marked invalid. These parameters are easily identified in the hierarchy by the lines emanating upward from a particular parameter. This process continues recursively up the dependency tree. The invalidation process stops climbing a particular branch when either a previously invalid node is found, or a node is found without any upward connections.

After several nodes have been modified, the resulting tree will consist of a set of valid and invalid parameters. This process can be seen in Fig. 6. In this figure $F_0$ has been modified. The result is the invalidation of $X$. This in turn results in the invalidation of $x_{\text{steady-state}}$. Invalid parameters are shown in gray. Other parameters in the system are left alone as they are not affected by changes to $F_0$. 
Figure 6 - Invalidation Process

When an attempt to access an invalid parameter is detected, a validation process occurs. Validation involves calling a function registered with the invalid parameter. This function knows how to calculate the value of the parameter from its dependencies. During this function call, the value of each of the dependencies will be accessed. If any of the dependencies are invalid, then a validation process starts for that parameter. The validation process stops once a parameter is calculated from a valid set of dependencies.

The validation process can be demonstrated using the hierarchy that was modified in Fig. 6. In this particular state, if $x_{\text{steady-state}}$ is accessed a validation process will be started. The validation process is started because $x_{\text{steady-state}}$ is currently invalid. The validation process calls a function which attempts to evaluate $x_{\text{steady-state}}$ using $X$, $\omega$, $\phi$, and $t$. At this point, only $X$ is currently invalid. The values for $\omega$, $\phi$, and $t$ are immediately used while a validation process for $X$ occurs before its value can be used. Once valid values for $X$, $\omega$, $\phi$, and $t$ have been obtained, a new value for $x_{\text{steady-state}}$ is calculated and $x_{\text{steady-state}}$ is marked valid.

Chapter 5 - Dynamic Integration System
Modeling an analysis as a dependency hierarchy has several advantages:

- **easy access.** Any parameter in the dependency tree may be accessed without knowledge of its dependencies. The validation of parameters is automatically handled by the underlying dependency tree management routines. This means that to use a parameter, a list of parameters from which to calculate the parameter does not need to be supplied.

- **easy management.** Management of data dependencies has always been a problem in analysis modules. A dependency hierarchy simplifies this problem by forcing the programmer to specify the immediate dependencies for a particular parameter. A hierarchy is formed when a parameter with dependencies becomes a dependency for another parameter. The simplicity of building a hierarchy this way is that only the immediate dependencies for a particular parameter need be specified.

- **simplified integration.** The primary problem of integrating two analyses is connecting the outputs of one module to the inputs of the other. If two analysis modules are created using dependency hierarchies, then this process is reduced to the problem of grafting one branch of the tree to another. Integration is essentially the process of building a bigger dependency hierarchy.

- **efficient evaluation.** The lazy-evaluation form of interaction with the analysis is the most efficient form possible. When a parameter is modified, only other parameters dependent on that parameter are marked invalid. Likewise, when a parameter is validated, only dependencies of that parameter are validated. A parameter is never calculated a second time until a dependency is modified.

The concept of the dependency hierarchy has to this point been presented in a conceptual nature. The key to using a dependency hierarchy is finding a way to model the hierarchy and to find a way to interact with its nodes. Dynamic Variables and Dynamic Responders fulfill this purpose.
5.1.1.1 Comparisons to Constraints

A dependency hierarchy can be viewed as a system of constraints between parameters. This view differs from the common implementation of constraints. Constraints are usually associated with user-interfaces or some sort of geometric display. Items are constrained to one another. For instance, the midpoint of a line is constrained to be the point lying halfway between two other points. This is the point of view taken by such systems as the ThingLab [Born81][Free89] and the Kaleidoscope [Free90][Free92] projects.

These types of systems usually define language extensions for expressing constraints. The compiler interprets these expressions into a pool of constraints. Perturbing an input, such as moving the endpoint of a line, causes a constraint satisfier to attempt to determine a new state in which all constraints are satisfied. This is accomplished by iterating through the constraints until an acceptable solution is found. Once found, that solution is translated into a set of actions to be performed. The dependency hierarchy developed in this dissertation differs from these types of systems in the following ways:

- **only natural language features are used.** The use of a special language severely limits portability and the ability to integrate with existing codes. For this reason the Dynamic Integration System is implemented entirely in C++ with no pre-processing necessary.

- **no constraint compilation exists.** Only one type of constraint exists in the Dynamic Integration System - the dependency constraint between two parameters. Perturbing an input always results in the need for some specific output parameter to be re-evaluated. Removing constraint compilation removes flexibility, but greatly enhances performance thus making dependency hierarchies applicable to engineering analysis-type modules where performance is an issue.
• **constraints are absolutely satisfied.** In a constraint management system it is possible that certain constraints remain unsatisfied due to conflicting constraints. In such systems, constraints may be weighted for added expressiveness. The Dynamic Integration System requires that a constraint be absolutely satisfied. When modeling an equation, the output of that equation cannot be used unless all inputs are valid. Violation of this constraint would result in a faulty analysis.

• **only uni-directional constraints are employed.** Most constraint systems allow general statements like \( x = a + b \) to be employed. This can be interpreted in three ways: \( x = a + b \), \( a = x - b \), or \( b = x - a \). The constraint compilation process attempts to find values of \( a, b \), and \( x \) so that one of these conditions is met, i.e., \( x, a, \) or \( b \) may be changed to satisfy the constraint. Since the Dynamic Integration System is optimized for speed and does not support constraint compilation, constraints are unidirectional. In terms of the above example, this means that only \( x \) may be changed to meet the constraint.

### 5.1.1.2 Comparison to Miscellaneous Systems

Some constraint-based systems have been optimized for the use in numerical-type problems by implementing some of the above conditions. Kolb has built a constraint-based system for designing aircraft [Kolb88][Kolb92]. The system allows the designer to specify a pool of equations. A constraint satisfier evaluates these equations, inverting them if necessary, to determine values that are consistent.

Kroo also has constructed dependency hierarchies [Kroo92]. His hierarchies are, however, built among procedural routines in an analysis. One routine becomes the dependent of others. This is a slight variation of the dependencies presented here where variables are dependent on one another. Kroo also dynamically forms execution paths (a form of constraint compilation) which slows analyses as they grow large.
Newton et al. have created a symbolic spreadsheet for purposes of simulation [Newt91]. Unlike a conventional spreadsheet, cells are not arranged into rows and columns. Instead, cell arrangement is hierarchical in nature. These cells may be addressed from within a program by referencing a parent and its successive children until the specific cell is found. This differs from the Dynamic Integration System where cells (nodes in the hierarchy) are globally identifiable by a unique name. The contents of cells may either be values or formulas. Formulas are interactively specified by user, dynamically compiled, and loaded at run-time. References in formulas are bound using lazy-evaluation techniques. In this manner, a dependency hierarchy is constructed among cells.

5.1.2 Event-Response Pairs

Event-response pairs are known by several names in the literature: active objects, active databases, or simply events and sensors. The different names arise from the fact that the style of programming was independently arrived at and applied to several fields including concurrent constraints, database management and software development.

The concept of events and responses is fairly simple. One object performs some action that triggers some event. In response to the event, some other object reacts. The most common use of events is in the field of user-interface design. Events, including button presses or mouse movements are triggered by the user. An application responds to these events by performing some action. This process is diagrammed in Fig. 7.
Figure 7 - Event-Response Pair

Events were originally implemented in user-interfaces to free-up applications from having to continuously poll input devices. Other subtle advantages are, however, also present. One such advantage is that objects responding to an event are not tied to the object triggering the event. This is a de-coupling mechanism and provides a convenient separation between the interaction of modules. This advantage is one of the reasons behind the implementation of an event-response mechanism in the Dynamic Integration System.

Events are implemented in the Dynamic Integration System as actions that occur to variables. These variables are, in terms of the previous section, nodes in the dependency hierarchy. Events are triggered whenever the value or state of one of these variables changes.
The primary purpose of event-response pairs in the Dynamic Integration System is to allow one module to know when another module modifies a particular parameter. This is a problem that is very evident in programs that rely heavily on global data. One module sets up the global space with specific values. Another module slightly disrupts the order. If the first module is not made aware of the change, it will continue based on the assumption that no data has changed. This can be a very dangerous assumption.

Attaching events to variables allows modules dependent on data to be automatically made aware of changes to the data. There is no need for the module making the changes to notify other modules of the changes. This is opposite of the traditional approach to data management. The interaction between modules using event-response pairs is contrasted with the traditional approach in Fig. 8.

![Diagram]

Figure 8 - Application of Event-Response Pairs

Chapter 5 - Dynamic Integration System
The traditional approach requires that each module directly communicate with other modules. This assumes that each module knows the protocols of the other modules. The approach also makes it difficult to introduce new modules into an application because existing connections must be modified or new ones constructed. The event-response approach does not require direct communication between modules. When one module makes a change, it broadcasts the change in the form of an event. Other modules dependent on a parameter “tune into” certain events. Using this approach, minimal connections are built between modules. The addition of a new module to an application involves only tuning into and broadcasting the correct events.

Some other advantages to using event-response pairs include the following:

- **improved management of data.** As discussed, it is not the responsibility of the modules making changes to data to notify other modules dependent on the data. It is instead the responsibility of modules dependent on data to monitor events pertaining to that data. This provides a large boost to the robustness of the code because complex management schemes are eliminated.

- **increased encapsulation.** Since each module does not need to be aware of data residing in other modules, knowledge is localized. The less a module has to know about its surroundings the better the module can be encapsulated.

- **increased reusability.** By improving encapsulation and decreasing coupling, the ability to reuse a module increases. The resiliency of the module to changes outside the module is improved. This allows modules to be designed and used without prior knowledge of where they will be used.

- **eased integration.** Implementing and integrating a new module involves tuning into the proper events. This does not require change to existing code.

- **simplified design.** Modules using events can be designed autonomously. A module does not need to be aware of the effects of the changes to parameters it is making.
5.1.2.1 Comparisons to Event-Response Mechanisms

The concept of events is not new. Event-response mechanisms have been applied, both in theory and practice, to several fields. This section examines some of these approaches.

The event-response mechanism proposed by Murata and Kusumoto [Mura89] is a system involving daemon processes - processes that lay dormant until awakened by some trigger. The problem the authors were trying to solve was the same problem identified by Wirfs-Brock [Wirf89]. The problem is that building a flexible system involves distributing knowledge throughout the application. This, however, often compromises the maintainability and reusability of the system. Distributed knowledge confuses and complicates the process of consistency management. Murata and Kusumoto approached the problem by attaching daemon processes to the methods of objects that modify data. When these functions are executed, the daemons awaken and call registered functions that maintain consistency among data. The authors state that using this methodology permits manager-type objects to release their constituent objects without worrying about maintaining the relationships between them.

- The concepts applied by Murata and Kusumoto are also used in this dissertation. Differences lie in the fact that the daemon processes are attached to data rather than the member functions of classes\(^5\). Anywhere the data is modified, these daemons awaken. This difference specializes the Dynamic Integration System to numerical type applications.

Vitek et al. have approached the application of events for purposes of improving reusability [Vite90]. The authors present a system composed of events, sensors, and an

\(^5\) True daemon processes are not actually used in the Dynamic Integration System. The use of daemons implies concurrency which is not an issue addressed in this dissertation.
ether. Events claim spots on the ether and sensors tune to those spots. Objects create sensors for monitoring certain events. Objects may also generate events that will then be picked up by sensors. Using this methodology, objects may indirectly communicate with each other by sharing common events and sensors. Reusability is enhanced because objects may be created independent of their environment.

As Vitek notes, "object-oriented programming suffers from a closed-world assumption: only objects that conform to known interfaces can be composed." This problem is also acknowledged by the Dynamic Integration System. As a result, a goal for this dissertation is to be able to create analyses and supporting modules without prior knowledge of the application they will be used in. This is accomplished by using an event/sensor mechanism that is specialized to the application of numerical programs. This involves optimizing efficiency of the event/response mechanism by defining a limited set of events only applicable to numerical parameters.

An example application of Vitek et al.'s ideas can be found in the Field environment [Reis90]. The Field environment employs a server that acts as the ether described in Vitek et al.'s work. Modules register events with the server that they are interested in receiving. As an application executes, modules notify the server of events. The server then selectively broadcasts the events to the modules that are interested in the event.

The Field environment represents a generic, rather inefficient, implementation of an event-response mechanism. Events are represented by text strings. For each event generated, the server must search the list of recorded events for a match before knowing where to broadcast messages. The overhead involved in this search prohibits this kind of event-response mechanism for use in a mathematical analysis.
The applications described thus far have used events at a fine-grained level, implementing events/responses at the object level. Research applying events has also occurred on a large-grained basis. The COOP project presents such an example [Berr92].

The COOP project aims to integrate several large applications. Each application is encapsulated with an object-like structure. Five methods of communication are described for these mega-objects. One of these methods is a notification-oriented control scheme. This type of response mechanism is applied at a high level among several, possibly concurrent, objects.

5.1.2.2 Comparisons to Active Databases

While the concepts behind an active database are similar to the event-response mechanisms used in the Dynamic Integration System, the implementations are actually quite different. Active databases grew out of the need to maintain consistency among objects in a database. Initially, mechanisms were only provided for an object to monitor changes to its state. More recent research has been applied to reacting to changes in states to other objects. The current technique for accomplishing this is to register events that are triggered when member functions from an object are executed. Some methods even allow events to be generated at the beginning and end of functions [Anwa92]. Most active databases take the process of events one step further using rules. Rules are associated with events to ensure the integrity of an object or set of objects. In addition, triggers may be registered to occur based on complex (i.e. conditions involving AND/OR type statements) conditions. One goal of current active database research is to provide "seamless" mechanisms for incorporating events [Kapp94]. This means either extending or creating a database or object-oriented language to provide event-like features.
5.2 Dynamic Integration System

The Dynamic Integration System consists of two primary entities: Dynamic Variables and Dynamic Responders. Together these two types of objects are used to construct dependency hierarchies. Additional features are provided by the Dynamic Entity Responder and Dynamic Registry Responder classes. This section describes the general structure of these classes and their purpose.

5.2.1 Dynamic Variables

Dynamic Variables are objects used to model nodes in the dependency hierarchy tree. Text strings representing names are used to identify Dynamic Variables. A Dynamic Variable is created by supplying one of these text strings, herein referred to as a Dynamic Identifier and identified using bold-italic text, to the Dynamic Integration System. The Dynamic Integration System then returns an instance of a Dynamic Variable. This Dynamic Variable acts just like any other built-in type such as a float or int variable with the added feature that it is a node in a dependency hierarchy.

Dynamic Variables have been implemented as a C++ class. Common operators such as “=”, “+”, “+=” have been defined for Dynamic Variables so that they behave like normal types. When one of these operators makes a change to the Dynamic Variable, it also performs an invalidation process. Likewise, whenever an operator needs to access a value, it performs a validation process if needed.

The ability to overload operators in C++ allows Dynamic Variables to perform just as any other data type. This enables the programmer to use them without knowledge of the underlying Dynamic Integration System code. Using Dynamic Variables, interaction with the dependency tree is automatically performed through the overloaded operators.
As such, Dynamic Variables may be used in complex expressions while still interacting with the dependency hierarchy. An example of such an expression might be:

\[ x = \sin(y \times z / w) \]

where \( x \) and \( w \) are Dynamic Variables while \( y \) is an int and \( z \) is a float.

Multiple instances of Dynamic Variables may exist simultaneously. Each module that needs to interact with a node in the dependency hierarchy does so by creating an instance of the Dynamic Variable representing the node. As previously described, the Dynamic Variable is created by passing a Dynamic Identifier describing the parameter to the Dynamic Integration System.

An example of two modules instantiating two Dynamic Variables with the same name may be a user-interface module and an analysis module. This type of interaction is shown in Fig. 9. The user-interface module creates a Dynamic Variable for purposes of setting its value. The analysis module creates a Dynamic Variable so that it can access its value to perform other calculations. Both modules become effectively “dynamically integrated” because they interact with one another through the common Dynamic Variable. Each module individually interacts with the dependency hierarchy, one setting a value, the other retrieving it. By interacting with the hierarchy, both modules indirectly interact with each other.
Figure 9 - Example Dynamic Variable Usage

Dynamic Variables may have other Dynamic Variables as dependencies. If this is the case, the user is not allowed to modify the value of that Dynamic Variable because it is constrained by its dependencies. Only the immediate dependencies for a parameter need to be specified. The dependency hierarchy is automatically formed by the Dynamic Integration System. Since secondary effects are managed by the dependency hierarchy, the programmer can concentrate on one Dynamic Variable at a time. An intentional side-effect of this is that it is very easy to integrate several Dynamic Variables and their associated dependencies. This is accomplished by matching the Dynamic Identifiers of Dynamic Variables in two dependency hierarchies such that the nodes at the top of one tree become the nodes at the bottom of another. These integrated Dynamic Variables need not be designed by the same programmer or reside in the same module. The integration process essentially becomes a grafting process of connecting several branches (modules) to form a tree (application).
5.2.2 Dynamic Responders

The Dynamic Integration System validates invalid Dynamic Variables by calling some function that knows how to calculate a value for the parameter. This function is provided by the programmer and is registered with the Dynamic Integration System using a Dynamic Responder.

Dynamic Responders are objects that assist in the process of registering functions. A virtual function called request_validation() exists in each Dynamic Responder. When a Dynamic Variable associated with the Dynamic Responder needs to be validated, the Dynamic Integration System calls the request_validation() function from the Dynamic Responder. The Dynamic Responder may then call the necessary analysis function(s) required to validate the Dynamic Variable.

In addition to being able to respond to the request_validation() function, Dynamic Responders may also register functions that respond when Dynamic Variables are validated, invalidated, or modified. This is useful as an integration process. Modules that are dependent on specific parameters may create Dynamic Responders which register functions that are called whenever the specific event occurs. An example of this situation is a user interface module. If the interface module displays the value of several parameters on-screen, it is desirable to change those values whenever parameters in the program change. This can be easily accomplished with Dynamic Responders by registering functions to update the screen whenever a Dynamic Variable is modified.

5.2.3 Dynamic Entity Responders and Dynamic Registry Responders

In general, most interaction with the Dynamic Integration System can and should be accomplished using Dynamic Variables and Dynamic Responders. There are, however,
specialized circumstances when a more detailed level of interaction is necessary. This is where Dynamic Entity Responders and Dynamic Registry Responders come into play.

A Dynamic Entity Responder is used to respond to changes to Dynamic Entities. Changes include such occurrences as changes in states, addition or removal of dependencies, or the addition or removal of a Dynamic Variable linked to the Dynamic Entity. A Dynamic Registry Responder responds to changes occurring to Dynamic Variables in the Dynamic Responder. This includes anything from a change in state of a Dynamic Variable to the creation of a new Dynamic Variable. A Dynamic Registry Responder is primarily used as a catch-all responder as it receives events pertinent to all Dynamic Variables.

5.3 Framework Internals

The Dynamic Integration System is essentially a nonpersistent active database specializing in the storage of a set of singular parameters. Features are provided that simplify interaction with the database (primarily Dynamic Variables) and disguise its use. The user should not have to think of the system as a database. Instead the user should think of dependency hierarchies with the ability to respond to changes in the hierarchy. This section examines the Dynamic Integration System from an internal, implementation point of view.

The core unit of the Dynamic Integration System is the Dynamic Registry. The Dynamic Registry is a nonpersistent database of Dynamic Entities. Dynamic Entities are the basic unit of information storage in the Dynamic Integration System. Dynamic Entities also contain most of the intelligence associated with the system. This includes the ability to build dependency hierarchies and the responsibility of notifying Dynamic Responders when actions occur.
The Dynamic Registry is implemented as a set of Dynamic Entities. A set is implemented as a doubly-linked list of elements. While a set is not the most efficient form of storage for a database, it is suitable for the Dynamic Registry. The Dynamic Registry is more of a repository than a traditional database as queries and searches are usually not performed on it. Fig. 10 shows a graphical depiction of the Dynamic Registry.

![Dynamic Registry Diagram]

**Figure 10 - Dynamic Registry**

It should be noted that data members and functions in the Dynamic Registry are all declared static. This allows for multiple instantiations of the DRegistry class without multiple instantiations of the set representing the registry. Static member functions may also be accessed without instantiating the class using the syntax DRegistry::function().

The Dynamic Entity class is the workhorse of the Dynamic Integration System. A Dynamic Entity contains all information pertaining to a particular parameter. This includes a name, the data type, the data, state information, and dependencies. The graphical representation of the Dynamic Entity class is shown in Fig. 11.
Figure 11 - Dynamic Entity

Dynamic Entities are closely related to Dynamic Variables. A Dynamic Entity is created each time a Dynamic Variable is instantiated with a Dynamic Identifier not matching any of the existing Dynamic Entities in the Dynamic Registry. If a Dynamic Entity matching the desired name already exists in the registry, then the newly created Dynamic Variable is just added to the Dynamic Entity’s current reference set. Dynamic Variables provide a convenient interface to Dynamic Entities. End programmers usually never directly interact with Dynamic Entities. Instead they create one or more Dynamic Variables that relay operations to a Dynamic Entity. For each Dynamic Entity, there may be one or
more Dynamic Variables. For each Dynamic Variable, there is one and only one Dynamic Entity.

The Dynamic Entity stores the data usually thought of as associated with a Dynamic Variable. The Dynamic Entity does not know the type of data and stores it as a void pointer. The data is simply an abstract concept to the Dynamic Entity, having only a valid or invalid state. The Dynamic Entity does not perform any operations on the data and only provides a storage mechanism for it. Classes derived from the Dynamic Variable class such as Dfloat, Dint, or Ddouble do the actual allocation of the data since they know the data type. The first Dynamic Variable created with a specific Dynamic Identifier will allocate the data and store it in the Dynamic Entity. Once allocated, the data remains stored in the Dynamic Entity until the Dynamic Entity is destroyed.

Dynamic Variables access the data through functions in the Dynamic Entity when they need to perform some module-specific function on the data such as changing its value or looking at the value. If a Dynamic Variable is accessing the data, then the Dynamic Entity will first determine if the data is valid. If the data is valid, then a pointer to the data is returned. If the data is invalid, then the Dynamic Entity attempts to validate the data. If the Dynamic Variable modifies the value of the data, then it will notify the Dynamic Entity. The Dynamic Entity will then invalidate all of its forward references.

The Dynamic Entity maintains references to all dependencies, forward references, and Dynamic Variables linked to the Dynamic Entity. Elements in the forward reference set are invalidated whenever the current Dynamic Entity changes or becomes invalid. The backward reference set contains the dependencies used to validate the value of the data. Messages are sent to each of the elements in the current reference set each time an event occurs in the Dynamic Entity. Events include changes to the value of the data, changes in the state of the Dynamic Entity, etc.
The Dynamic Entity also maintains a special link to a Dynamic Variable called the *validator*. The validator is the Dynamic Variable that is connected to a Dynamic Responder that knows how to validate the value of the data. When the Dynamic Entity needs to be validated, the request is passed to the validator.

The Dynamic Variable class, as previously mentioned, is a simplified interface to a Dynamic Entity. The contents of a Dynamic Variable are shown in Fig. 12. The Dynamic Variable maintains a pointer to its Dynamic Entity. This pathway is used to get access to the data stored in the Dynamic Entity and to inform the Dynamic Entity when the data is changed. Each Dynamic Variable may optionally be linked to a Dynamic Responder. If the Dynamic Variable is linked to a Dynamic Responder, then the Dynamic Entity will send messages to that Dynamic Responder each time the data changes, is invalidated, or is validated. If the Dynamic Variable is not connected to a Dynamic Responder, then the Dynamic Entity does nothing but store the Dynamic Variable in the current reference set.

![Dynamic Variable Diagram](image)

**Figure 12 - Dynamic Variable**

A sample application using the Dynamic Integration System is graphically depicted in Fig. 13. In this example, the Dynamic Registry contains three distinct Dynamic Entities. These Dynamic Entities are linked to by Dynamic Variables residing in three modules. One Dynamic Variable each in module 1 and module 3 is linked to a Dynamic Responder. The Dynamic Responder in module 1 will receive messages each time the
second Dynamic Entity is modified by the second Dynamic Variable in module 3. The Dynamic Responder in module 3 will likewise receive messages when the Dynamic Variable in module 1 modifies the first Dynamic Entity. The third Dynamic Entity is not connected to any Dynamic Responders, so no messages will be sent if a change occurs to it.

![Diagram](image)

Figure 13 - Example Application

### 5.4 Framework Design Documents

Many ways are available to document object-oriented designs and applications. These different forms are usually specified by an object-oriented methodology. This section presents the documentation as prescribed by the Responsibility-Driven approach (see Appendix C for more information). The design documents include:
• **inheritance hierarchies.** A staple of most object-oriented documentation, inheritance hierarchies show the inheritance relationships between classes.

• **CRC (Class-Responsibility-Collaboration) cards.** This documentation states the responsibilities of each class. Contracts binding client and server classes are also specified.

• **collaborations graphs.** These graphs graphically depict the communications pathways between classes and subsystems.

• **walk-throughs.** These documents step through the design from various inputs.

### 5.4.1 Inheritance Hierarchies

Inheritance is not a major form of reuse within the Dynamic Integration System. Instantiation of classes within other classes is used more frequently.

The first place where inheritance occurs is with the **DVariable** class. The inheritance hierarchy for the **DVariable** class is shown in Fig. 14. A **DVariable** is an abstraction used by the Dynamic Integration System to represent some form of communication between a module and the Dynamic Integration System. A **DVariable** can be thought of as a plug that modules use to plug into a **DENTITY** in the **DRegistry**. Using this abstraction the Dynamic Integration System may deal with **DVariables** without specific knowledge of their implementation within a module.
Figure 14 - Dynamic Variable Class Hierarchy

The DType class inherits from the DVariable class. The DType class is a template class that implements common operators such as +, *, =, etc. Specific types inherit from the DType class. Currently these types include doubles, floats, and ints.

The only current forms of a DVariable that may be used in an application are forms that mimic data types normally found in C++ such as doubles, floats, or ints. In the future this list may be enlarged by creating new classes that inherit from the DVariable class. An example may be a structure type or an array type. These classes would supplement the DType class as direct descendants of the DVariable class.

The second place where inheritance takes place in the Dynamic Integration System is with the linked list utility classes. This is shown in Fig. 15. The DLL_Container class is a DLL_Element with the additional feature that it points to some other object, i.e. it is a linked list of containers of other objects.
Figure 15 - Linked List Class Hierarchy

The remaining classes in the Dynamic Integration System do not directly involve themselves with inheritance. Several classes are, however, designed as abstract base classes that other classes should inherit. These classes are primarily the Responder classes which include:

- DResponder
- DREsponder
- DRResponder

These classes are designed so that any class in a module may inherit from them. The class implements the desired functions that it is interested in from a list of functions present in the particular Responder. These functions are virtual functions representing events such as notify_of_change(). If the class does not implement one of the functions, then a default action occurs.

5.4.2 CRC Subsystem Cards

CRC (Class Responsibility Collaboration) cards are mechanisms for documenting and describing classes. Normally CRC diagrams are written on index cards for convenience while developing code. For purposes of more permanent storage, those cards have been transcribed into a more suitable format in this dissertation.
**BackwardReferenceSet**

**Superclasses:** none

**Subclasses:** none

**Description:** This is a set of all the DEntities a particular DEntity is dependent on.

**Contracts**

1.1 **maintain backward references**

   - **add a reference**
     
     add_ref( DEntity* )
     
     uses GSet 15.1 (manage elements)

   - **remove a reference**
     
     remove_ref( DEntity* )
     
     uses GSet 15.1 (manage elements)

   - **get access to a reference**
     
     operator[ int index ] returns DEntity*
     
     uses GSet 15.1 (manage elements)

     operator[ int index ] const returns const DEntity*
     
     uses GSet 15.1 (manage elements)

   - **retrieve number of references**
     
     get_number_of_refs() returns int
     
     uses GSet 15.1 (manage elements)

   - **know if is empty**
     
     is_empty() returns Flag
     
     uses GSet 15.1 (manage elements)

   - **know if reference is in set**
     
     is_ref_in_set( const char* name ) returns Flag
     
     uses GSet 15.1 (manage elements)

**CurrentReferenceSet**

**Superclasses:** none
Subclasses: none

Description: This is a set of DVariables linked to a particular DEntity.

Contracts

2.1 maintain current references

add a reference

add_ref (DVariable* )

uses GSet 15.1 (manage elements)

remove a reference

remove_ref (DVariable* )

uses GSet 15.1 (manage elements)

retrieve number of references

get_number_ofRefs() returns int

uses GSet 15.1 (manage elements)

know if is empty

is_empty() returns Flag

uses GSet 15.1 (manage elements)

2.2 manage event notifications

notify all references of events

notify_of_change (const DVariable* )

uses GSet 15.1 (manage elements), DVariable 13.1 (provide interface to DEntity), DResponder 10.1 (respond to events)

This function will send the notify_of_change() message to all the current references with Dynamic Responders except the Dynamic Variable that was the one that initiated the change - the DVariable argument in the call.

notify_of_validiation()

uses GSet 15.1 (manage elements), DVariable 13.1 (provide interface to DEntity), DResponder 10.1 (respond to events)

notify_of_invalidation()

uses GSet 15.1 (manage elements), DVariable 13.1 (provide interface to DEntity), DResponder 10.1 (respond to events)

notify_of_state_to_changesable()

uses GSet 15.1 (manage elements), DVariable 13.1 (provide interface to DEntity), DResponder 10.1 (respond to events)
notify_of_state_to_constrained()
uses GSet 15.1 (manage elements), DVariable 13.1 (provide
interface to DEntity), DResponder 10.1 (respond to events)

Data

Superclasses: none

Subclasses: none

Description: This class maintains a void pointer to some sort of data. The class is not
responsible for allocating or deleting the data. This is a utility class which eliminates the
need to constantly check for valid data when using the pointer.

Contracts

3.1 manage data

store a pointer

set_data( const char* type, void *data )
This function stores a void pointer and records its type. If data has been
previously been set, then an error is generated.

release the pointer

release()
This function does not delete the data, it simply breaks the connection to
the void pointer.

grant access to the pointer

get_pointer() returns void*
This function first checks to make sure the data has been set before
returning it.

know if the data has been set

is_data_set() returns Flag

know what kind of data has been set

get_data_type() returns const char*
**Ddouble, Dint, Dfloat**

**Superclasses:** DType

**Subclasses:** none

**Description:** These classes put type-specific interfaces onto the DType and DVariable classes.

**Contracts**

13.1 provide interface to DEntity
inherited from DType

13.2 maintain a link to a DResponder
inherited from DType

12.1 provide interface to data
inherited from DType

**Private Responsibilities**

*know what type it is*

**DEntity**

**Superclasses:** Named_Object

**Subclasses:** none

**Description:** A DEntity maintains the concept of a dynamic entity - the objects stored in the Dynamic Registry. All DVariables with the same name are essentially interfaces to a single DEntity. The DEntity is the underlying socket with which multiple modules communicate.

**Contracts**

16.1 manage name
Inherited from Named_Object
4.1 manage data

store the data

set_data( void*, const char* description )
uses Data 3.1 (manage data)
this function should only be called right after the DEntity is created.

grant access to the data.

get_data_pointer() returns void*
uses Data 3.1 (manage data), DEntity 4.5 (manage states)
use this function when the intent is to use the value of the data. The
DEntity will validate the data before returning the pointer

get_data_pointer_without_validating() returns void*
uses Data 3.1 (manage data), DEntity 4.5 (manage states)
use this function when the intent is to modify the data. The DEntity will
not attempt to validate the data before returning it. A check is made,
though, to make sure the DEntity is changeable.

get_data_pointer_without_validating_absolute()
returns void*
uses Data 3.1 (manage data)
use this function when the value of a constrained Dynamic Variable needs
to be set. No checking of any kind is performed. This function should be
used sparingly and with caution.

respond to changes to data

notify_of_change( const DVariable* )
uses DRegistry 9.4 (relay events to DResponders),
ForwardReferenceSet 14.2 (invalidate forward references),
CurrentReferenceSet 2.2 (manage event notifications)
The DVariable argument is the DVariable which initiated the change. The
DResponder associated with that DVariable will not receive the change
message as it is assumed already aware of the change.

know about data

is_data_set() returns Flag
uses Data 3.1 (manage data)

get_data_type() returns const char*
uses Data 3.1 (manage data)
4.2 manage dependencies

add dependencies

set_dependency( const char* name, DVariable* )
uses DRegistry 9.1 (manage DEntities)
this function first checks to make sure the name specified corresponds to a
DEntity. The DVariable argument is then checked against the Validator if
it has already been specified. Lastly a check for circular dependencies is
performed.

remove dependencies

remove_dependency( const char* name )
uses ForwardReferenceSet 14.1 (maintain forward references),
BackwardReferenceSet 1.1 (maintain backward references),
DRegistry 9.1 (manage DEntities)
first a check is made to verify that a DEntity corresponds to the
name specified.

remove_all_dependencies()
uses BackwardReferenceSet 1.1 (maintain backward references)

know about forward references

get_number_of_forward_references() returns int
uses ForwardReferenceSet 14.1 (maintain forward references)
get_forward_reference( int index ) returns const DEntity*
uses ForwardReferenceSet 14.1 (maintain forward references)

know about backward references

get_number_of_backward_references() returns int
uses BackwardReferenceSet 1.1 (maintain backward references)
get_backward_reference( int index ) returns const DEntity*
uses BackwardReferenceSet 1.1 (maintain backward references)

4.3 manage links to DVariables

add current reference

add_current_reference( DVariable* )
this function should only be used by the DRegistry class
uses CurrentReferenceSet 2.1 (maintain current references)
remove current reference
remove_current_reference( DVariable* )

uses CurrentReferenceSet 2.1 (maintain current references)
this function should only be used by the DRegistry class

know about current references
get_number_of_current_references() returns int

uses CurrentReferenceSet 2.1 (maintain current references)
is_connected() returns Flag

uses CurrentReferenceSet 2.1 (maintain current references)
this function checks to see if any current references exist

4.4 manage DEResponders

add DEResponders
add_DEResponder( DResponder* )

uses DResponderSet 6.1 (maintain DEResponders)

remove DEResponders
remove_DEResponder( DResponder* )

uses DResponderSet 6.1 (maintain DEResponders)

4.5 manage states
Note that transforming a state involves performing actions in order to change the state while switching it does not. A transformation performs the actions in addition to switching the state.

transform to a valid state
validate()

uses DVariable 13.1 (provide interface to DEntity), DResponder 10.2 (request validation), DEntity 4.1 (manage data), DEntity 4.5 (manage states)
This function will try to make an INVALID DEntity VALID by using the Validator's DResponder. If the DEntity is already VALID, then nothing happens.

transform to an invalid state
invalidate()

uses ForwardReferenceSet 14.2 (invalidate forward references), DEntity 4.5 (manage states)
This function will make a VALID DEntity INVALID and call mark_as_invalid(). If the DEntity is already INVALID, then nothing happens.
switch state to valid

mark_as_valid()

This function is called anytime the DEntity needs to set the state to VALID. This function just returns if the state is already VALID. Outside the class, the function should be used sparingly (primarily only in conjunction with the get_data_pointer_without_validating_absolute() function).

switch state to invalid

mark_as_invalid()

This function is called anytime the DEntity needs to set the state to INVALID. This function just returns if the state is already INVALID. The function should be used sparingly outside the class.

know about states

is_valid() returns Flag
get_state() returns DEntityState

Private Responsibilities

manage forward references
manage backward references
manage current references
notify DEResponderSet when triggers occur
notify DRegistry when triggers occur
notify CurrentReferenceSet when triggers occur
store a validator
Disallow circular dependencies

search_for_dependency_loop( DEntity* ) returns Flag

DEResponder

Superclasses: none

Subclasses: user defined

Description: This is an object, that once connected to a DEntity, receives messages whenever actions occur in the DEntity.
Contracts

5.1 respond to events

respond to events

notify_of_invalidation()
notify_of_validation()
notify_of_state_to_constrained()
notify_of_state_to_changeable()
notify_of_cref_add()
notify_of_cref_remove()
notify_of_bref_add( const DEntity )
notify_of_bref_remove( const DEntity )
notify_of_fref_add( const DEntity )
notify_of_fref_remove( const DEntity )

Private Responsibilities

Maintain connection with DEntity

uses DEntity 4.4 (manage DEResponders), DRegistry 9.1 (manage DEntities)

DEResponderSet

Superclasses: none

Subclasses: none

Description: This is a set of DEResponders for a particular DEntity.

Contracts

6.1 maintain DEResponders

add DEResponders

add_DEResponder( DEResponder* )

uses GSet 15.1 (manage elements)

remove DEResponders

remove_DEResponder( DEResponder* )

uses GSet 15.1 (manage elements)
6.2 notify DEResponders of events

send notifications to DEResponders

notify_of_invalidation()
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_validation()
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_state_to_constrained()
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_state_to_changeable()
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_cref_add()
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_cref_remove()
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_bref_add( const DEntity )
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_bref_remove( const DEntity )
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_fref_add( const DEntity )
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)

notify_of_fref_remove( const DEntity )
  uses GSet 15.1 (manage elements), DEResponder 5.1 (respond to events)
**DLL_Container**

**Superclasses:** DLL_Element

**Subclasses:** none

**Description:** This class is a doubly linked list of objects, each pointing to some kind of data. The class is a template class, so each container maintains the same kind of data.

**Contracts**

8.1 **general linked-list functions**

Inherited from DLL_Element

7.1 **data management**

*store the data*

DLL_Container( const T, char* group_name )

*retrieve the data*

get_object() returns T

get_container( const T ) returns DLL_Container<T>*

uses DLL_Element

this function searches the linked list for the element containing the T data specified

**DLL_Element**

**Superclasses:** none

**Subclasses:** DLL_Container

**Description:** By inheriting from this class, an object becomes a member of a doubly linked list. The class is a template class, so the subclass must specify what type it is. Members of the list may then only be of this type.
Contracts

8.1 general linked-list functions

    membership functions
    add_element_to_list( T* )
    insert_element_into_list( T* )
    remove_this_element_from_list()
    count_elements_in_list() returns int
    get_first_element() returns T*
    get_last_element() returns T*
    get_next_element() returns T*
    get_previous_element() returns T*

DRegistry

Superclasses: none
Subclasses: none

Description: Maintain a set of DEntities. For each Dynamic Identifier, there is a corresponding DEntity. This class maintains that relationship by creating and destroying DEntities as dynamic variables are added and removed from the Dynamic Integration System.

Contracts

9.1 manage DEntities

    access DEntities
    get_DEntity( int index ) returns const DEntity
    uses GSet 15.1 (maintain elements)
    get_DEntity( const char* name ) returns const DEntity
    uses GSet 15.1 (maintain elements), DEntity 16.1 (manage name)
    internal_get_DEntity( int index ) returns DEntity
    uses GSet 15.1 (maintain elements)
this is a private function used by Dynamic Integration System classes to get non-const DEntities from the DRegistry.
internal_get_DEntity( const char* name ) returns DEntity
  uses GSet 15.1 (maintain elements), DEntity 16.1 (manage name)
  this is a private function used by Dynamic Integration System classes to
  get non-const DEntities from the DRegistry.

know about DEntities
  does_DEntity_exist( const char* name ) returns Flag
    uses GSet 15.1 (maintain elements)
  get_number_of_DEntities() returns int
    uses GSet 15.1 (maintain elements)

Private Responsibilities

  maintain set of DEntities
    uses GSet 15.1 (maintain elements)

  maintain set of DRResponders
    uses GSet 15.1 (maintain elements)

9.2 register DVariables
  this contract is only to be used by DVariable

  add DVariables
    add_DVariable( DVariable*, const char* name )
    this function will create a DEntity if one does not already exist for the
    specified name
    uses GSet 15.1 (maintain elements), DRegistry 9.1 (maintain
    DEntities)

  remove DVariables
    remove_DVariable( DVariable* )
    this function will delete the DEntity if it no longer has any DVariable
    references after the one specified is removed
    uses GSet 15.1 (maintain elements), DEntity 4.3 (manage links to
    DVariables), DVariable 13.1 (provide interface to DEntity),

9.3 register DRResponders
  this contract is only to be used by DRResponder

  add DRResponders
    add_DRResponder( DRResponder* )
    uses GSet 15.1 (maintain elements)
remove DRResponders
remove DRResponder( DRResponder* )
uses GSet 15.1 (maintain elements)

9.4 relay events to DRResponders
this contract is only to be used by DEntity

relay events to DRResponders

notify_of_entry_add( const DEntity* )
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_entry_remove( const DEntity* )
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_invalidiation( const DEntity* )
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_validiation( const DEntity*)
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_state_to_constrained( const DEntity*)
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_state_to_changeable( const DEntity*)
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_change( const DEntity* )
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_dependency_add( const DEntity*, const DEntity* )
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)

notify_of_dependency_remove( const DEntity*, const DEntity* )
uses GSet 15.1 (maintain elements), DRResponder 11.1 (respond to events)
**DResponder**

**Superclasses:** none

**Subclasses:** specified by programmer

**Description:** This class may be connected to a DVariable to perform special actions in response to things that happen to the DEntity.

**Contracts**

10.1 **respond to events**

   *respond to events*
   
   `notify_of_change( const DVariable* )`
   `notify_of_invalidation( const DVariable* )`
   `notify_of_validation( const DVariable* )`
   `notify_of_state_to_constrained( const DVariable* )`
   `notify_of_state_to_changeable( const DVariable* )`

10.2 **request validation**

   *request validation*
   
   `request_validation( const DVariable* )` returns Flag

**DRResponder**

**Superclasses:** none

**Subclasses:** user-defined

**Description:** This is an object, that once connected to the DRegistry, received messages whenever actions occur in the DRegistry.
Contracts

11.1 respond to events
   
   respond to events
   
   notify_of_entry_add( const DEntity* )
   notify_of_entry_remove( const DEntity* )
   notify_of_invalidation( const DEntity* )
   notify_of_validation( const DEntity* )
   notify_of_state_to_constrained( const DEntity* )
   notify_of_state_to_changeable( const DEntity* )
   notify_of_change( const DEntity* )
   notify_of_dependency_add( const DEntity*, const DEntity* )
   notify_of_dependency_remove( const DEntity*, const DEntity* )

DType

Superclasses: DVariable

Subclasses: Dfloat, Ddouble, Dint

Description: This class inherits from the DVariable class and gives it an actual data type. The class is a template class, so this type may vary.

Contracts

13.1 provide interface to DEntity
   inherited from DVariable

13.2 maintain a link to a DResponder
   inherited from DType

12.1 provide interface to data

   access operators

   modification operators
   
   uses DVariable 13.3 (respond to changes)

   special functions
   
   get_current_value() returns T
   
   returns the value whether valid or not
set_value( const T& )
   uses DEntity 4.5 (manage states) and DVariable 13.3 (respond to changes)

sets a new value and subsequently marks valid. Used for validation outside of a request_validation() call

Private Responsibilities
 allocate data
know how to convert DEntity data to desired format
   uses DEntity 4.1 (manage data)

DVariable

Superclasses: none

Subclasses: DType

Description: Instances of this class are taken by modules to tie into the Dynamic Integration System. A DVariable acts as an interface for a DEntity. For each DVariable, there is one DEntity. For each DEntity, there is at least one DVariable.

This class serves as a type-independent place holder for the DType class. Other objects can interact with any Dynamic Variable regardless of their type by interacting with the methods of this class. All Dynamic Variables, regardless of type, inherit from this class.

Contracts
13.1 provide interface to DEntity

set dependencies
   set_dependency( const char* name )
      uses DEntity 4.2 (manage dependencies)
   set_dependency( const DVariable* )
      uses DEntity 4.2 (manage dependencies)
   set_dependency( const DVariable& )
      uses DEntity 4.2 (manage dependencies)

remove dependencies
   remove_dependency( const char* name )
      uses DEntity 4.2 (manage dependencies)
remove_dependency( const DVariable* )
uses DEntity 4.2 (manage dependencies)

remove_dependency( const DVariable& )
uses DEntity 4.2 (manage dependencies)

validate
validate()
uses DEntity 4.5 (manage states)

know state
is_valid() returns Flag
uses DEntity 4.5 (manage states)
get_state() returns DEntityState
uses DEntity 4.5 (manage states)

know name
get_name() returns const char*
uses DEntity 16.1 (manage name)

know connections
get_DEntity() returns DEntity

13.2 maintain a link to a DResponder

set connection
DVariable( const char* name, DResponder* )
a link to a DResponder can only be made once. This happens in the constructor. If a link is not specified in the constructor, it cannot be made later

know connections
get_DResponder() returns DResponder

Private Responsibilities
manage connection to a DResponder

13.3 respond to changes
this contract is only used by DType

respond to changes
use DEntity 4.1 (manage data)
**ForwardReferenceSet**

**Superclasses:** none

**Subclasses:** none

**Description:** This is a set of forward DEntity references - the DEntities that have some common DEntity as a dependent.

**Contracts**

14.1 *maintain forward references*

*add a reference*

```c
add_ref( DEntity* )
uses GSet 15.1 (manage elements)
```

*remove a reference*

```c
remove_ref( DEntity* )
uses GSet 15.1 (manage elements)
```

*get access to a reference*

```c
operator[ int index ] returns DEntity*
uses GSet 15.1 (manage elements)
```

*retrieve number of references*

```c
get_number_of.refs() returns int
uses GSet 15.1 (manage elements)
```

*know if is empty*

```c
is_empty() returns Flag
uses GSet 15.1 (manage elements)
```

14.2 *invalidate references*

*invalidate references*

```c
invalidate()
uses GSet 15.1 (manage elements)
```

this function recursively invalidates forward references
**GSet**

**Superclasses:** Named_Object

**Subclasses:** none

**Description:** this class maintains a group of elements. The class is a template class.

**Contracts**

16.1 **manage name**

inherited from Named_Object

15.1 **manage elements**

*add elements*

```cpp
add_element( T* )
```

uses DLL_Container 7.1 (data management), DLL_Container 8.1 (general linked-list functions)

*remove elements*

```cpp
remove_element( T* )
```

uses DLL_Container 7.1 (data management), DLL_Container 8.1 (general linked-list functions)

```cpp
remove_element( int index )
```

uses DLL_Container 8.1 (general linked-list functions)

```cpp
remove_all_elements()
```

uses DLL_Container 8.1 (general linked-list functions)

*access elements*

```cpp
operator[]( int index ) returns T*
```

uses DLL_Container 7.1 (data management), DLL_Container 8.1 (general linked-list functions)

*satisfy queries*

```cpp
is_element_in_set( T* ) returns Flag
```

uses DLL_Container 7.1 (data management), DLL_Container 8.1 (general linked-list functions)

```cpp
get_number_of_elements() returns int
```

uses DLL_Container 8.1 (general linked-list functions)

```cpp
is_empty() returns Flag
```

uses DLL_Container 8.1 (general linked-list functions)
**Named Object**

**Superclasses:** none

**Subclasses:** GSet

**Description:** This object has a name. It is useful for other classes to inherit from it.

**Contracts**

16.1 **manage name**

**change name**

change_name( const char* )

**get name**

get_name() returns const char*

---

**5.4.3 Collaboration Graphs**

Collaborations graphs show the communication pathways between various elements in a system. The collaborations graphs for the Dynamic Integration System are shown in Figs. 16 through 21. These diagrams vary slightly from the one specified for the Responsibility-Driven design approach. Some contract “ports” in these diagrams are shaded. These contracts are internal contracts not meant to be used by classes outside the system. Responsibility-Driven Design does not natively support private contracts. The numbers specified in each port correspond to contract numbers previously designated on the CRC cards.
Figure 16 - Dynamic Integration System Collaborations Graph
Figure 17 - DEntity Subsystem Collaborations Graph
Figure 18 - DVariable Collaborations Graph

Figure 19 - DRegistry Collaborations Graph
Figure 20 - Set Subsystem Collaborations Graph

Figure 21 - DResponder Collaborations Graph
5.4.4 Walk-Throughs

Walk-throughs are a design verification tool. A walk-through steps through the execution of a code from some specific entry point. Many walk-throughs were performed on the Dynamic Integration System before a single line of code was written. These walk-throughs were informal and undocumented. To demonstrate this procedure, this section provides formal documented walk-throughs for the following situations:

- Dfloat creation
- Dfloat destruction
- Dfloat modification
- Dfloat access
- Dfloat dependency add
- Dfloat dependency remove

A class-function pair is also provided for each step in the walk-throughs. This is normally not provided because functions usually have not been specified at this point in the design. Since this is a review partially created once the design was already completed, class-function pairs are provided for the purpose of connecting the walk-through to actual code.

It should be noted that while Dfloats were chosen for the walk-throughs, the actions performed for Dints and Ddoubles are nearly identical.
5.4.4.1 Dfloat Creation

The order of function calls is

1. DVariable Constructor
2. DType Constructor
3. Dfloat Constructor

The Dfloat constructor passes proper initialization arguments to the DType constructor which passes them to the DVariable constructor. Once the DVariable constructor is completed, the DType is constructed followed by the Dfloat construction. The following things happen starting with the DVariable constructor:

I. DVariable::DVariable(). The DResponder, if it exists, is recorded.

II. DVariable::DVariable(). The DRegistry is asked to add the DVariable.

   A. DRegistry::add_DVariable(). The DRegistry determines if a DEntity for the DVariable exists. If it doesn’t, the DRegistry creates an instance of the DEntity class, adds it to the registry, and notifies the DResponders of the addition. If the DEntity already exists, the DRegistry checks to see if it has the same data type as the DVariable. If it doesn’t, then an error is generated.

   B. DRegistry::add_DVariable(). The DRegistry adds the DVariable to the DEntity

      1. DEntity::add_current_reference(). The DEntity adds the DVariable to the CurrentReferenceSet

         a) CurrentReferenceSet::add_ref(). The CurrentReferenceSet class adds the DVariable to either its with_responders or without_responders DVariable set depending on whether the DVariable is connected to a DResponder.

      2. DEntity::add_current_reference(). The DEntity notifies the DResponders that a current reference has been added.

III. DVariable::DVariable(). The DEntity corresponding to the DVariable is retrieved from the DRegistry and recorded.
IV. **DType::DType().** If not already set (by a previous instance of the DType class), the DType constructor allocates and sets the data for the DEntity class using the type specified by the DFloat class. If data has already been set, then the type of the DVariable being constructed is verified against the existing type.

A. **DEntity::set_data().** The DEntity sets data by passing the data to the Data class.

1. **Data::set_data().** The Data class verifies that data has not previously been set and the records the type of data and saves the data.

### 5.4.4.2 DFloat Destruction

The order that destructors are called is opposite of the order of constructors. The order of function calls is:

1. DFloat Destructor
2. DType Destructor
3. DVariable Destructor

The DFloat and DType constructors do not do anything so this walk-through will start with the DVariable destructor:

1. **DVariable::~DVariable().** The DVariable destructor removes itself from the DRegistry.

   A. **DRegistry::remove_DVariable().** The DRegistry first finds the DEntity corresponding to the DVariable.

   B. **DRegistry::remove_DVariable().** The DRegistry has the DEntity remove the DVariable.

1. **DEntity::remove_current_reference().** The DEntity removes the DVariable from the CurrentReferenceSet class.

   a) **CurrentReferenceSet::remove_ref().** The CurrentReferenceSet removes the DVariable from the set it is storing it in.

2. **DEntity::remove_current_reference().** The DEntity notifies the DEResponders that a current reference has been removed.
3. **DEntity::remove_current_reference**(). If the DVariable was the DVariable that was the validator, then all dependencies are removed and the validator is set to NULL.

   a) **DEntity::remove_all_dependencies**(). All elements are removed from the BackwardReferenceSet. This involves first removing “this” DEntity from the DEntity’s ForwardReferenceSet.

   (1) **DEntity::remove_forward_reference**(). The reference is removed from the ForwardReferenceSet and the DEResponders are notified of the change.

   b) **DEntity::remove_all_dependencies**(). The second step in removing a dependence is to remove “this” from the backward references of the dependency DEntity.

   (1) **DEntity::remove_backward_reference**(). The reference is removed from the BackwardReferenceSet, and the DRegistry and DEResponders are notified of the change.

   (2) **DEntity::remove_backward_reference**(). If all dependencies have now been removed, then the DEntity is marked as valid, the validator is set to NULL, and the state is changed to CHANGEABLE.

      (a) **DEntity::mark_as_valid**(). If the DEntity is currently invalid, the state is changed to valid and the CurrentReferenceSet, DEResponders, and DRegistry are notified of the validation.

      (b) **DEntity::change_state_to_changeable**(). If the state is not currently CHANGEABLE, then it is changed to CHANGEABLE and the CurrentReferenceSet, DEResponders, and DRegistry are notified of the state change.

C. **DRegistry::remove_DVariable**(). If the DVariable no longer has any references to any DVariables, then the DEntity is removed from the DRegistry, the DEResponders are notified of the element being removed, and the DEntity is deleted.

1. **DEntity::~DEntity**(). All remaining forward and backward references are removed.

2. **DEntity::~DEntity**(). The data allocated for the DEntity is deleted.
5.4.4.3 Dfloat Modification

A Dfloat can be modified through any number of operators. These operators are all implemented in the DType class and perform roughly the same function (with respect to the Dynamic Integration System). For purposes of this walk-through we will start with the equals operator for the DType class. Walk-throughs for the other operators are similar.

I. DType::operator=(). The DType class first gets access to the data from the DEntity.
   A. DType::get_data_pointer_without_validating(). Get a type-cast pointer from the DEntity.
      1. DEntity::get_data_pointer_without_validating(). If the DEntity state is CHANGEABLE, then the data is retrieved from the Data class otherwise an error is generated.

II. DType::operator=(). If the new value is different than the current value, then the data is modified and the DVariable is informed that the data has been modified.
   A. DVariable::value_is_modified(). The DEntity is notified that the data has been modified.
      1. DEntity::notify_of_change(). If the DEntity is not currently validating itself (at which time a notify_of_change() message signifies that a new valid value has been set) then the DRegistry and CurrentReferenceSet are notified of the change and the forward references are recursively invalidated as expanded below.
         a) ForwardReferenceSet::invalidate(). Each DEntity in the set is invalidated.

(1) DEntity::invalidate(). If the DEntity is currently valid, the DEntity marks itself invalid, informs the CurrentReferenceSet, DRegistry, and DResponders of the change, and invalidates the ForwardReferenceSet.

5.4.4.4 Dfloat Access

An access operator uses the value stored by a Dynamic Variable. Modification operators also access the value, but they do so with the intent of changing it so the current validity of the value is not taken into consideration. Pure access operators such as += or > access the value with the intent of doing something with it. The value accessed must therefore
be validated before it can be returned. This process is demonstrated in the walk-through for the += operator below:

I. **DType::operator+=().** The DType gets access to the data
   
   A. **DType::get_data_pointer().** Get a type-cast pointer from the DEntity.
      
      1. **DEntity::get_data_pointer().** If the data is currently invalid, the DEntity tries to validate it
         
         a) **DEntity::validate().** Request validation from the validator. If successful, mark the DEntity as valid.

      2. **DEntity::mark_as_valid().** Notify CurrentReferenceSet, DResponders, and DRegistry of validation.

      3. **DEntity::notify_of_change().** Notify the DRegistry and CurrentReferenceSet of the change. Recursively invalidate the forward references.
         
         a) **ForwardReferenceSet::invalidate().** Each DEntity in the set is invalidated.

      1. **DEntity::get_data_pointer().** The DEntity returns data from the Data class.

II. **DType::operator+=().** If the value changed after performing the operation, then the DVariable is notified of the change.

   A. **DVariable::value_is_modified().** The DEntity is notified that the data has been modified.

      1. **DEntity::notify_of_change().** Notify the DRegistry and CurrentReferenceSet of the change. Recursively invalidate the forward references.
         
         a) **ForwardReferenceSet::invalidate().** Each DEntity in the set is invalidated.

         (1) **DEntity::invalidate().** If the DEntity is currently valid, the DEntity marks itself invalid, informs the CurrentReferenceSet, DRegistry, and DResponders of the change, and invalidates the ForwardReferenceSet.

III. **DType::operator+=().** The new modified value is returned.
5.4.4.5 Dfloat Dependency Add

Dependencies are managed in the DEntity class. They are set through the DVariable class or one of its descendants. The walk-through begins with the DVariable class:

1. **DVariable::set_dependency().** The message is passed on to the DEntity
   
   A. **DEntity::set_dependency().** If a validator has not yet been specified, then the DVariable used to set the dependency becomes the validator. If a validator has already been specified, then it is verified that the DVariable used to set the dependency is the same as the validator. If not an error is generated.

   B. **DEntity::set_dependency().** The DEntity corresponding to the dependency is retrieved from the DRegistry.

   C. **DEntity::set_dependency().** “this” is added as a forward reference to the dependency DEntity.

   1. **DEntity::add_forward_reference().** The reference is added to the ForwardReferenceSet and the DEResponders are notified of the new reference.

   D. **DEntity::set_dependency().** The dependency DEntity has “this” added to its backward references.

   1. **DEntity::add_backward_reference().** The reference is added to the BackwardReferenceSet and the DEResponders are notified of the new reference.

   2. **DEntity::add_backward_reference().** The DRegistry is notified of a dependency add.

   3. **DEntity::add_backward_reference().** The DEntity is marked as invalid.

   a) **DEntity::mark_as_invalid().** If the DEntity is currently valid, the state is changed to invalid and the CurrentReferenceSet, DEResponders, and DRegistry are notified of the invalidation. The ForwardReferenceSet is then invalidated and the state is changed to CONSTRAINED.

   (1) **ForwardReferenceSet::invalidate().** Each DEntity in the set is invalidated.

   (a) **DEntity::invalidate().** If the DEntity is currently valid, the DEntity marks itself invalid, informs the CurrentReferenceSet, DRegistry, and DEResponders of the change, and invalidates the ForwardReferenceSet.
(2) **DEntity::change_state_to_constrained()**. If the state is not currently CONSTRAINED, then it is changed to CONSTRAINED and the CurrentReferenceSet, DEResponders, and DRegistry are notified of the state change.

### 5.4.4.6 Dfloat Dependency Remove

Dependencies are managed in the DEntity class. They are removed through the DVariable class or one of its descendants. The walk-through begins with the DVariable class:

1. **DVariable::remove_dependency()**. The message is passed on to the DEntity

   A. **DEntity::remove_dependency()**. The DEntity corresponding to the dependency is retrieved from the DRegistry.

   B. **DEntity::remove_dependency()**. "this" is removed as a forward reference to the dependency DEntity.

      1. **DEntity::remove_forward_reference()**. The reference is removed from the ForwardReferenceSet and the DEResponders are notified of the change.

   C. **DEntity::remove_dependency()**. The dependency DEntity has "this" removed from its backward references.

      1. **DEntity::remove_backward_reference()**. The reference is removed from the BackwardReferenceSet, and the DRegistry and DEResponders are notified of the change.

      2. **DEntity::remove_backward_reference()**. If all dependencies have now been removed, then the DEntity is marked as invalid, the validator is set to NULL, and the state is changed to CHANGEABLE.

         a) **DEntity::mark_as_valid()**. If the DEntity is currently invalid, the state is changed to valid and the CurrentReferenceSet, DEResponders, and DRegistry are notified of the validation.

         b) **DEntity::change_state_to_changeable()**. If the state is not currently CHANGEABLE, then it is changed to CHANGEABLE and the CurrentReferenceSet, DEResponders, and DRegistry are notified of the state change.
5.5 Patterns

Patterns are a technique used to describe frameworks [John92b]. They are essentially examples used to cover a wide range of applicability for the framework. This section presents a series of patterns. These examples do not cover all possible situations, but do cover a range of the most common uses. In each example, Dynamic Variables are implemented as Dfloats. These Dynamic Variables could just as easily have been Ddoubles or Dints.

5.5.1 Creating and using a Dynamic Variable

One goal for the creation of Dynamic Data Types was to create a type that performed similarly to a regular built-in type. This involves creating common operators such as +, =, +-, etc. Once these operators are implemented, a Dynamic Variable may be used just as any other variable may be used. Figure 22 shows a sample program that creates two Dynamic Variables and performs various operations using them.

```cpp
void main()
{
    // create a Dynamic Variables
    Dfloat var1("x");
    Dint var2("y");
    double z;

    // set values for variables
    var1 = 4;
    var2 = 5;
    z = 2;

    // perform a calculation
    var1 = 2*z/var2;

    // print variables
    cout << "var1 = " << var1 << "\n";
    cout << "var2 = " << var2 << "\n";
    cout << "z = " << z << "\n";
}
```

Figure 22 - Example Dynamic Variable Usage
The Dynamic Data Types that are instantiated are a dfloat and a dint that mimic the properties of a float and int, respectively. Dynamic Variables are all created by specifying a Dynamic Identifier in the constructor. In this example, var1 uses the $x$ Dynamic Identifier and var2 uses the $y$ Dynamic Identifier. There is no correlation between the name used for a Dynamic Variable and the Dynamic Identifier specified for creating the Dynamic Variable. If a Dynamic Variable has previously been created using the Dynamic Identifier specified, then the newly created Dynamic Variable will link to it. If the data types (i.e. float or int) of the two Dynamic Variables do not match, then an error will be generated.

In this example program, a regular data type (a double) is also created. Operations are then performed on the three variables (var1, var2, and z). Except for the creation process, the usage for each of the variables does not differ.

### 5.5.2 Monitoring the value of a Dynamic Variable

A significant difference between Dynamic Variables and global variables is that a module may be made aware of changes to Dynamic Variables that occur outside the module. This can be used for any of the following reasons:

- monitoring a parameter to assure that it stays within a valid range
- monitoring a parameter to display its value on-screen
- monitoring a parameter to debug the execution of an analysis
- monitoring a parameter to keep other parameters “up-to-date” with respect to the parameter

When global data is used, none of the above scenarios are possible. The programmer is instead left guessing what happens to a global variable once control leaves a localized region.
Sample code demonstrating Dynamic Variable monitoring is shown in Figs. 23 and 24. A class called Monitor is created. This class does something every time a parameter modeled using a Dynamic Variable is modified. The Monitor class has a function called do_something_with_parameter() which will do whatever is necessary. A convenient way to extend this class is to make this function virtual.

```cpp
class Monitor : public DResponder
{
private:
    DFloat parameter;

    // INTERNAL FUNCTIONS
    void do_something_with_parameter();
    void notify_of_change( const DVariable* );

public:
    // CONSTRUCTOR
    Monitor();
};
```

**Figure 23 - Example Dynamic Variable Monitoring Class Declaration**

```cpp
Monitor::Monitor() : parameter( "x", this )
{
}

// --------------------------------------------------------
Monitor::do_something_with_parameter()
{
    // do whatever
}

// --------------------------------------------------------
Monitor::notify_of_change( const DVariable* )
{
    do_something_with_parameter();
}
```

**Figure 24 - Example Dynamic Variable Monitoring Class Definition**
The Monitor class inherits from the DResponder class because it needs to receive messages pertaining to a Dynamic Variable. The particular message that the Monitor class is interested in is the notify_of_change() message that it inherits and polymorphically implements from the DResponder class. The Monitor class only implements functions from the DResponder class in which it is interested. Functions such notify_of_invalidation() can be safely ignored if not needed.

The Monitor class also has a Dfloat data member called parameter. The parameter variable is the Dynamic Variable that is being monitored. The constructor for parameter specifies a Dynamic Identifier and a connection to the Monitor object. In this example the Dynamic Identifier is x. The connection causes messages pertinent to x to be sent to the Monitor object. If a Dynamic Variable is not connected to a Dynamic Responder, then the Dynamic Responder will not receive messages pertaining to the Dynamic Variable.

The Monitor class implements the notify_of_change() message to call the do_something_with_parameter() function. No check of the argument passed to notify_of_change() is necessary because only messages from x are sent to the Monitor object. An example where a check is necessary is shown in the next section.

**Special Note:**
The Dynamic Responder that is connected to a Dynamic Variable will not receive messages when that particular Dynamic Variable is modified. It is assumed that changes to that particular Dynamic Variable occur in a member function of a class derived from the DResponder class. Change messages are only sent to a Dynamic Responder from Dynamic Variables not connected to the Dynamic Responder.
5.5.3 Monitoring the value of multiple Dynamic Variables

This example is simply an extension of the last example. Instead of monitoring a single parameter, multiple parameters will now be monitored. This can be accomplished in one of two ways:

- create several "monitoring" objects as described in the previous section
- create a single object which monitors several parameters

In terms of efficiency, the first approach will be the fastest as fewer comparisons are needed to determine which parameter has been modified. The time spent performing these comparisons is, however, usually insignificant. The advantage of building a single class to multiple several parameters is that the total number of classes is reduced and management of the monitoring effort is decreased through centralized control.

Sample code demonstrating the process of monitoring multiple Dynamic Variables is shown in Figs. 25 and 26. A class called Monitor is created. This class is similar to the Monitor class created in the previous section.

```cpp
class Monitor : public DResponder
{  
private:
  Dfloat parameter1, parameter2, parameter3;

  // INTERNAL FUNCTIONS
  void do_something_with_parameter1();
  void do_something_with_parameter2();
  void do_something_with_parameter3();
  void notify_of_change( const DVariable* );

public:
  // CONSTRUCTOR
  Monitor();
};
```

Figure 25 - Example Multiple Dynamic Variable Monitoring Class Declaration
Figure 26 - Example Multiple Dynamic Variable Monitoring Class Definition

The implementation of the Monitor class primarily differs from the previous example in the definition of the notify_of_change() message. Since in this example the Monitor class is connected to multiple Dynamic Variables, it will receive the notify_of_change() message when any of the Dynamic Variables is modified. The Monitor object must then determine which Dynamic Variable was modified by comparing the argument passed to the notify_of_change() function and the Dynamic Variables contained in the class. After it is determined which Dynamic Variables has changed, the appropriate action may be taken.
5.5.4 Connecting a user interface element to a Dynamic Variable

Dynamic Variables represent the key parameters in a program. Integrating a user-interface element with one of those parameters involves one or both of the following:

- the ability to update the display when the parameter is modified.
- the ability to modify the parameter through the interface.

User interface elements can consist of something as simple as a number displayed on-screen to something as complex as a three-dimensional rendering of an object. In either case the interaction with Dynamic Variables is the same. When the user interface element changes a parameter, it will also change the Dynamic Variable corresponding to the parameter. When the user interface element detects a change to a Dynamic Variable, it will refresh the display. This example extends the monitoring examples by allowing the monitoring routine to also change the value of the Dynamic Variable being monitored.

Sample code demonstrating the use of Dynamic Variables is shown in Figs. 27 and 28. A class called UI_Element is created. This class displays something on-screen that is related to some parameter that is modeled as a Dynamic Variable. The UI_Element class has a function called refresh_display() which will redraw whatever is displayed on-screen. The UI_Element class also has functions (which are not shown) that handle all user-interaction. These user-interaction functions call the change_value() member function whenever they change something on-screen and want to reflect that change in the application.
class UI_Element : public DResponder
{
private:
    Dfloat parameter;

    // INTERNAL FUNCTIONS
    void refresh_display();
    void change_value( float v );
    void notify_of_change( const DVariable* );

public:
    // CONSTRUCTOR
    UI_Element();
};

Figure 27 - Example User-Interface Element Class Declaration

UI_Element::UI_Element() : parameter( "Aspect Ratio", this )
{
}

="#-------------------------------------
UI_Element::refresh_display()
{
    // do necessary stuff to update display
}

="#-------------------------------------
UI_Element::change_value( float value )
{
    parameter = value;
}

="#-------------------------------------
UI_Element::notify_of_change( const DVariable* )
{
    refresh_display();
}

Figure 28 - Example User-Interface Element Class Definition

The UI_Element class inherits from the DResponder class because it needs to receive messages pertaining to a Dynamic Variable. The particular message that the UI_Element class is interested in is the notify_of_change() message that it inherits and

Chapter 5 - Dynamic Integration System

123
polymorphically implements from the DResponder class. The UI_Element class only implements functions from the DResponder class in which it is interested. Functions such notify_of_invalidation() can be safely ignored if not needed.

The UI_Element class also has a Dfloat data member called parameter. The parameter variable is the Dynamic Variable that is displayed on-screen in some manner. The constructor for parameter specifies a Dynamic Identifier and a connection to the UI_Element object. In this example the Dynamic Identifier is Aspect Ratio. The connection causes messages pertinent to Aspect Ratio to be sent to the UI_Element object.

The UI_Element class implements the notify_of_change() message to call the update_display() function. No check of the argument passed to the notify_of_change() function is necessary because only messages from Aspect Ratio are sent to the UI_Element. When the user makes some change on-screen, the UI_Element calls the change_value() function. This function simply sets the value of parameter to the desired new value. As with the monitoring examples, multiple Dynamic Variables could be implemented in this example by adding if-else statements to the notify_of_change() message.

5.5.5 Creating an analysis using Dynamic Variables

One of the primary purposes for creating the Dynamic Integration System was to form a new method for modeling engineering analyses. This involves building a dependency hierarchy as discussed in previous sections. This section first explains how to model an equation using Dynamic Variables. The simplest method using eager-evaluation is first presented. The slightly more complicated, but much more effective lazy-evaluation technique is then presented.
5.5.5.1 *Modeling an equation*

This section will model a simple equation. The equation is:

\[
\text{area} = \text{width} \times \text{height}
\]

(1)

The purpose of modeling this equation using Dynamic Variables is so that other modules may use the computation involved in the equation without directly knowing how to execute the computation. This concept is depicted in Fig. 29. The goal is to create a module with three Dynamic Variables with the following Dynamic Identifiers: *area*, *width*, and *height*. Anytime *area* is accessed the value retrieved will be constrained to be the product of *width* and *height*. Sample code showing the desired method for interacting with *area*, *width*, and *height* is shown in Fig. 30.

![Module Integration Example](image)

*Figure 29 - Module Integration Example*
void main()
{
    // create the analysis object
    Equation analysis;

    // create the Dynamic Variables
    Dfloat var1( "area" );
    Dfloat var2( "height" );
    Dfloat var3( "width" );

    // set values for "width" and "height"
    var2 = 4;
    var3 = 5;

    // access value for "area". will print "var1 = 20"
    cout << "var1 = " << var1 << "\n";
}

Figure 30 - Example Dynamic Integration System Analysis Usage

The significance of the code in Fig. 30 is that there is never any direct interaction with the analysis. The value of area is affected only by the existence of the analysis object. The programmer using area does not have to call the analysis to determine the value of area. This makes it possible to change the implementation of the analysis without affecting the application. As a result, analysis modules may be substituted in and out with requiring changes to an application.

5.5.5.1.1 Eager-evaluation

Eager-evaluation techniques will be implemented by monitoring the values of height and width. Each time a change is detected to either height or width, the value of area is recalculated. The eager-evaluation system is diagrammed in Fig. 31.
Figure 31 - Eager-evaluation Execution Diagram

Module 2 in Fig. 31 can be represented by the code previously presented in Fig. 30. The implementation of module 1 involves two sections of code: the analysis class declaration shown in Fig. 32 and the analysis class definition shown in Fig. 33.

class Equation : public DResponer
{
private:
   // INTERNAL DATA
   Dfloat area, width, height;  // Dynamic Floats

   // INTERNAL FUNCTIONS
   // inherited from DResponder
   void notify_of_change( const DVariable* );

public:
   // CONSTRUCTOR
   Equation::Equation();
};

Figure 32 - Example Eager-evaluation Analysis Class Declaration

The Equation class declared in Fig. 32 follows the patterns developed by the monitoring examples. The Equation class therefore inherits from the DResponder class and

Chapter 5 - Dynamic Integration System
instantiates three Dfloats. The notify_of_change() message is also polymorphically implemented from the DResponder class.

```
Equation::Equation() :
    area("area"), // don't connect
    height("height", this), // connect
    width("width", this) // connect
{
    area = height * width; // set the initial value for area
}
```

```
void Equation::notify_of_change( const DVariable *var )
{
    area = height * width; // recalculate area
}
```

Figure 33 - Example Eager-evaluation Analysis Class Definition

The definition of the Equation class, as shown in Fig. 33, first creates three Dfloats. The Dfloats (area, height, and width) are given the Dynamic Identifiers area, height, and width. Only the height and width Dynamic Variables are connected to the Equation object. The area Dynamic Variable is not connected because the Equation class is not interested in receiving messages pertaining to it. The only messages the Equation class is interested in are the notify_of_change() messages when applied to width or height. When these messages are received, the value of area is recalculated.

The implementation of the notify_of_change() message does not perform any checking on the Dynamic Variable argument passed to it. This is because the message will only be called when width or height has changed. Implementing an Equation class that evaluates multiple equations involves modifying the notify_of_change() function to determine which Dynamic Variable has changed in a manner similar to the one described for monitoring multiple Dynamic Variables.
Special Note:
Infinite loops can easily occur using eager-evaluation. An infinite loop will occur in the above example if for some reason width or height was modified by some other object in response to a change in area. The change to area causes a change to width or height which in turn results in a change to area and so on... It is the responsibility of the programmer to avoid infinite loops when using eager-evaluation. Lazy-evaluation techniques trap and disallow loops.

5.5.5.1.2 Lazy-evaluation
The disadvantage to eager-evaluation is that many unnecessary calculations are performed. In the example being presented this can be seen in the calculation of area. When the values of height and width are both changed, area is calculated twice - once for each change in height and width. The result of one of these calculations is never used and therefore is redundant. The solution to eliminating this extra computation is to use lazy-evaluation - calculate area only when it absolutely needs to be calculated.

Lazy-evaluation is implemented by making height and width dependencies of area. Subsequent modifications to height and width will then cause area to be marked invalid. Whenever an attempt to access the value of area is then detected, the Dynamic Integration System will send a message to a Dynamic Responder associated with the Dynamic Variable to validate itself. This process is diagrammed in Fig. 34.
Figure 34 - Lazy-evaluation Execution Diagram

The source code used to implement module 1 is shown in Figs. 35 and 36.

```cpp
class Equation : public DResponder
{
    private:
        // INTERNAL DATA
        Dfloat area, width, height; // Dynamic Floats

        // INTERNAL FUNCTIONS
        // inherited from DResponder
        Flag request_validation( const DVariable* );

    public:
        // CONSTRUCTOR
        Equation::Equation();
};
```

Figure 35 - Example Lazy-evaluation Analysis Class Declaration
Equation::Equation():
  area( "area", this ), // connect
  height( "height" ), // don't connect
  width( "width" ) // don't connect
{
  area.set_dependency( height );
  area.set_dependency( width );
}

//........................................................................
Flag Equation::request_validation( const DVariable *var )
{
  area = height * width; // recalculate area
  return True;
}

Figure 36 - Example Lazy-evaluation Analysis Class Definition

The declaration of the Equation class differs from the eager-evaluation implementation only in that the notify_of_change() message has now been replaced with the request_validation() message. This Equation class is not concerned with changes to Dynamic Variables, instead its purpose is to know how to calculate the value of area when the Dynamic Integration System needs it.

The definition of the Equation class, shown in Fig. 36, first creates three Dynamic Variables in the constructor. This is similar to the previous eager-evaluation example except that this time only the area Dynamic Variable is connected to the Equation object. The height and width Dynamic Variables are not connected to anything. The constructor also differs in the fact that the set_dependency() function is used to make area dependent on width and height.

Once dependencies have been set, the request_validation() message can be put to use. This function will be called by the Dynamic Integration System whenever area needs to be validated. area is invalidated whenever changes to width or height occur. In more general terms, a Dynamic Variable is invalidated any time one of its dependencies is
modified. The Dynamic Integration System tries to validate \texttt{area} whenever it is invalid and an attempt to access its value is detected. If the validation process is successful, then a flag indicating so is returned. If for some reason the Dynamic Variable cannot be validated, \texttt{false} (0) should be returned instead of \texttt{true} (1).

Just as the monitoring classes could monitor more than one Dynamic Variable at a time, the Equation class can be used to maintain several lazy-evaluation equations. The required changes mainly involve implementing an \texttt{if-else} branch in the \texttt{request_validation()} member function. In the described example, the \texttt{request_validation()} message will only be sent to the Dynamic Responder when \texttt{area} needs to be validated. This is because only \texttt{area} is connected to the Dynamic Responder.

\textbf{Special Notes:}

1. Once a dependency is set for a Dynamic Variable, the state of that Dynamic Variable changes from CHANGEABLE to CONSTRAINED if it is not already CONSTRAINED. This means that the value of Dynamic Variable \texttt{cannot} be set anywhere other than from within the \texttt{request_validation()} call. If an attempt to change the value of a CONSTRAINED Dynamic Variable is detected outside a \texttt{request_validation()} call for the Dynamic Variable then an error will be generated.

2. Each time a dependency is added to a Dynamic Variable, the Dynamic Variable is marked INVALID. This eliminates the need to give a Dynamic Variable with dependencies an initial value - it will be requested when needed. Once all dependencies have been removed from a Dynamic Variable, that Dynamic Variable will be marked VALID if it was INVALID.

3. For each Dynamic Variable group (i.e. all Dynamic Variables sharing a common Dynamic Identifier) there can be only one Dynamic Responder responsible for validating the group. The Dynamic Variable from the group associated with this Dynamic Responder is referred to as the \texttt{validator} Dynamic Variable. The first
Dynamic Variable from a Dynamic Variable group to set a dependency becomes the validator for that group. Subsequent dependencies can only be set or removed from the validator Dynamic Variable.

5.5.6 Connect existing C or FORTRAN code to Dynamic Variables

The Dynamic Integration System has been implemented so that existing code can be utilized. The idea behind using legacy code is to build wrappers around the existing functions using Dynamic Variables and Dynamic Responders. The Dynamic Variables represent the parameters for the function while the Dynamic Responder is used to execute the function. This section presents methods for implementing an existing routine using both eager and lazy evaluation techniques.

The existing analyses are assumed to have one of the following formats:

\[
\text{output} = \text{func( input1, input2, ... )}
\]

or

\[
\text{func( input1, input2, ..., output1, output2, ... )}
\]

Existing analysis modules that rely heavily on global variables for input or output need to be reformatted into a combination of the above styles.

5.5.6.1 Eager-evaluation

The eager-evaluation implementation varies only slightly from a native C++ implementation. In fact, the class declaration for a class as shown in Fig. 37 does not vary from the implementation shown in Fig. 33.
class Equation : public DResponder
{
private:
   // INTERNAL DATA
   Dfloat    area, width, height; // Dynamic Floats

   // INTERNAL FUNCTIONS
   // inherited from DResponder
   void notify_of_change( const DVariable*);

public:
   // CONSTRUCTOR
   Equation::Equation();
};

Figure 37 - Example Eager-evaluation Legacy Analysis Class Declaration

The Equation class implements three Dfloats as before and polymorphically implements the notify_of_change() function. The definition for the Equation class is shown in Fig. 38.

float get_area( float, float ); // define external functions

//-------------------------------------------------------------------------------
Equation::Equation()
      :
        area( "area" ), // don't connect
        height( "height", this ), // connect
        width( "width", this ) // connect
      {
         area = get_area( height, width ); // set the initial value for area
      }

//-------------------------------------------------------------------------------
void Equation::notify_of_change( const DVariable *var )
      {
         area = get_area( height, width ); // recalculate area
      }

Figure 38 - Example Eager-evaluation Legacy Analysis Class Definition

This definition of the Equation class is similar to the definition of a C++ implementation. The difference is that the actual computations required to calculate area are executed in a different module and are accessed via the get_area() function.

Chapter 5 - Dynamic Integration System
It should be noted that slight variations occur when the function is implemented in FORTRAN. Arguments passed to FORTRAN functions are of type reference to something where the something is usually either a float, double, or int. While Dynamic Variables behave as built-in types, they are not identical. As a result it is not possible to take a reference to a Dynamic Variable of the type desired. A simple work-around is to use dummy variables as shown in the new implementation of the request_validation() function shown in Fig. 39.

```c
float get_area( float&, float& ); // define external functions

void Equation::notify_of_change( const DVariable *var )
{
    float dummy1 = height;
    float dummy2 = width;
    area = get_area( dummy1, dummy2 ); // recalculate area
}
```

**Figure 39 - Example notify_of_change() Alternate Implementation #1**

A similar problem occurs in C or FORTRAN functions that attempt to return results as arguments. Again, the same work-around applies as shown in Fig. 40.

```c
void get_area( float&, float&, float& ); // define external functions

//-----------------------------------------------------------------------
void Equation::notify_of_change( const DVariable *var )
{
    float dummy1 = height;
    float dummy2 = width;
    float dummy3;
    get_area( dummy1, dummy2, dummy3 ); // recalculate area
    area = dummy3;
}
```

**Figure 40 - Example notify_of_change() Alternate Implementation #2**

*Chapter 5 - Dynamic Integration System*
5.5.6.2 Lazy-evaluation

Lazy-evaluation techniques are applied to existing functions the same way eager-evaluation techniques are. The class declaration is shown in Fig. 41.

```cpp
class Equation : public DResponder
{
private:
   // INTERNAL DATA
   Dfloat area, width, height;       // Dynamic Floats

   // INTERNAL FUNCTIONS
   // inherited from DResponder
   Flag request_validation( const DVariable* );

public:
   // CONSTRUCTOR
   Equation(): Equation();
};
```

**Figure 41 - Example Lazy-evaluation Legacy Analysis Class Declaration**

The `Equation` class again implements three Dynamic Variables and the `request_validation()` function. The `Equation` class definition, shown in Fig. 42 is very similar to the eager-evaluation implementation where `notify_of_change()` is replaced by `request_validation()`.
float get_area( float, float ); // define external functions

//---------Equation():
  area( "area", this ), // connect
  height( "height" ), // don't connect
  width( "width" ) // don't connect
{
  area.set_dependency( height );
  area.set_dependency( width );
}

//-------------------------------
Flag Equation::request_validation( const DVariable *var )
{
  area = get_area( height, width ); // recalculate area
  return True;
}

Figure 42 - Example Lazy-evaluation Legacy Analysis Class Definition

The problems that apply to the eager-evaluation implementation, namely the inability to take float references to Dynamic Variables, also apply to the lazy-evaluation implementation. Again, the same work-around of using dummy variables also applies here.

An additional problem that presents itself with lazy-evaluation but not eager-evaluation is the problem of validating two Dynamic Variables at once. Consider the following function:

get_params( width, height, area, perimeter )

The function is similar to the previously used get_area() function except that a perimeter value is returned in addition to the area parameter. Modeling these parameters would involve the following Dynamic Variables:
• width
• height
• area
• perimeter

width and height are both dependencies of area and perimeter. Changing either width or height causes both area and perimeter to be invalidated. Subsequent requests for either area or perimeter require that the `get_params()` function be executed. While we are only validating one Dynamic Variable, a valid value for the other is also simultaneously being calculated. Since we are not currently validating the other Dynamic Variable, it is not possible to specify a new value for it using the `=` operator. For special circumstances like this, the `set_value()` function has been created. The `set_value()` function sets the value of a Dynamic Variable and marks the Dynamic Variable as valid regardless of whether its dependencies are valid or not. An implementation of the `notify_of_change()` function using the `set_value()` function is shown in Fig. 43.
void get_params( float&, float&, float&, float& ); // define external functions

//..............................................................................
void Equation::request_validation( const DVariable *var )
{
    float dummy1 = height;
    float dummy2 = width;
    float dummy3;
    float dummy4;
    get_params( dummy1, dummy2, dummy3, dummy4 ); // recalculate area & perimeter

    // determine which validation request this is for
    if ( var == &area )
    {
        area = dummy3;
        perimeter.set_value( dummy4 );
    }
    else if ( var == &perimeter )
    {
        perimeter = dummy4;
        area.set_value( dummy3 );
    }

    return True;
}

Figure 43 - Example request_validation() Alternate Implementation

When validating several parameters at once, the = operator can be used only for the Dynamic Variable that is currently being validated. This can be safely accomplished by testing the Dynamic Variable parameter passed to the request_validation() function. An alternate approach is to just use the set_value() function for both the area and perimeter Dynamic Variables. This eliminates the need to determine which Dynamic Variable is being validated.

5.5.7 Performing iteration

Iteration occurs when a dependency loop forms. A dependency loop forms when it is necessary to have the output of an equation to form one of the inputs. A graphical depiction of this concept is shown in Fig. 44.
Figure 44 - Cyclic Dependency

An example of iteration occurs in the aircraft industry when the weight of an aircraft is determined. Simplifying the analysis, the following equations can be used:

\[
W_{\text{total}} = W_{\text{wing}} + W_{\text{fuel}} + W_{\text{engine}} + \ldots
\]  

(2)

\[
W_{\text{wing}} = 0.0051K_S S_w^{0.649} S_{CSW}^{0.01} \frac{AR^{0.5}}{(t/c)_{\text{root}}} \frac{(NW_{\text{total}})^{0.557}(1+\lambda)^{0.1}}{\cos \Lambda_{25}}
\]  

(3)

The first equation calculates the weight of the aircraft by summing the individual components. The second equation calculates the weight of one of the components - the wing. Unfortunately, the calculation for the wing is based on the total weight of the aircraft. Thus, changing the weight of the wing changes the weight of the aircraft, which again causes the weight of the wing to be changed. This process continues until a consistent set of values is found.

Iteration can be handled by introducing a new parameter - \((W_{\text{total}})_{\text{est}}\). This parameter replaces the \(W_{\text{total}}\) parameter of the second equation. By taking an initial estimate of the total weight, we can calculate \(W_{\text{wing}}\). From there, \(W_{\text{total}}\) can be formed. If \(W_{\text{total}}\) is close to \((W_{\text{total}})_{\text{est}}\), then the estimate was a good one. If the two values do not match, then a new estimate should be formed until one can be found that results in the satisfactory calculation of \(W_{\text{total}}\).
A dependency hierarchy using an estimated value to simulate a cyclic dependency is shown in Fig. 45. The estimated value parameter is introduced as a dependency of the parameter that was formerly dependent on the actual parameter. The estimated value is then constrained to the actual value. An iterator object will be introduced to manage this process.

![Cyclic Dependency Simulated](image)

**Figure 45 - Simulated Cyclic Dependency**

This iteration process can be demonstrated with actual code modeling a simple analysis. Consider modeling the following two equations:

\[ x = y^2 \]  \hspace{1cm} (4)  
\[ y = \frac{1}{2} x \]  \hspace{1cm} (5)

Using some fairly simple mathematics, this system can be solved to find that \( x \) equals four and \( y \) equals 2. Assuming this system cannot be solved directly, such as in the case of the aircraft weight analysis problem, we will approach it for demonstration purposes using an iterative numerical technique. The first step is to redefine the second equation to the following form:
\[ y = \frac{1}{2} x_{est} \]  
(6)

This modified form removes any dependency loops. The goal now will be to try to keep \( x \) and \( x_{est} \) equal to one another. This will be accomplished by creating an object that monitors changes to \( x \). Since \( x \) is a constrained Dynamic Variable, the only time it will be changed is during a validation process by the Dynamic Integration System. During this time we will change \( x_{est} \) each time the Dynamic Integration System forms a new value for \( x \) until the two values are equal. The Dynamic Integration System will then return the correct, iterated value for \( x \) to the application.

The class declaration for an iteration object is shown in Fig. 46. This class will maintain two Dynamic Variables: one for \( x \) and one for \( x_{est} \). These two Dynamic Variables will be constrained to one another using some epsilon tolerance. Iteration will also be limited to some specified number of cycles.
class Iterator : public DResponder
{
private:
  Dfloat x;
  Dfloat x_est;
  float epsilon;
  int iteration_count;
  int last_iteration_count;
  int iteration_limit;

  // INTERNAL FUNCTIONS
  void make_new_guess();

  // Inherited from DResponder
  void notify_of_change( const DVariable* );

public:
  // CONSTRUCTOR
  Iterator();

  // MEMBER FUNCTIONS
  int get_iteration_count() { return last_iteration_count; }
};

Figure 46 - Example Iterative Analysis Class Declaration

The definitions for the functions in the Iterator class are shown in Fig. 47.
Figure 47 - Example Iterative Analysis Class Definition

The constructor initializes values for the iteration limit and the tolerance. The `notify_of_change()` function is the main part of the `Iterator` class. This function is called each time the Dynamic Integration System calculates a new value for $x$. The function first examines whether the difference between $x$ and $x_{est}$ is greater than epsilon. If it is, then a new value for $x_{est}$ is calculated. The actual formation of the new estimate is
performed in the `make_new_guess()` function shown in Fig. 48. A more sophisticated iteration routine would revise this routine to make a more educated guess.

```cpp
void Iterator::make_new_guess()
{
    // A more sophisticated algorithm can be substituted here
    x_est ← (x_est - x)/10.0;
}
```

**Figure 48 - Sample Stepping Algorithm**

The process of changing the value of $x_{est}$ in the `make_new_guess()` function results in the invalidation of $x$. Since the Dynamic Integration System is currently trying to validate $x$, it will try to calculate a new value again. Changing to this new value will cause the `notify_of_change()` function to be called again. This process will repeat until the `notify_of_change()` function no longer modifies $x_{est}$.

Code for using the Iterator class is shown in Fig. 49. The program first instantiates the necessary analysis objects and an instance of the Iterator class. The proper Dynamic Variables are then created. Iteration is first approached by specifying an initial guess for $x_{est}$. An attempt to access $x$ via a print statement then triggers the iteration algorithm. The values that are then printed reflect the values that result after iteration.
void main()
{
    // create the equation and iteration objects
    Equation1  eq1;
    Equation2  eq2;
    Iterator   iter;

    // create the Dynamic Variables
    DFloat   x( "x" );
    DFloat   x_est( "estimated x" );
    DFloat   y( "y" );

    // set an initial guess
    x_est = 2;

    // print out values (and perform the iteration in the process)
    cout << "x = " << x << "n";
    cout << "y = " << y << "n";
    cout << "x_est = " << x_est << "n";
    cout << "Converged in " << iter.get_iteration_count() << " cycles\n";
    cout.flush();
}

Figure 49 - Example Iterative Analysis Sample Program

The output from the sample program is shown in Fig. 50. As shown, for the given iteration criteria and method, the system converged in 102 cycles to a close approximation of the actual values of four and two.

x = 3.999814
y = 1.999953
x_est = 3.999907
Converged in 102 cycles

Figure 50 - Example Iterative Analysis Sample Program Output

The iteration process, as viewed by the Dynamic Integration System, is shown in Fig. 51. When an attempt is made to access the actual value parameter, x in the previous example, a validation process is started if the parameter is invalid. This validation process starts by requesting, from the proper Dynamic Responder, that a new value for the Dynamic Integration System
Variable be calculated. If this request if fulfilled, then a new value will be calculated for the Dynamic Variable. This will cause an iterator object to receive the `notify_of_change()` message. If the difference between the actual and estimated values is greater than desired, a new value is specified for the estimated value by the iterator object. This change to the estimated value results in the invalidation of all forward references - including the actual value. Once the invalidation process completes, control returns to the access operator. The access operator tests the validity of the actual value and, if it is valid, returns the value. If the value is invalid, due to a change in the estimated value and resulting invalidation processes, then the validation process is restarted. Validation ends once the iterator object stops making changes to the estimated value - the access operator will be able to return a valid value.
Actual Value: access operator
An access operator initiates a validation process. This involves calling the validation() function until a valid value is returned.

Actual Value: invalidate()
The actual value is marked invalid.

The iteration process starts with the access operator and continues until the Iterator object does not set a new value for the estimated value. At that time, the access operator will be able to return a valid value.

Actual Value: validate()
The validate() function validates a Dynamic Variable using the request validation() function from a Dynamic Responder. Once validated, a notify_of_change() message is propagated.

Estimated Value: value_has_changed()
If the Iterator changed the estimated value, then the value_has_changed() message is propagated.

Iterator Object: notify_of_change()
The Iterator object picks up the notify_of_change() message and if needed, sets a new value for the estimated value.

Figure 51 - Iteration from the Dynamic Integration System viewpoint

Special Note:
Iteration only occurs when accessing the value that is being governed by the iterating object. In the above example this means that iteration will only occur when x is accessed. An access to y will not trigger iteration. Instead a value will be returned based on the current value of x_{est}. If the value for y is needed before x is accessed anywhere, iteration can be requested by manually validating x. This is accomplished with the validate() function that may be applied to any Dynamic Variable.
5.5.8 Obtaining parameters related to a Dynamic Variable

The state of a Dynamic Variable can easily be determined by simply querying the Dynamic Variable. The code shown in Fig. 52 demonstrates how this can be accomplished.

```cpp
void main()
{
    // create a Dynamic Variable
    Dfloat varl( "area" );

    // create variables which will be used to store responses to queries
    Flag valid_state;
    DEntityState constraint_state;
    float value;

    // query the Dynamic Variable
    valid_state = varl.is_valid();
    constraint_state = varl.get_state();
    value = varl.get_current_value();

    // create pointers
    DResponder *dresponder;
    DEntity *dentity;

    // get references
    dresponder = varl.get_DResponder();
    entity = varl.get_DEntity();

    // more advanced queries can be performed from the DEntity
    cout << "num dependencies = " << dentity->get_number_of_backward_references() ;
}
```

Figure 52 - Example Dynamic Integration System Analysis Usage

The get_current_value() function differs from a normal access operator in that no validation process is involved. If the Dynamic Variable is invalid, then the invalid value is what will be returned. References associated with a Dynamic Variable, such as a DResponder or DEntity can also be obtained from a Dynamic Variable. A DEntity is useful for performing more advanced queries pertaining to a Dynamic Variable such as information about forward and backward references.
5.5.9 **Monitoring the state of a Dynamic Variable**

Sometimes it is necessary to perform actions when a Dynamic Variable changes state. Examples of this include:

- If a geometric display associated with a Dynamic Variable is very time consuming to draw, it may be desirable to only redraw it when the user requests. Monitoring when a Dynamic Variable becomes invalid is a way to let the user know that the drawing needs to be redisplayed.

- If the value of a parameter usually interactively specified via a user interface is sometimes internally calculated by an analysis, then it is necessary to alter the user interface to indicate that the value is being calculated internally. This is usually graphically shown by desensitizing a control. The desensitizing action may be set up to occur when the Dynamic Variable shifts between the CHANGEABLE and CONSTRAINED states indicating the presence of an analysis routine constraining the Dynamic Variable.

Code associated with monitoring the state of a Dynamic Variable is shown in Figs. 53 and 54.
class StateMonitor : public DResponder
{
private:
    // INTERNAL DATA
    float area; // Dynamic Floats

    // INTERNAL FUNCTIONS
    // inherited from DResponder
    void notify_of_validation( const DVariable* );
    void notify_of_invalidation( const DVariable* );
    void notify_of_state_to_constrained( const DVariable* );
    void notify_of_state_to_changeable( const DVariable* );

public:
    // CONSTRUCTOR
    StateMonitor::StateMonitor();
};

Figure 53 - Example Dynamic Variable State Monitoring Class Declaration

StateMonitor::StateMonitor():
    area( "area", this ) // connect
{
}

//-----------------------------------------------
void StateMonitor::notify_of_validation( const DVariable *var )
{
    // do whatever
}

//-----------------------------------------------
void StateMonitor::notify_of_invalidation( const DVariable *var )
{
    // do whatever
}

//-----------------------------------------------
void StateMonitor::notify_of_state_to_constrained( const DVariable *var )
{
    // do whatever
}

//-----------------------------------------------
void StateMonitor::notify_of_state_to_changeable( const DVariable *var )
{
    // do whatever
}

Figure 54 - Example Dynamic Variable State Monitoring Class Definition

Chapter 5 - Dynamic Integration System
5.5.10 Monitoring actions occurring to a Dynamic Entity

Sometimes it is necessary to monitor even more information about a Dynamic Variable such as when dependencies are added or removed. This type of information cannot be monitored from a Dynamic Responder. A special class called a Dynamic Entity Responder is utilized for this purpose. Sample code for using a Dynamic Entity Responder is shown in Figs. 55 and 56. The code parallels that of Dynamic Variables and Dynamic Responders.

```
class DEntityMonitor : public DEResponder
{
private:
   // INTERNAL FUNCTIONS
   // inherited from DEResponder
   void notify_of_validation();
   void notify_of_bref_add( const DEntity* );
   void notify_of_state_to_changeable();

public:
   // CONSTRUCTOR
   DEntityMonitor : DEntityMonitor( const char* dynamic_identifier );
};
```

Figure 55 - Example Dynamic Entity Monitor Class Declaration
Figure 56 - Example Dynamic Entity Monitor Monitoring Class Definition

In this example three actions are being monitored for a particular Dynamic Entity. These three actions are a subset of the following actions that may be monitored:

- `notify_of_invalidation()`
- `notify_of_validation()`
- `notify_of_state_to_constrained()`
- `notify_of_state_to_changeable()`
- `notify_of_cref_add( const DEntity* )`
- `notify_of_cref_remove( const DEntity* )`
- `notify_of_bref_add( const DEntity* )`
- `notify_of_bref_remove( const DEntity* )`
- `notify_of_fref_add( const DEntity* )`
- `notify_of_fref_remove( const DEntity* )`
5.5.11 Monitoring action occurring in the Dynamic Registry

Just as the actions pertaining to a particular Dynamic Entity can be monitored, actions pertaining to all Dynamic Entities can be monitored via the Dynamic Registry and a Dynamic Registry Responder. Figures 57 and 58 show how to implement a Dynamic Registry Responder to monitor actions occurring to Dynamic Entities in the Dynamic Registry.

```cpp
class DRegistryMonitor : public DEResponder
{
private:
    // INTERNAL FUNCTIONS
    // inherited from DEResponder
    void notify_of_entry_add( const DEntity* );
    void notify_of_validation( const DEntity* );

public:
    // CONSTRUCTOR
    DRegistryMonitor:: DRegistryMonitor();
};
```

**Figure 57 - Example Dynamic Registry Monitor Class Declaration**

```cpp
DRegistryMonitor:: DRegistryMonitor() : DEResponder()
{
}

//---------------------------------
void DRegistryMonitor:: notify_of_entry_add( const DEntity *dentity )
{
    // do whatever
}

//---------------------------------
void DRegistryMonitor:: notify_of_validation( const DEntity *dentity )
{
    // do whatever
}
```

**Figure 58 - Example Dynamic Registry Monitor Monitoring Class Definition**
In this example two actions are being monitored in the Dynamic Registry. These two actions are a subset of the following actions that may be monitored:

- notify_of_entry_add( const DEntity* )
- notify_of_entry_remove( const DEntity* )
- notify_of_invalidation( const DEntity* )
- notify_of_validation( const DEntity* )
- notify_of_state_to_constrained( const DEntity* )
- notify_of_state_to_changeable( const DEntity* )
- notify_of_dependency_add( const DEntity*, const DEntity* )
- notify_of_dependency_remove( const DEntity*, const DEntity* )

5.5.12 How to model a bi-directional constraint

Dependency hierarchies formed from Dynamic Variables are based on unidirectional constraints. This system assumes that certain parameters will always be input parameters and others will be output parameters. Sometimes it is convenient to vary which parameters are input parameters based on which values the user can specify. This is a situation where full-blown constraint satisfiers are useful.

A constraint satisfaction module is not provided by the Dynamic Integration System, so bi-directional constraints cannot natively be expressed. Instead, dependency hierarchies can be manually inverted to simulate bi-directional constraints.

As an example of the need for a bi-directional constraint, consider the following relation:

\[ a = b + c \]

This equation can be interpreted as any of the following equations
\[
\begin{align*}
    a &= b + c \\
b &= a - c \\
c &= a - b
\end{align*}
\]

Depending on which parameters are known, one of the equations can be used to solve for the third parameter. Using Dynamic Variables, a system can be designed that activates one and only one of those equations at a time. The methodology is to create three classes, one for modeling each equation. Another class is then created for managing which of the three previous classes is active.

The three classes used to model each of the equations follow the same format that has been previously described. The classes are called AConstraint, BConstraint, and CConstraint. The AConstraint class is shown in Fig. 59. The code for the BConstraint and CConstraint classes is not shown, but is very similar to the code for the AConstraint class, differing only in which relation is modeled.
```cpp
class AConstraint : public DResponder
{
private:
    Dfloat      a, b, c;

    // INTERNAL FUNCTIONS
    Flag          request_validation( const DVariable* );

public:
    AConstraint();
};

//--------------------------------------------------------------------------------
AConstraint::AConstraint() : a( "A", this ), b( "B" ), c( "C" )
{
    z.set_dependency( b );
    a.set_dependency( c );
}

//--------------------------------------------------------------------------------
Flag AConstraint::request_validation( const DVariable *_var )
{
    a = b + c;
    return YES;
}
```

**Figure 59 - Sample Constraint Class**

The class that manages the AConstraint, BConstraint, and CConstraint classes is shown in Fig. 60. The class maintains a pointer to one of the three constraint classes. Each time a new constraint is to be activated, the old constraint is deleted and the new one put in its place. Constraints can be activated with the constrain_A(), constrain_B(), and constrain_C() functions.
class Constraint : public DResponder
{
private:
    DResponder *active_constraint;

public:
    // CONSTRUCTOR
    Constraint();

    // DESTRUCTOR
    ~Constraint();

    // MEMBER FUNCTIONS
    constrain_A();
    constrain_B();
    constrain_C();
};

Figure 60 - Bi-directional Constraint Class Declaration

The definition for the Constraint class is shown in Fig. 61. The constructor creates a AConstraint object and makes it the active constraint by default. The destructor deletes the active constraint. The member functions replace the active constraint with either an AConstraint, BConstraint, or CConstraint object.
Figure 61 - Bi-directional Constraint Class Definition

The Constraint class is put to use in Fig. 62. Three Dynamic Variables corresponding to $A$, $B$, and $C$ are first created. The constraint object is then created. Values may then be specified for $B$ or $C$. $A$ is automatically constrained. To make $B$ constrained, the `constrain_B()` function is called.
```c
int main()
{
    // create dynamic variables
    Dfloat a( "A" );
    Dfloat b( "B" );
    Dfloat c( "C" );

    // create the constraint object
    Constraint constraint;

    // set initial values
    b = 2;
    c = 5;

    // print out values
    cout << "( a, b, c ) = ( " << a << ", " << b << ", " << c << " )\n";
    cout.flush();

    // switch the active constraint
    constraint.constrain_B();

    // set a new value
    c = 10;

    // and print out values again
    cout << "( a, b, c ) = ( " << a << ", " << b << ", " << c << " )\n";
    cout.flush();
}
```

Figure 62 - Bi-directional Constraint Sample Program

The output from the sample program is shown in Fig. 63.

```
( a, b, c ) = ( 7, 2, 5 )
( a, b, c ) = ( 7, -3, 10 )
```

Figure 63 - Sample Program Output
5.5.13 How to create a new Dynamic Data Type

The Dynamic Data Types provided in this dissertation model built-in data types such as floats or ints. In a more complex program, it may be desirable to model a more sophisticated data type. This section demonstrates how to make a standard type into a Dynamic Data Type. The data type used in this example will model a point consisting of x, y, and z coordinates.

The first job in creating a Dynamic Data Type is determining what data needs to be stored and then building a structure to model that data. For our example a point is being modeled. A simple class maintaining x, y, and z coordinates is therefore created as shown in Fig. 64. This class is rather simple and only serves as a structure to maintain the data. If desired, a more complex class could be created that implements constructors, member functions, etc.

```cpp
class DPointData
{
public:
    float x, y, z;
};
```

**Figure 64 - DPointData Class**

The next step is to create a class representing the Dynamic Data Type. This is accomplished by inheriting from the DVariable class. The class that will be created for this example is called DPoint. The class declaration for DPoint is shown in Fig. 65.
class DPoint : public DVariable
{
private:
    // INTERNAL UTILITY FUNCTIONS
    // for getting data from DEntity
    inline DPointData* get_data();
    inline DPointData* get_data_without_validating();

public:
    // CONSTRUCTOR
    DPoint( const char *name, DResponder*_dres = NULL );

    // MEMBER FUNCTIONS
    void set_x( float );
    void set_y( float );
    void set_z( float );
    float get_x();
    float get_y();
    float get_z();

    // for printing
    friend ostream& operator<<( ostream &os, const DPoint& _point );
};

Figure 65 - DPoint Class Declaration

The DPoint class provides a basic set of functions. A constructor similar to the
constructors of other Dynamic Data Types is provided. Member functions for accessing
and modifying the coordinates of the point along with a friend function for printing the
coordinates of the class are also included. Two internal utility functions (get_data() and
get_data_without_validating()) are implemented for retrieving the data from the
dEntity class.

The constructor is probably the most complex function for a Dynamic Data Type. The
constructor body for the DPoint class is shown in Fig. 66.
DPoin::DPoin( const char * _name, DResponder * _dres ):
    DVariable( _name, _dres )
{
    SWString data_type( "DPoin" );

    // if this is the first DVariable, then create data now
    if ( dentity->is_data_set() == NO )
    {
        dentity->set_data( new DPointData, data_type );
    }
    // otherwise just make sure the data types match
    else
    {
        if ( data_type != dentity->get_data_type() )
        {
            cout << "ERROR: DPoint::DPoint() - cannot create DVariable <" <<
                get_name() << ">
            cout << " with type <" << data_type << "> because it
            cout << " already exists with type <"
                dentity->get_data_type() << ">
            cout.flush();
        }
    }
}

Figure 66 - DPoint Class Constructor Definition

The constructor for a Dynamic Data Type class has two primary responsibilities:

- if no previous Dynamic Variables have been created for the Dynamic Variable group,
  then the constructor of the Dynamic Data Type should allocate the data the Dynamic
  Entity is to maintain.
- if the data maintained by the Dynamic Entity has already been allocated, then the
  constructor of the Dynamic Data Type is responsible for making sure the data type of
  the new Dynamic Variable being created matches the existing type.

The concept and placement of data storage in the Dynamic Integration System can be
confusing. Dynamic Variables do not themselves maintain the data. Data is instead
maintained in a Dynamic Entity. This is so that multiple Dynamic Variables sharing the
same Dynamic Identifier may access the same data. Dynamic Entities are unaware of
what type of data they maintain. For this reason the DEntity class cannot be responsible for allocating data. The first Dynamic Variable created for a Dynamic Variable group is therefore responsible for allocating the data. It would like-wise make sense for the last Dynamic Variable destroyed from a Dynamic Variable group to be responsible for deleting the data. This responsibility is, however, delegated to the DEntity class as the DEntity simply deletes the data in its destructor. This greatly simplifies the destructors for Dynamic Variables as the responsibility of deleting the data has been removed.

A final note about data allocation is that the DVariable class is also unaware of the data type. The DVariable class exists so that the DEntity may interact with a Dynamic Data Type in a type-independent manner. This simplifies the implementation of functions for setting dependencies, accessing Dynamic Responders, etc. The DVariable class is designed such that the class that inherits from it will be responsible for data type-specific function such as allocation and usage.

The implementation of the remaining DPoint functions is shown in Fig. 67. The “get” and “set” functions are only shown for the x coordinate. The functions for the y and z coordinates are identical with the substitution of y or z for x.
Figure 67 - DPoint Class Definition

The get_data() and get_data_without_validating() functions are implemented to simply call the corresponding function in the DEntity class to return a type-cast pointer. The get_data() function is used whenever an attempt to read a value for the Dynamic Variable is used. This function first validates the data if necessary before returning it. If the motive is to set a new value for the data, then there is no point in validating it before a new value is set. In this case, the get_data_without_validating() function can be used.
The `get_x()` and `set_x()` functions retrieve the data from the DEntity either using the `get_data()` or `get_data_without_validating()` functions. The `get_x()` function simply returns the value it retrieves. The `set_x()` function retrieves the data, gives it a new value and calls the `value_is_modified()` function. This function tells the DEntity that the data it maintains has been modified. Appropriate actions such as invalidating forward references can then occur.

The way that the DPoint class will be demonstrated is with a simple class that constrains a point to be the midpoint of two other points using lazy-evaluation. The class declaration and definition are shown in Fig. 68.
class Constraint : public DResponder
{
private:
    DPoint end1, end2, midpoint;
    Flag request_validation( const DVariable* );

public:
    // CONSTRUCTOR
    Constraint( const char* name1, const char* name2, const char* name3 ) :
        end1( name1 ),
        end2( name2 ),
        midpoint( name3, this )
    {
        midpoint.set_dependency( end1 );
        midpoint.set_dependency( end2 );
    }

    //--------------------------------------------------------------
    Flag Constraint::request_validation( const DVariable* )
    {
        midpoint.set_x( (end1.get_x() + end2.get_x())/2 );
        midpoint.set_y( (end1.get_y() + end2.get_y())/2 );
        midpoint.set_z( (end1.get_z() + end2.get_z())/2 );

        return YES;
    }

Figure 68 - Sample Class Using DPoint

The Constraint class is implemented just as is any other Dynamic Responder by inheriting from the DResponder class and by maintaining several Dynamic Variables, some of which are connected to it. The class creates three Dynamic Variables in the form of DPoints and makes the two end points dependents of the midpoint. It should be noted that the only limitation on dependencies is that both variables are descendants of the DVariable class. This means that a DPoint can be dependent on a Dint, Dfloat or any other Dynamic Data Type. The request_validation() function is implemented to
calculate the coordinates of the midpoint such that they are half-way between the two end points.

Sample code using the `Constraint` class and several `DPoints` is shown in Fig. 69. The program creates an instance of the `Constraint` class and three `DPoints` corresponding to the ones created by the `Constraint` object. The end point `DPoints` are then given values. The values of all three `DPoints` are then printed.

```cpp
void main()
{
    // create the object which maintains the mid-point
    Constraint constraint( "end1", "end2", "midpoint" );

    // create DPoints for use by this module
    DPoint pt1( "end1" );
    DPoint pt2( "end2" );
    DPoint pt3( "midpoint" );

    // set initial values for points
    pt1.set_x( 0 );
    pt2.set_x( 0 );
    pt1.set_y( 2 );
    pt2.set_y( 8 );

    // print out points, mid-point will be constrained by Constraint object
    cout << "end point 1 = " << pt1 << "\n";
    cout << "end point 2 = " << pt2 << "\n";
    cout << "mid point = " << pt3 << "\n";
}
```

**Figure 69 - DPoint Sample Program**

The output from the sample program is shown in Fig. 70. As expected, the value of the midpoint is constrained to be halfway between the two end points.
end point 1 = ( 0, 2, 0 )
end point 2 = ( 0, 8, 0 )
mid point = ( 0, 5, 0 )

Figure 70 - DPoint Sample Program Output

Special Note:

An existing data type already defined by an application can also be made dynamic. To do this, define a new type that inherits from the DVariable class. Let the DEntity store an instance of the existing data type as its data. Define member functions in the dynamic class that are wrappers around each of the functions in the existing class. A wrapper function retrieves the instance of the existing class from the DEntity and performs the desired action on it. Functions that modify the data in any manner, call the value_is_modified() function.
Chapter 6  
A Design Methodology

This chapter presents a methodology for using the Dynamic Integration System. It is not a goal to create a methodology from scratch, as the field is already overcrowded with methodologies, but rather to supplement and extend an existing one. The chapter is broken down into two sections. The first discusses the overall structure with which an application should be designed. The second discusses how to achieve and document such a design.

6.1 Application Structure

Dynamic Variables provide a controlled information exchange method for modules. A dependency hierarchy is formed by constraining Dynamic Variables by other Dynamic Variables. To this point, the dependency hierarchy has been discussed as if it were a separate entity from which each module could interact. In fact, the dependency hierarchy does not really exist; it is a conceptual entity constructed by combining relationships developed between and within modules.

Each module models a few relationships using Dynamic Variables. Together, the relationships modeled by all modules form an implicit dependency hierarchy. This hierarchy is formed by connecting parameters modeled by each relationship. The connections are automatically and dynamically managed by the Dynamic Integration System.
This concept is depicted in Fig. 71. In this figure three relationships exist, possibly in separate modules. Relationships are signified by nodes connected by lines. Each node is a Dynamic Variable. By connecting the outputs of some relations to the inputs of others, an implicit dependency hierarchy is formed. This hierarchy can be thought of as a single unit, but is really just a set of several smaller, linked relationships. The mechanism for connecting these smaller relationships is to use common Dynamic Identifiers for Dynamic Variables. Two nodes with the same Dynamic Identifier are essentially the same node.

![Diagram showing individual relations and a dependency hierarchy](image)

**Figure 71 - Dependency Hierarchy Formation**

The purpose of this section is to describe how to design a module using Dynamic Integration techniques. The ability to construct a dependency hierarchy from several independent smaller parts provides a convenient mechanism for separating modules. In terms of an analysis module, the design technique is to construct a set of objects, each maintaining some relationship between parameters.
The preferred structure for a module created using Dynamic Integration techniques is shown in Fig. 72. The interface to the module consists of three parts:

- **the traditional procedural interface.** Dynamic Integration techniques simplify the procedural interface to a module, and sometime even remove it. However, there is still a need for procedural functions and it is more than appropriate to continue to use them.

- **the dependency hierarchy contributions.** For the numerous advantages that have been previously discussed, modeling an analysis using Dynamic Variables and a dependency hierarchy is very convenient. Also as previously discussed, the dependency hierarchy is not a separate entity that must be maintained. Instead each module contributes parts of the hierarchy in the form of equation objects. Not all equations and relationships need to be modeled using Dynamic Variables, only the ones that affect or are needed by other modules.

- **the dynamic variable interaction.** While relationship objects are constructed using Dynamic Variables, there are other forms of interaction with Dynamic Variables that warrant their own attention. Modules that respond to changes in Dynamic Variables or provide means for setting the value of Dynamic Variables are usually not connected to an analysis. Instead, these modules form the user-interface, display some geometry, or maintain the structure of the program. These modules typically view the dependency hierarchy as a separate entity rather than as being composed of individual relations.
Figure 72 - Module Content

Together several modules will interact to form an application. The goal is to compose this application from as many reusable components as possible. By moving as much of the interface from the procedural sections to the Dynamic Integration sections, module anonymity is increased. The result of de-coupling modules is that each module is easy to replace, i.e. a reusable design has been achieved. The structure of a typical application is shown in Fig. 73.
Figure 73 - Application Structure

The modules in Fig. 73 communicate via their dynamic interfaces. Some modules communicate directly with one another by instantiating Dynamic Variables with common Dynamic Identifiers. Other modules create relationships that are used to construct the dependency hierarchy. Still other modules interact with the dependency hierarchy by modifying and retrieving values of parameters.

Using Dynamic Integration, modules can interact through the interfaces provided by each module. Information flow through Dynamic Variables occurs through one of the two following channels:

- **dependency hierarchy.** Some modules contribute to the dependency hierarchy. Others perturb the hierarchy and still others monitor values in the hierarchy. The dependency hierarchy is treated as a nonpersistent active database with built-in constraints
between elements. The key is that while one module “thinks” that it is interacting with the hierarchy, it is actually indirectly interacting with another module. Thus, unknowingly, the two modules are automatically integrated.

- **shared Dynamic Variables.** Some Dynamic Variables are not associated with the dependency hierarchy. These Dynamic Variables may or may not be associated with an analysis. An example of this type of variable is a time value that increments a time value at certain intervals. Other modules may read this value without knowing how or where it is set.

Assembled together, modules may communicate in an orderly and organized fashion using the channels provided by Dynamic Integration.

### 6.2 Methodology Documentation

There are many object-oriented methodologies in existence, the most notable of which are documented in chapter 4. The goal of this dissertation is to modify one of these methods to incorporate the concept of Dynamic Integration. The methodology to be extended is the *Responsibility-Driven Approach* promoted by Wirfs-Brock *et al.* [Wirf89b] The criteria that were used for selecting this methodology include

- **guidelines provided for selecting objects.** Some methodologies assume that objects are easily found. Objects are easy to find, but finding the optimum set of objects is very difficult. Wirfs-Brock provides excellent guidelines for choosing objects.

- **reuse encouraged** While object-oriented design is supposed to natively support reuse, a poor design is no better than a non-object-oriented code. As Wirfs-Brock and Wilkerson note [Wirf89a], “object-oriented programming languages, through their support of definable and reusable classes, offer the potential of greatly increased programmer productivity. However, this potential can be realized only if users of these languages design for reusability.” As noted by Nierstrasz *et al.* [Nier92], “reuse
occurs by design, not by accident.” Reuse is central to the design of engineering-based software. When compared to other methodologies, as noted by Sharble and Cohen [Shar93], “the responsibility-driven approach seems to be significantly more effective for the production of software that can be maintained, extended, and reused.” The direct attention paid to reuse by Wirfs-Brock et al. is one of the advantages of Responsibility-Driven Design.

- **methods provided for disseminating borders among code.** Engineering codes are usually developed from components created for different purposes. It is only at some time after each component was created that it becomes a goal to integrate them. Once integrated, a natural distinction still exists between the two codes. This separation can be naturally modeled using the subsystem declaration in Responsibility-Driven Design.

This dissertation adds to the documentation of classes and subsystems (in collaborations graphs) found in Responsibility-Driven Design. Classes and subsystems are both addressed with the generic name of module in this section. The current documentation for a module consists of a block with sockets for the contracts supported by the module. An example is shown in Fig. 74. The module shown supports two contracts through which other objects may interact.

![Module Name with Connectors](image)

**Figure 74 - Current Responsibility-Driven Design Module Specification**

The problems associated with contracts is that both classes participating in the contract (the client and the server) must have full, prior knowledge of the contract. Dynamic
Integration overcomes this limitation by allowing multiple modules to interact without directly communicating with one another. These new forms of communication require new documentation techniques.

The new form of documentation introduced is called a Dynamic Module Specification, or a DMS for short. A Dynamic Module Specification is an extension of the module specification found in collaborations graphs. An example of a DMS is shown in Fig. 75.

![Dynamic Module Specification Diagram]

**Figure 75 - Dynamic Module Specification Diagram**

In Fig. 75 several symbols are introduced. These symbols are described below:

- **arrow pointing from a Dynamic Identifier to module.** This relationship indicates that the module uses the Dynamic Variable in some manner. The Dynamic Variable may be accessed by the module at any time.
- **arrow pointing from a module to a Dynamic Identifier.** This relationship indicates that the module can validate the Dynamic Variable. When the Dynamic Variable is invalid and an attempt to access occurs, the module will be asked to validate the Dynamic Variable.
• **arrow in module pointing from a Dynamic Identifier to a Dynamic Identifier.**
  This relationship is initiated by the module and marks a dependency between Dynamic Variables. Future modification to the pointing Dynamic Variable causes the pointed to Dynamic Variable to be marked as invalid.

• **open circle pointing from a module to a Dynamic Identifier.** This relationship indicates that the module is listening to changes in the Dynamic Entity. Any change to the Dynamic Variable causes some action to occur in the module.

• **closed circle pointing from a Dynamic Identifier to a module.** This relationship indicates that the module makes modifications to the Dynamic Variable. These changes may occur at anytime.

An example Dynamic Module Specification is shown in Fig. 76. This specification describes a lazy-evaluation implementation of the Adder module described later in Chapter 8. The purpose of the Adder module is to add two values together to form a third. In this example, the values being added are **transient x** and **steady-state x** which together form **x**. The diagram signifies this by noting that the Adder module is responsible for validating **x**. Inside the module, the dependencies for **x** are noted. Lastly it is noted that the Adder module accesses values for **transient x** and **steady-state x**.

![Diagram showing the relationship between a Dynamic Module and Dynamic Variables](image)

**Figure 76 - Example DMS Diagram: Lazy-Evaluation Adder Module**

An alternate implementation and specification for the Adder module are shown in Fig. 77. This implementation is based on eager-evaluation techniques. Any changes to **transient x** or **steady-state x** cause **x** to be immediately re-evaluated. This is specified in
the diagram by drawing arrows from \textit{transient} \( x \) and \textit{steady-state} \( x \) to the Adder module signifying that the Adder module is to be made aware when these values change. The arrow pointing to \( x \) signifies that the Adder module actively changes the value for \( x \).

![Diagram of Adder Module with arrows indicating transient and steady-state inputs.]

\textbf{Figure 77 - Example DMS Diagram: Eager-Evaluation Adder Module}
Several tools have been created to assist in the development of Dynamic Integration System code. Some of these tools are designed to rapidly construct interfaces to analysis modules, some assist in debugging applications and some serve both purposes. The tools developed include:

- Dependency Hierarchy Viewer
- Monitor
- Optimizer
- Grapher
- Dynamic Registry Viewer
- Dynamic Registry Monitor
- Type-Ahead
- Examiner

Each of the tools developed interacts with either the dependency hierarchy or the Dynamic Integration System. There is never any direct interaction with an analysis code. This means that each of the modules is reusable without changing any code.

The tools are created using Motif 1.2, OpenGL 1.0 and C++ 3.0, and have currently been tested on IBM RS6000s running AIX 3.2.5.
7.1 Dependency Hierarchy Viewer

The Dependency Hierarchy Viewer graphically shows the dependency hierarchy formed by Dynamic Variables and Dynamic Responders. The module opens by displaying a single node. This node, as with all nodes, may be expanded up and down. Expansion directions are disabled if the node has no dependencies. Nodes may also be moved on-screen and positions may be stored in a data file for later restoration.

Each node is either displayed with a gray or black label. A black label implies a valid state while a gray label implies an invalid state. As a program executes and Dynamic Variables alternate between valid and invalid states, these labels also dynamically change color. Invalid Dynamic Variables may be validated from this module. The Examiner module, as described later, may be launched from any node in the hierarchy tree.

A sample hierarchy is shown in Fig. 78. The figure shows an expanded hierarchy for the mass-spring-damper analysis described in Appendix A. If the hierarchy grows larger than the window, then scroll bars will appear on the bottom and right of the window. The scroll bars allow the user to move about the hierarchy without viewing it all at once.
Figure 78 - Dependency Hierarchy Viewer Module

7.2 Monitor

The Monitor module is used to access and set values for Dynamic Variables. The module consists of a set of text fields with numbers displayed to the right. The user may type any
name in the text field. If the name corresponds to a Dynamic Variable in the application, the number at the right will be set to the value of the corresponding Dynamic Variable. Using this technique, users may monitor any variable in a program that is represented with a Dynamic Variable.

If the Dynamic Variable does not have any dependencies, i.e. it is not constrained by some numerical relationship with other Dynamic Variables, then the number to the right will be displayed in a text field. The user is then free to alter the value of the Dynamic Variable by typing in a new value. This provides a very flexible mechanism for entering data into an analysis module and allows for quick and easy experimentation.

If the Dynamic Variable does have dependencies, then the user cannot change the value of the Dynamic Variable as it is constrained by some relationship. This type of Dynamic Variable is indicated in the Monitor module by displaying an etched frame around the number rather than putting the number in a text field. The number will also be displayed in different colors depending on its validity. A valid number is shown in black while an invalid number is shown in gray. The user may validate an invalid Dynamic Variable by clicking on its number.

The Monitor module is shown in Fig. 79.
Figure 79 - Monitor Module

7.3 Optimizer

The optimizer module was developed in conjunction with Brett Malone, a research associate and Ph.D. candidate working on the ACSYNT project at Virginia Tech. As its title suggests, the purpose of the Optimizer is to optimize (maximize or minimize) some output of an analysis by systematically varying inputs. The Optimizer module is shown in Fig. 80.
Figure 80 - Optimizer Module

The optimizer interface can be broken down into four groups: input parameters, constraints, the merit parameter, and the optimization box.

Inputs are Dynamic Variables that the optimizer will manipulate in order to try to optimize the merit parameter. Inputs may be selected from a list of unconstrained Dynamic Variables. Each input may be given an upper and/or lower bound within which the optimizer will keep the value of the variable.
Constraints are Dynamic Variables that bind the optimizer. Constraints may be selected from a list of Dynamic Variables with dependencies, i.e. Dynamic Variables whose values are calculated by the analysis rather than specified by the user. As with inputs, each constraint may be given an upper and/or lower bound. The optimizer will monitor the value of each of the constraints as the optimization process proceeds to make sure they stay within the prescribed bounds.

The merit parameter is the parameter that is to be optimized. The parameter may be a dummy value that is formed by composing several other outputs into one value. The optimize box allows the user to either minimize or maximize the merit parameter and provides a button for initiating the optimization process.

7.4 Grapher

The Grapher module, shown in Fig. 81, creates an X-Y plot of any two Dynamic Variables. The Dynamic Identifiers for the Dynamic Variables are entered in the appropriate text fields. A range may be specified for the X coordinate along with a number of data points to sample. A range may also be specified for the Y coordinate, or the Graphing module can perform an autoscale function.
Figure 81 - Grapher Module

7.5 Dynamic Registry Viewer

The Dynamic Registry Viewer, as shown in Fig. 82, is a simple tool that displays the contents of the Dynamic Registry. The Dynamic Identifiers displayed are segregated according to whether the Dynamic Variables they correspond to are CHANGEABLE or CONSTRAINED. As Dynamic Entities are created and destroyed, the lists displayed
dynamically update. Double-clicking on any entry launches the Examiner module for the particular Dynamic Variable.

![Dynamic Registry Viewer Module](image)

**Figure 82 - Dynamic Registry Viewer Module**

### 7.6 Dynamic Registry Monitor

The Dynamic Registry Monitor is used to monitor the events in the Dynamic Integration System as they happen. For each event that occurs a message will be displayed indicating which Dynamic Variable the event is happening to and what the event is. A history of 500 message is maintained in a scrolled text window as shown in Fig. 83.
7.7 Type-Ahead

The Type-Ahead module provides a quick and easy way to set the value of any Dynamic Variable. Using syntax in the form of “set <Dynamic Identifier> to <value>,” the value of a Dynamic Variable may be specified. The Type-Ahead module is shown in Fig. 84.

7.8 Examiner

The Examiner module, shown in Fig. 85, displays parameters relating to a Dynamic Variable. Information displayed includes a Dynamic Identifier, the data type, the valid and constraint states, and information pertaining to dependencies. All information is
dynamically updated as a program executes and states of various Dynamic Variables change. Entries in the forward and backward references list may be double-clicked to launch another instance of the Examiner module.

Figure 85 - Examiner Module
Chapter 8

Example Applications

This chapter presents and discusses several applications created using Dynamic Integration techniques. These examples were created to demonstrate the applicability of the Dynamic Integration System to various types of problems, including portability, legacy compatibility, and plug-n-play capabilities.

8.1 Mass-Spring-Damper Analysis Program

The first example program models a simple mass-spring-damper system such as the one described in Appendix A. The program provides the user with an interactive window for viewing the system, several user-interface modules for interacting with the system, and an analysis code that performs calculations pertinent to the system. The sample program was created using Microsoft Windows 3.1™ and is shown in Fig. 86. Since this demo was implemented on a non-Unix environment, it cannot take advantage of the tools described in the previous chapter. At the end of this section, a Unix equivalent of the demo (that can use the support tools) is briefly described.
Figure 86 - Mass-Spring-Damper Environment

8.1.1 Modules

The example program is composed of the following modules:

- **Geometry.** This module displays the mass-spring-damper system. The window is redrawn each time the Timer module "ticks." This creates the effect of animation.
- **Timer.** This module maintains a time value. The module ticks at certain time intervals. A tick is signified by incrementing the time Dynamic Variable.
• **Analysis.** The analysis of a mass-spring-damper system can be broken down into two parts: the transient solution and the steady-state solution. Summing the results of each of these solutions yields the position of the mass. The analysis is therefore broken down into a Transient Analysis Module, a Steady-State Analysis Module and an Adder module.

• **Monitor.** This module displays the value of any Dynamic Variable in the system. The user types in the name of the variable and a value appears to the right. The value updates as a program is running.

• **Recorder.** This module displays the value of any Dynamic Variable in a strip chart manner. The user can enter the name of any Dynamic Variable. The module accesses the value of this Dynamic Variable at specified time intervals and graphically displays it.

• **Type-Ahead.** This module provides the ability to interactively enter values for Dynamic Variables using Dynamic Identifiers. The Type-Ahead Module permits statements such as “set mass to 2.0” to be entered.

Each module communicates with the others exclusively through Dynamic Variables. No other form of communication between the modules exists. The Dynamic Variables implemented by each module are shown in Table 1.
### Table 1 - Dynamic Variable Utilization

<table>
<thead>
<tr>
<th>Module</th>
<th>Dynamic Identifiers</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>time</td>
<td>when time is changed, the geometry is updated</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>this value locates the mass-spring-damper system</td>
</tr>
<tr>
<td>Timer</td>
<td>time</td>
<td>this value is incremented at specific time intervals</td>
</tr>
</tbody>
</table>
| Transient Analysis | time  
|                 | mass                                 | input values                                                             |
|                 | stiffness                             |                                                                          |
|                 | damping                               |                                                                          |
|                 | initial position                      |                                                                          |
|                 | initial velocity                      |                                                                          |
|                 | damping ratio                         | output values                                                            |
|                 | natural frequency                     |                                                                          |
|                 | damped natural frequency              |                                                                          |
|                 | transient x                           |                                                                          |
| Steady-State Analysis | time  
|                 | stiffness                             | input values                                                             |
|                 | force constant                        |                                                                          |
|                 | excitation frequency                  |                                                                          |
|                 | damping ratio                         |                                                                          |
|                 | natural frequency                     |                                                                          |
|                 | magnitude x                           | output values                                                            |
|                 | phi                                   |                                                                          |
|                 | steady-state x                        |                                                                          |
| Adder           | transient x                           | input values                                                             |
|                 | steady-state x                        |                                                                          |
|                 | x                                     | output value                                                             |
| Monitor         | user specified                        | this value is continuously displayed                                     |
| Recorder        | user specified                        | this value is recorded in a strip-chart fashion                           |
| Type-Ahead      | user specified                        | this value is set to a user-specified value                               |
8.1.2 Module Assembly

Using Dynamic Integration, modules are not manually assembled. Instead they are all put into the same environment and allowed to interact on their own. The modules interact by sharing Dynamic Variables, responding to changes in Dynamic Variables, and by providing dependencies (and the supporting relationships) between Dynamic Variables. To demonstrate how the modules interact, a simple diagram (Fig. 87) and outline are provided for what happens when the Timer module increments the *time* Dynamic Variable:

![Diagram of module interaction](image-url)

**Figure 87 - Flow of Control for Mass-Spring-Damper Program**
Every animation cycle, the same actions occur. Those actions are:

1) The Timer modules increments the *time* value.
2) The Dynamic Variables dependent on *time* in the analysis modules are marked invalid. This includes the steady-state *x*, transient *x*, and *x* Dynamic Variables.
3) Since the Geometry Module is monitoring to *time*, the Dynamic Integration System sends it the notify_of_change() message. Upon receiving this message, the Geometry Module will begin to redraw its display. This involves accessing the value of *x*.
4) Since *x* is invalid, the Dynamic Integration System will attempt to validate it before returning a value to the Geometry Module. The validation process will first send the request_validation() message to the Adder Module.
5) The Adder Module will validate *x* by summing steady-state *x* and transient *x*. Accessing steady-state *x* and transient *x* will start validation processes for those two variables if they are invalid. This will cause the request_validation() message to be sent to both the Steady-State and Transient Analysis Modules.

The significance of the integration method employed in the example is that each module is unaware of the others. Functions in one module are never directly called from another module. Modules indirectly interact by instantiating Dynamic Variables with the same Dynamic Identifiers. The interaction between these Dynamic Variables is hidden from each module and automatically managed by the Dynamic Integration System. The resulting benefit is that modules may be designed independent of the environment they are to interact with thus enhancing reusability.

The reusability concept is exemplified by the user-interface modules (Monitor, Recorder, and Type-Ahead Modules). Each module has the ability to either display or set the value of a Dynamic Variable. It does not matter where the Dynamic Variable is being used.
throughout the rest of the system. These modules are designed and implemented independently of the other modules. This results in a module that is completely reusable. The next project created using Dynamic Variables will be able to utilize the functionality of these modules without altering one line of code.

8.1.3 Dynamic Module Specification Diagrams

To further document the modules implemented in this example program, Dynamic Module Specifications are included. These specifications are also intended to further clarify the manner in which they should be used.

The specifications are shown in Figs. 88 through 93.

**Figure 88 - Timer Module Dynamic Module Specification**

**Figure 89 - Geometry Module Dynamic Module Specification**
Figure 90 - Adder Module Dynamic Module Specification

Figure 91 - Transient Analysis Module Dynamic Module Specification

Figure 92 - Steady-State Analysis Module Dynamic Module Specification
Figure 93 - User-Interface Modules Dynamic Module Specifications

8.1.4 Alternate Implementation

The mass-spring-damper application has also been implemented in the UNIX environment. The UNIX application, shown in Fig. 94, animates the system and shows a trace of the **steady-state \( x \), transient \( x \), and \( x \) Dynamic Variables. The significance of this application is that it uses the same analysis as the previous example. The difference now is that a new interface has been created..
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mass</td>
<td>6.000</td>
</tr>
<tr>
<td>stiffness</td>
<td>1.000</td>
</tr>
<tr>
<td>damping</td>
<td>0.100</td>
</tr>
<tr>
<td>initial position</td>
<td>1.000</td>
</tr>
<tr>
<td>initial velocity</td>
<td>0.000</td>
</tr>
<tr>
<td>force constant</td>
<td>0.750</td>
</tr>
<tr>
<td>excitation frequency</td>
<td>0.200</td>
</tr>
</tbody>
</table>

Figure 94 - UNIX Mass-Spring-Damper Application
8.2 Aircraft Analysis Example

This example application was designed to demonstrate the capability of the Dynamic Integration System to model and integrate several dissimilar real-world analyses. The analyses are borrowed from the aircraft design field and represent either modules that are currently part of ACSYNT or modules that could be used to build a future ACSYNT. The analyses along with the Optimizer, Graphing, and Monitor modules, represent enough code to perform a simple design evaluation for an aircraft. It should be acknowledged that these modules were created with great assistance from Brett Malone, the resident aero-specialist.

The analyses modeled include the following:

- wing geometry
- weights
- cycle propulsion
- Lewis table-lookup propulsion
- lift surface
- aerodynamic forces
- drag buildup
- dynamic pressure

The analyses cover a diverse range of situations that represent a good test for the Dynamic Integration System. Some analyses were written in FORTRAN, requiring the construction of Dynamic Integration wrappers. Other analyses were written from the ground up in C++ with Dynamic Integration in mind. One analysis, the weights analysis, involved iteration between Dynamic Variables in order to converge. Finally, the dual-propulsion codes were implemented to demonstrate plug-n-play capabilities.
8.2.1 Analysis Controller

A Dynamic Integration System analysis is implemented by creating an object that coordinates the Dynamic Variables in the analysis. This object may be created or destroyed on the fly, enabling the user to activate or deactivate the analysis.

The interface that controls the activation of analyses for this example is shown in Fig. 95. Check boxes are used to represent each analysis. The propulsion codes are a special case because the two codes model the same analysis using different techniques. A special window controls these analyses using mutually exclusive radio buttons. The propulsion codes are explored further in a later section.

![Analysis Controller](image)

Figure 95 - Analysis Controller

8.2.2 Wing Geometry Analysis

The wing geometry analysis provides a way to coordinate the six parameters that describe a typical wing. The current implementation computes three of the parameters with respect to the other three. A future, more advanced system, may let the user actively
select which three are configured as outputs. The DMS for the current implementation is shown in Fig. 96.

![Wing Geometry Analysis DMS](image)

**Figure 96 - Wing Geometry Analysis DMS**

### 8.2.3 Weights Analysis

The weights analysis introduces the first test for the Dynamic Integration System - iteration. Iteration occurs when it is attempted to calculate the total weight of the aircraft. The parameters of interest are shown below:

\[
\begin{align*}
W_{total} & \quad \text{Takeoff gross weight} \\
W_{wing} & \quad \text{Wing weight} \\
W_{fuel} & \quad \text{Fuel weight used in cruise segment} \\
W_{eng} & \quad \text{Engine weight} \\
W_{fixed} & \quad \text{Fixed weight} \\
W_{fclm} & \quad \text{Fuel weight used in climb segment} \\
W_{cargo} & \quad \text{Cargo weight} \\
K_S & \quad \text{Wing structure technology factor} \\
S_W & \quad \text{Wing area} \\
S_{CSW} & \quad \text{Wing mounted control surface area}
\end{align*}
\]
\[ AR \quad \text{Aspect Ratio} \]

\[ (t/c) \quad \text{Wing thickness ratio} \]

\[ N \quad \text{Ultimate load factor} \]

\[ \Lambda \quad \text{Wing quarter-chord sweep} \]

\[ \lambda \quad \text{Taper ratio} \]

The weight of the total aircraft is fairly easy to calculate - it is the sum of the components, fuel, and contents. The relationship is shown below:

\[ W_{\text{total}} = W_{\text{wing}} + W_{\text{fuel}} + W_{\text{eng}} + W_{\text{fixed}} + W_{\text{fclm}} + W_{\text{cargo}} \quad (7) \]

The relationship gets tricky when the weights of the individual components are formed. Consider the formula for calculating the wing weight shown below:

\[ W_{\text{wing}} = 0.0051K_{S}S_{W}^{0.649}S_{C_{SW}}^{0.1} \frac{AR^{0.5}}{(t/c)_{\text{root}}^{0.4}} \frac{(NW_{\text{total}})^{0.557}(1+\lambda)^{0.1}}{\cos \Lambda_{25\%}} \quad (8) \]

The problem with this equation is that it is based on the total weight of the aircraft, a value that we are in the process of trying to calculate. The work-around for this problem using the Dynamic Integration System is similar to the method used in procedural programming - introduce an estimated total weight term:

\[ W_{\text{total_{est}}} \quad \text{Estimated takeoff gross weight} \]

The estimated total weight term is then substituted into any equation that was previously basing its output on the total weight. The wing weight equation then becomes:
\[ W_{\text{wing}} = 0.0051 K_S S^0.649 C_{SW}^0.1 \frac{AR^{0.5}}{(t/c)^{0.4}} \left( \frac{NW_{\text{total est}}}{c} \right)^{0.557} (1 + \lambda)^{0.1} \cos \Lambda_{25\%} \] (3)

The weight equation now starts with an initial guess. The actual weight is then formulated based on this guess. If the actual value and the guess are reasonably close, then the system has converged. If they differ significantly, then a new guess is made and the procedure repeats. This iteration process is automated by an iteration object that is described in Section 5.5.7.

The DMS for the weights analysis is shown in Fig. 97.

Figure 97 - Weights Analysis DMS
8.2.4 Propulsion Analyses

The propulsion analyses present the next set of tests for the Dynamic Integration System - plug-n-play and legacy FORTRAN compatibility requirements. The propulsion analysis can be performed in two manners: analytically and empirically. The analytical method calculates values on the fly while the empirical method interpolates values from a table. Both of these analyses are existing FORTRAN codes extracted from ACSYNT. The controller for the propulsion codes is shown in Fig. 98. Method selection is performed interactively using a series of radio buttons. Choosing a method deletes the previous analysis object (if any) and allocates a new one. This action essentially unplugs the existing analysis and plugs in the new analysis.

![PropWin Analysis](image)

**Figure 98 - Propulsion Analysis Controller**
The requirement for using legacy FORTRAN code is accomplished by wrapping the FORTRAN global space with Dynamic Variables. The concept is depicted in Fig. 99.

**Figure 99 - DIS - FORTRAN Integration**

Each parameter that was modeled using a global variable in the FORTRAN code is now additionally modeled with a Dynamic Variable. The resulting Dynamic Variables are segregated into inputs and outputs based on the functions performed by the FORTRAN analysis. Dependencies are then specified between the Dynamic Variables. Subsequent changes to the input Dynamic Variables result in the Dynamic Interface Wrapper writing the value of the Dynamic Variable to the FORTRAN global space. The Dynamic Integration System also automatically invalidates all the output Dynamic Variables. Invalid output Dynamic Variables are validated by executing the FORTRAN analysis and copying the results to the output Dynamic Variables.

Creating a Dynamic Interface provides the same access to the FORTRAN code that a native Dynamic Integration System analysis enjoys. The module can be plugged into any application that is built using Dynamic Integration System techniques. Likewise, any
application using Dynamic Integration System techniques can gain access to the functionality provided by the FORTRAN code.

The DMS for the cycle propulsion analysis is shown in Fig. 100 and the DMS for the Lewis table lookup analysis is shown in Fig. 101.

Figure 100 - Cycle Propulsion Analysis DMS
8.2.5 Lift Surface Analysis

The lift surface analysis incorporates a vortex lattice method encoded by Brett Malone. The analysis is coded in C++ and directly uses Dynamic Variables. The DMS for the analysis is shown in Fig. 102.
8.2.6 Aerodynamic Forces Analysis

The aerodynamic forces analysis introduces Dynamic Variables that link together some of the other analyses. For instance, lift coefficient is an output of the lift surface analysis while AC drag is an input to the propulsion codes. The DMS for the aerodynamics forces analysis is shown in Fig. 103.

![Aerodynamic Forces Analysis DMS](image)

Figure 103 - Aerodynamic Forces Analysis DMS

8.2.7 Drag Buildup Analysis

The drag buildup analysis, like the aerodynamic forces analysis, fills in some connections between various Dynamic Variables. The DMS is shown in Fig. 104.

![Drag Buildup Analysis DMS](image)

Figure 104 - Drag Buildup Analysis DMS

---

Chapter 8 - Example Applications
8.2.8 Dynamic Pressure Analysis

The dynamic pressure analysis rounds out the list of analyses currently implemented. The DMS for the dynamic pressure analysis is shown in Fig. 105.

![Dynamic Pressure Analysis Diagram]

*Figure 105 - Dynamic Pressure Analysis DMS*
Chapter 9  

Conclusions

This dissertation has provided facilities for creating engineering software, namely the Dynamic Integration System. Previously, engineers created software on an ad-hoc basis. This lack of focus in the design led to software that was difficult to maintain. Integration of codes had to be performed on a case-by-case basis. Following the guidelines developed here, engineering software can be developed that is well suited for reuse and integration without affecting the local application domain.

The new design process is based on the concept of Dynamic Variables. A Dynamic Variable is a variable that is identified by some text identifier known as a Dynamic Identifier. Multiple modules may instantiate Dynamic Variables with common Dynamic Identifiers. The Dynamic Variables are then implicitly linked.

Events are associated with Dynamic Variables so that certain functions may be called whenever a Dynamic Variable changes state. Example events are change notifications, invalidation notifications, etc. In addition, dependencies may be setup between Dynamic Variables. One Dynamic Variable may be specified as a dependency to another. When the dependency is modified, the first Dynamic Variable is marked invalid. Subsequent requests for the Dynamic Variable result in a validation process for the Dynamic Variable which involves calling some specified function that knows how to calculate a new value for the Dynamic Variable.

Analyses are built from Dynamic Variables by constructing relationship objects. These objects each manage the dependencies for one particular Dynamic Variable. An analysis
creates many of these objects, possibly thousands. The objects are linked to one another by properly choosing Dynamic Identifiers for the Dynamic Variables used in the objects. The output of one object is linked to the input of another. In this manner, a set of relationship objects forms a dependency hierarchy.

The significance of the dependency hierarchy is that it is not a separate entity. Instead, it is a conceptual entity formed by linking relationship objects. The linking process is performed solely by matching Dynamic Identifiers in the objects. This removes the need for direct communication between relationship objects. As a direct result, the dependency hierarchy may be constructed from equation objects existing in separate modules. This is a very flexible integration mechanism.

The benefits of modeling an analysis or application using Dynamic Variables include:

- **increased flexibility of interaction.** The level of interaction with an existing analysis module is limited to what the analysis programmer provides. This is not due to a lack of thoughtfulness on the analysis programmers part, but rather to problems inherent in procedural interfaces. Dynamic Variables provide a much richer level of interaction with an analysis by providing equally protected access to parameters without imposing an extra burden on the analysis programmer.

- **improved management of data.** Integrating multiple analysis modules involves coordinating data flow between the modules. Consistency must be maintained with respect to the values of parameters modeled by each of the modules. The Dynamic Integration System implicitly maintains consistency between Dynamic Variables, greatly simplifying integration procedures.

- **improved efficiency of analysis execution.** There are two reasons that efficiency of analysis modules increases: 1) improved interaction mechanisms and 2) the introduction of lazy-evaluation techniques. Using Dynamic Variables, it is possible
to pick and choose the parameters that are to be calculated by the analysis. This is opposed to systems that must execute the entire analysis to retrieve a single value. Lazy-evaluation techniques improve efficiency by only calculating parameters that need to be calculated.

- **ability to visualize relationships.** The support tools provided in this dissertation provide unprecedented levels of interaction with analysis modules. Parameters can be interactively examined. Values can easily be accessed and specified. The direct dependencies between parameters can also be visualized using the Dependency Hierarchy module.

- **ability to create generic, reusable modules.** By de-coupling modules, the Dynamic Integration System permits the development of modules independently of the environments they are to be used in. Examples of such modules are the support tools developed in this dissertation.

To use the Dynamic Integration System, a design methodology has been created. This methodology is an extension of the Responsibility-Driven Design technique developed by Wirfs-Brock et al.. As a design notation for the methodology, the Dynamic Module Specification diagram has been created. In addition to the methodology, a general description for the structure of an engineering application has also been specified.

By following the methodology specified in this dissertation, the problems associated with developing and integrating engineering software can be greatly reduced. Focus for a project can be provided and strategies can be imposed that coordinate the efforts of multiple developers. This adds order to what is now generally a chaotic process.
Cited References


References 216


References


References 230


Uncited References


**abstract class** - An abstract class is a class that is designed for the sole purpose of being inherited from by other classes. An abstract class declares one or more virtual functions that derived classes define. Abstract classes provide a means for extensibility in a design.

**changeable** - With respect to a Dynamic Entity or Dynamic Variable, refers to a state where the value of the Dynamic Variable is *changeable* by the user. The value is not constrained by any dependencies.

**class** - As applied to object-oriented design, a class is an encapsulation of data and associated functions. A class is usually an abstraction of some object in the real world.

**client-server** - In terms of object-oriented design, two classes can form a client-server pair. The client makes requests from the server. In terms of Responsibility-Driven design, a contract binds a client-server pair.

**collaboration** - In terms of Responsibility-Driven design, a collaboration represents requests from a client to a server in fulfillment of a client responsibility.

**collaborations graph** - In terms of Responsibility-Driven design, a graph showing collaborations between classes and subsystems. An example is shown in the figure below:

![Collaborations Graph](image)

**concrete class** - A concrete class is a class that may be instantiated by an application as opposed to an abstract class.
**consistent** - With respect to this dissertation, a set of variables is said to be consistent if the their values are in agreement with the relationships that bind them. Changing the value of an input to an analysis makes the set of parameters consisting of the inputs and outputs of the analysis invalid.

**constrained** - With respect to a Dynamic Entity or Dynamic Variable, refers to a state where the value of the Dynamic Variable is *constrained* by the existence of dependencies. Some relationship exists whereby the value of the Dynamic Variable may be and is calculated from its dependencies. The user cannot change the value of a *constrained* Dynamic Variable as the value is governed by its dependencies.

**contract** - In terms of Responsibility-Driven design, a cohesive set of responsibilities in a server class that a client class can count on.

**coupling** - The amount of and strength of connections between modules. Two highly coupled modules are highly dependent on one-another. Highly coupled modules are difficult to reuse as they can perform their function without the assistance of other modules. Encapsulation within object-oriented design tends to decrease coupling and increase autonomy.

**declaration** - (a.k.a. class declaration) The part of a class usually stored in a header (.h) file. The class declaration does not specify how any part of the class is implemented, only what the class is composed of. A declaration is usually accompanied by a definition that specifies implementation details.

**definition** - (a.k.a. class definition) The part of a class usually stored in a source (.C or .cpp) file. The class definition specifies how the class is implemented. A class definition must be accompanied by a class definition that defines what the class is composed of.

**DEntity** - See Dynamic Entity.

**Dependency Hierarchy Viewer Module** - A support tool developed for the Dynamic Integration System. This module graphically displays the dependency hierarchy between Dynamic Entities. Nodes in the tree may be interactively expanded or collapsed. Invalid Dynamic Entities are shown in gray while valid ones are shown in white.

**DEResponder** - See Dynamic Entity Responder.

**DIS** - See Dynamic Integration System.

**DMS** - See Dynamic Module Specification.
**DRegistry** - See Dynamic Registry.

**DResponder** - See Dynamic Responder.

**DResponder** - See Dynamic Registry Responder.

**DVariable** - See Dynamic Variable.

**Dynamic Data Types** - Dynamic Data Types are types inherited from the DVariable class. Current implementations include Dfloat, Dint, and Ddouble. A Dynamic Data Type is a DVariable with a specific data type.

**Dynamic Entity** - (a.k.a. DEntity) An object stored in the Dynamic Registry that is used to store some system information. Dynamic Entities may have backward and forward references and may be either valid or invalid. Applications do not usually directly deal with Dynamic Entities, instead they instantiate Dynamic Variables that provide a more usable interface to the Dynamic Entity. It is possible to have several Dynamic Variables simultaneously linked to a single Dynamic Entity thus allowing several modules to share access to the Dynamic Entity. Dynamic Entity Responders may connect to a Dynamic Entity to receive messages when actions such as state changes occur within the Dynamic Entity.

**Dynamic Entity Responder** - (a.k.a. DEResponder) An object, which when connected to a Dynamic Entity, receives messages when actions occur in the Dynamic Entity. Example actions include the validation of the Dynamic Entity, the addition of a dependency, or the removal of a forward reference.

**Dynamic Identifier** - Every Dynamic Variable is associated with a Dynamic Identifier. The Dynamic Identifier identifies which parameter the Dynamic Variable represents. Multiple modules may implicitly link together by instantiating Dynamic Variables with common Dynamic Variables.

**Dynamic Integration System** - (a.k.a. DIS) The framework mainly consisting of the Dynamic Responder, Dynamic Variable, Dynamic Entity, and Dynamic Registry classes. The purpose of the framework is to provide tools that make creating engineering software easier and which produce more reusable and flexible software.

**Dynamic Module Specification** - (a.k.a. DMS) A design document that describes modules using Dynamic Variables.

**Dynamic Registry** - (a.k.a. DRegistry) The database that holds all of the Dynamic Entities. The Dynamic Responder also maintains a list of Dynamic Registry Responders that are notified of events that occur in the Dynamic Responder.
**Dynamic Registry Monitor Module** - A support tool developed for the Dynamic Integration System. This module displays events as they occur in the Dynamic Integration System. A history of the last 500 events is maintained by the module.

**Dynamic Registry Responder** - (a.k.a. DResponder) An object, which when connected to the Dynamic Registry, receives messages when events occur in the Dynamic Registry. Events include most things that happen in the Dynamic Integration System including such things as a Dynamic Entity being validated, a Dynamic Entity being created or a dependency being removed from a Dynamic Entity.

**Dynamic Registry Viewer Module** - A support tool developed for the Dynamic Integration System. This module displays all the Dynamic Entities in an application. The Entities are segregated into two lists based on their constraint state. Double-clicking on any entity launches the Examiner module.

**Dynamic Responder** - (a.k.a. DResponder) An object, which when connected to a Dynamic Variable, receives messages when actions occur to the Dynamic Variable. Such actions include changes in constraint state, changes in valid state, and changes in value. A Dynamic Responder may also be connected to a Dynamic Variable for purposes of calculating a new value for the Dynamic Variable when the Dynamic Variable needs to be validated.

**Dynamic Variable** - (a.k.a. DVariable) An object that is linked to a Dynamic Entity that provides a more user-friendly interface to the Dynamic Entity. Modules interact with Dynamic Entities by instantiating Dynamic Variables corresponding to the Dynamic Entity. Multiple modules may indirectly communicate by instantiating Dynamic Variables that connect to the same Dynamic Entity. Each Dynamic Variable may only connect to a single Dynamic Entity. A Dynamic Variable may also be connected to a single Dynamic Responder.

**Dynamic Variable group** - The set of all Dynamic Variables sharing a common Dynamic Identifier is referred to as a Dynamic Variable group.

**event** - An action that occurs in response to some trigger. Events in the Dynamic Integration System are virtually implemented in Responder classes. Once a trigger occurs, the Responder executes some function that is implemented in a derived class. An example is when a Dynamic Variable is changed. In response to the change trigger, the Dynamic Responder class executes the value_has_changed() method.
Examiner Module - A support tool developed for the Dynamic Integration System. This module shows information about a Dynamic Entity including relevant states, data type, and dependencies.

framework - As applied to object-oriented design, a framework is a set of classes that together serve some purpose. A framework is a distribution mechanism for object-oriented code similar to a function library is to procedural code.

global data - Variables that are declared global may be accessed from any location in a program. This may be used as a primitive integration mechanism as one module may write to global space while another reads. This mechanism quickly grows unwieldy when multiple modules read and write to a common global space - what is being modified? where and when is it occurring?

Graphing Module - A support tool developed for the Dynamic Integration System. The graphing module plots the value of one Dynamic Variable with respect to another.

in-line functions - As applied to C++, an in-line function is a function that is used when the normal function call overhead is prohibitive. In-line functions are compiled and then directly substituted into a code thus removing the call to the function. In-line functions are different from macros in that macros are not compiled code and do not support the associated abilities. Macros are simply text substitution in source code.

inheritance - As applied to object-oriented design, inheritance is a mechanism for reuse. If a class inherits from another, it assumes the properties of that class in addition to the ones it defines for itself.

inheritance hierarchy - A hierarchy showing the inheritance path from superclass to subclass. An example is shown in the figure below:

![Inheritance Diagram]

 instantiate - The process of taking an instance of a class. Objects are instantiated from classes.

invalid - With respect to a Dynamic Entity or Dynamic Variable, refers to a state where the value of the Dynamic Variable is invalid with respect to its dependencies.
maintenance - When applied to software, maintenance includes any of the following: bug-fixes, code enhancement, module replacement, feature addition, etc. Basically anything modification to a code once that code is released is referred to as maintenance.

Monitor Module - A support tool developed for the Dynamic Integration System. This module allows the user to interact with Dynamic Variables in an application. Dynamic Variables are specified by typing a Dynamic Identifier. If the Dynamic Variable is changeable, then the user will also be permitted to enter numerical values. Numbers corresponding to invalid Dynamic Variables are shown in gray while valid ones are white. Clicking on invalid numbers causes the Dynamic Integration System to validate it.

object - As applied to object-oriented design, an object is a specific instantiation of a class. The words "class" and "object" are sometimes interchanged however. In general a class represents an abstraction while an object represents an instance of the abstraction.

Optimizer Module - A support tool developed for the Dynamic Integration System. The module permits the user to specify various Dynamic Identifiers as input parameters and constraints. A merit parameter is also specified. The module either minimizes or maximizes the merit Dynamic Variable by systematically modifying the input Dynamic Variables while not violating the specified constraints.

overloaded operator - In C++, the ability for the programmer to define operators normally only associated with the programming language is referred to as operator overloading. Common operators such as +, and = have been overloaded for Dynamic Variables in the Dynamic Integration System to perform special functions in addition to their normally associated functions.

polymorphism - The ability of multiple dissimilar objects to conform to the same interface. Implemented as virtual functions in C++. Polymorphism permits an object-oriented application to be more flexible in terms of the objects it can interact with.

protocol - In terms of Responsibility-Driven design, a specific signature applied to a responsibility. A responsibility may have one or more protocols, but a protocol is only associated with one responsibility..
**responder** - An object that implements functions representing events. One of these functions is executed when triggered by some action. Responder classes in the Dynamic Integration System include the Dynamic Responder, Dynamic Entity Responder, and Dynamic Registry Responder classes. These classes respectively respond to events in the Dynamic Variable, Dynamic Entity, and Dynamic Registry classes.

**responsibility** - In terms of Responsibility-Driven design, a responsibility is some action or thing a class is responsible for.

**Responsibility-Driven design** - The design methodology developed by Wirfs-Brock et al. [Wirf90] Responsibility-Driven design is governed by defining responsibilities for classes.

**structure** - (a.k.a. data structure) A structure is a grouping of data into a new data type. Consider the following declaration for a structure:

```c
struct Point
{
  float x;
  float y;
  float z;
};
```

This declaration creates a new data type called Point that consists of three data members corresponding to the coordinates of the point. Data members may be accessed using the "." operator, e.g. to access the "x" coordinate of a point, use `point.x`. Structures do not exist in FORTRAN, but are present in C and C++.

**subsystem** - In terms of Responsibility-Driven design, a cohesive set of classes the work together to fulfill some role. Collaborations between classes in a subsystem are maximized while outside collaborations are minimized.

**trigger** - A trigger takes place when some action occurs. The action triggers some other event to occur. Triggers in the Dynamic Integration System occur when things happen in the Dynamic Registry, Dynamic Entity, or Dynamic Variable classes. An example trigger is a change in value for a Dynamic Variable. This action triggers a Dynamic Responder object to execute the `value_has_changed()` event.

**Type-Ahead Module** - A support tool developed for the Dynamic Integration System. This module lets users type in command like “set wing span to 210.” This particular command will set the Dynamic Variable `wing span` to the value 210.
**valid** - With respect to a Dynamic Entity or Dynamic Variable, refers to a state where the value of the Dynamic Variable is *valid* with respect to its dependencies.

**Validator** - The Dynamic Variable maintained by a Dynamic Entity that knows how to validate the Dynamic Entity. This Dynamic Variable is connected to a Dynamic Responder that has implemented the `request_validation()` function. Dependencies can only be set or removed through the validator.

**virtual function** - A virtual function is C++'s implementation of polymorphism. A base class may define a function as virtual. Derived classes may then choose to implement the function. When the function is called from the base class, it is redirected to the derived class if the function has been polymorphically redefined.
Appendix A  Mass-Spring-Damper Analysis

An example analysis used throughout this dissertation is based on a mass-spring-damper system such as the one shown in Fig. 106.

Figure 106 - Mass-Spring-Damper System

A.1 Parameters

This appendix presents the equations involved in modeling the system. The input parameters for the system are:

- $k$ - the stiffness coefficient of the spring
- $c$ - the damping coefficient of the damper
- $m$ - the mass of the mass
- $x_0$ - the initial position of the mass
- $x_0$ - the initial velocity of the mass
- $t$ - the elapsed time since the mass was set in motion
- $F_0$ - the excitation force constant
- $\omega$ - the frequency of the excitation force
The output parameters are:

- \( x_{\text{total}} \) - the position at time \( t \)
- \( x_{\text{transient}} \) - the transient position at time \( t \)
- \( x_{\text{steady-state}} \) - the steady-state position at time \( t \)
- \( X \) - the magnitude of the steady-state oscillation
- \( \phi \) - the phase between the excitation force and the oscillation
- \( \omega_n \) - the natural frequency
- \( \omega_d \) - the damped natural frequency
- \( \zeta \) - the damping ratio

### A.2 Analysis

The primary output calculated for the system is the position of the mass, \( x_{\text{total}} \). This output is formed by summing the transient and steady-state solutions as shown below:

\[
x_{\text{total}} = x_{\text{transient}} + x_{\text{steady-state}}
\]

(A.9)

### A.2.1 Steady-State Solution

The steady-state solution can be found to be:

\[
x_{\text{steady-state}} = X \sin(\omega t - \phi)
\]

(A.10)

where,

\[
X = \frac{\frac{F_0}{k}}{\left[ \sqrt{1 - \left( \frac{\omega}{\omega_n} \right)^2} \right]^2 + \left[ 2\zeta \left( \frac{\omega}{\omega_n} \right)^2 \right]^2}
\]

(A.11)

and,

\[
\tan \phi = \frac{2\zeta \left( \frac{\omega}{\omega_n} \right)}{1 - \left( \frac{\omega}{\omega_n} \right)^2}
\]

(A.12)
\[ \zeta = \frac{c}{2m \omega_n} \]  
\[ \omega_n = \sqrt{\frac{k}{m}} \]  

**(A.2.2 Transient Solution)**

The transient solution varies depending on the damping ratio, \( \zeta \). If the damping ratio is less than one, the system is said to be *underdamped* and will oscillate. The oscillation will obey the following equation:

\[ x_{\text{transient}} = e^{-\zeta \omega_n t} \left( \frac{\dot{x}_0 + \zeta \omega_n x_0}{\omega_n \sqrt{1 - \zeta^2}} \sin \sqrt{1 - \zeta^2} \omega_n t + x_0 \cos \sqrt{1 - \zeta^2} \omega_n t \right) \]  

If the damping ratio is exactly equal to one, then the system is *critically damped* and may oscillate. The equation for \( x \) can be shown to be:

\[ x_{\text{transient}} = [x_0 + (\dot{x}_0 + \omega_n x_0)t] e^{\zeta \omega_n t} \]  

If the damping ratio is greater than one, then the system is *overdamped* and will not oscillate. The equation for \( x \) can be shown to be:

\[ x_{\text{transient}} = Ae^{(-\zeta + \sqrt{1 - \zeta^2}) \omega_n t} + Be^{(-\zeta - \sqrt{1 - \zeta^2}) \omega_n t} \]  

where,

\[ A = \frac{\dot{x}_0 + (\zeta + \sqrt{\zeta^2 - 1}) \omega_n x_0}{2 \omega_n \sqrt{\zeta^2 - 1}} \]  

and,

\[ B = \frac{-\dot{x}_0 - (\zeta - \sqrt{\zeta^2 - 1}) \omega_n x_0}{2 \omega_n \sqrt{\zeta^2 - 1}} \]
This section provides a brief introduction of the concepts of object-oriented design. Examples of concepts are presented as they apply to the C++ programming language.

**B.1 Abstraction**

Abstraction, as applied to computer programming, is the level at which a program may be described. A low level of abstraction deals with computer specific terms such as floats, arrays, etc. A high level of abstraction on the other hand deals with terms that are closer to the problem domain such as airplanes, springs, ships, etc.

The higher the level of abstraction the better. The closer a programming language can model the problem domain, the easier a program will be able to create and comprehend. The progression of programming languages has been from assembly to procedural to object-oriented. Assembly languages are very low level and require lots of work on the programmers' behalf to model a problem domain. Procedural languages raise the level of abstraction by introducing functions and hierarchical decomposition. Some languages, such as C and Pascal, even provide custom data types. Object-oriented languages represent the latest shift to higher levels of abstraction by the introductions of encapsulation, inheritance, and polymorphism.
This appendix will explain object-oriented features while describing a simple example. The example application is a program that models shapes. Each shape has the property that it may be located somewhere in space, have a weight and volume.

**B.2 Encapsulation**

Encapsulation is the grouping of functions and data into a single entity - an object. Encapsulation is a form of modularization in object-oriented programming.

**B.2.1 Objects**

A program written in an object-oriented language is composed of many objects.

**B.2.2 Classes**

Classes are templates from which objects are instantiated. A class defines an object and specifies how it may behave. In C++ the process of creating a class is broken down into two parts: the declaration and the definition. A declaration specifies what is in a class and what the class can do. The definition provides the implementation details of how the declaration is to be performed.

To start the example, a simple shape will first be created. This shape is a rectangular block. The declaration for the `Block` class is shown in Fig. 107, while the definition is shown in Fig. 108.
class Block
{
private:
float x, y, z;
float width, height, depth;
float density;

public:
    // CONSTRUCTOR
    Block( float w, float h, float d);

    // MEMBER FUNCTIONS
    float get_volume();
    float get_weight();
    void set_location( float x, float y, float z);
};

Figure 107 - Block Class Declaration

//--------------------------------------------------------------------------
Block::Block( float w, float h, float d )
{
    width = w;
    height = h;
    depth = d;
    density = 1.0;
}

//--------------------------------------------------------------------------
float Block::get_volume()
{
    return width*height*depth;
}

//--------------------------------------------------------------------------
float Block::get_weight()
{
    float volume = get_volume();
    return density*volume;
}

//--------------------------------------------------------------------------
void Block::set_location( float _x, float _y, float _z )
{
    x = _x;
    y = _y;
    z = _z;
}

Figure 108 - Block Class Definition
Classes are composed of two primary components:

- **Data Members.** Data members are the elements maintained by the class. In the block class created, this includes variables for density, location, and size. In C++, data members are divided into three sections: private, protected, and public. These access modifiers control who can access the data and from where. In general, private data members are internal to the class while public members are accessible from the outside, with protected falling somewhere in between.

- **Member Functions.** Member functions provide the interface to a class. These functions allow outside objects to manipulate data stored inside the class. These functions should, preferably, reflect what the class is supposed to do rather than what data it stores. These functions should present a friendlier, more manageable interface than would be provided by providing direct access to all data members of a class. Like data members, in C++ member functions are segregated by the private, protected, and public access members.

**B.3 Inheritance**

Inheritance is a mechanism for reuse in object-oriented design. One class may inherit the properties of another. The need for this feature becomes apparent when a new class is introduced to the solid modeling example. Consider the introduction of a cylinder class as shown in Fig. 109 and Fig. 110.
class Cylinder
{
private:
    float   x, y, z;
    float   radius, height;
    float   density;
public:
    // CONSTRUCTOR
    Cylinder( float r, float h );

    // MEMBER FUNCTIONS
    float    get_volume();
    float    get_weight();
    void     set_location( float x, float y, float z );
};

Figure 109 - Cylinder Class Declaration

 hexadecimal:Cylinder( float r, float h )
{
    radius = r;
    height = h;
    density = 2.0;
}

 float Cylinder::get_volume()
{
    float pi = 3.141593;
    return pi*radius*radius*height;
}

 float Cylinder::get_weight()
{
    float volume = get_volume();
    return density*volume;
}

 void Cylinder::set_location( float _x, float _y, float _z )
{
    x = _x;
    y = _y;
    z = _z;
}

Figure 110 - Cylinder Class Definition
The Cylinder class has many of the same features that are present in the Block class. The traditional approach to programming requires that each of the common features be re-implemented for the new class. This can be error-prone and tedious if the class is large. Maintenance is also complicated is a simple change needs to be made to one of the classes - the same change needs to be propagated to each of the other classes.

Fortunately the inheritance mechanism in object-oriented design removes many of the burdens described. Inheritance factors out common responsibilities of classes and stores them in a centralized location that may be utilized by all classes. The Block and Cylinder class both share the properties that they maintain a location and density. These properties can be factored out into a new class such as the Solid class shown in Figs. 111 and 112.

```cpp
class Solid
{
    protected:
        float x, y, z;
        float density;

    public:
        // CONSTRUCTOR
        Solid( float density );

        // MEMBER FUNCTIONS
        void set_location( float x, float y, float z );
};
```

Figure 111 - Solid Class Declaration
Solid::Solid( float r, float h )
{
    radius = r;
    height = h;
    density = 2.0;
}

void Solid::set_location( float _x, float _y, float _z )
{
    x = _x;
    y = _y;
    z = _z;
}

Figure 112 - Solid Class Definition

By factoring out redundant responsibilities from all "solid" classes, the creation of new classes is simplified. Effort can be concentrated on identifying and implementing features specific to a class. Consider the new implementation of the Cylinder class utilizing inheritance shown in Figs. 113 and 114. Redundant code involving density and location is removed. Only the factors specific to the Cylinder class, such as the dimensions the cylinder, need to be specified.

class Cylinder : public Solid
{
private:
    float radius, height;

public:
    // CONSTRUCTOR
    Cylinder( float r, float h );

    // MEMBER FUNCTIONS
    float get_volume();
    float get_weight();
};

Figure 113 - Cylinder Class Declaration Using Inheritance

Appendix B - Object-Oriented Design with C++ Examples
Figure 114 - Cylinder Class Definition Using Inheritance

### B.4 Polymorphism

Polymorphism is the ability of an object to implement several different functions through a common interface. The purpose of polymorphism is to permit an application to interact with a group of classes sharing common responsibilities in a standardized way. Using polymorphism, the application may be extended to incorporate new features by introducing new functions that fit the mold of the existing classes. The existing application code does not need to be modified because it can interact with the group of classes through the polymorphic interface.

The need for polymorphism can be shown in the solid modeling program. Consider the requirement that each object be able to calculate a weight and volume. Thus far each new class has had to implement the `get_weight()` function because this function required the volume of the particular object. Using polymorphism, some of these common responsibilities can be shifted up the class hierarchy to the Solid class.
Polymorphism is implemented in C++ in the form of *virtual functions*. A virtual function is a function that is declared in a base class, but defined in a derived class. An example of this in the solid modeling program is the *get_volume()* function. This function is common to all solids, but implemented differently in each class. By making the function virtual, it may be considered a member function of the *Solid* class even though it is not implemented there. This then permits the declaration and definition of the *get_weight()* function in the *Solid* class rather than in each class derived from the *Solid* class.

The modified version of the *Solid* class implementing virtual functions is shown in Figs. 115 and 116.

```cpp
class Solid
{
protected:
    float       x, y, z;
    float       density;

public:
    // CONSTRUCTOR
    Solid( float density );

    // MEMBER FUNCTIONS
    virtual float   get_volume();
    float           get_weight();
    void            set_location( float x, float y, float z );
};
```

*Figure 115 - Solid Class Declaration Using Virtual Functions*
Figure 116 - Solid Class Definition Using Virtual Functions

Once the features of inheritance and polymorphism have been implemented into a design, the construction of new classes is made much easier. Only features specific to the new class need to be implemented. Consider the declarations and definitions of the Cylinder and Block classes shown in Figs. 117 through 120. The code is considerably shorter than the original code developed in Fig. 107 through 110.

Figure 117 - Cylinder Class Declaration Using Inheritance and Virtual Functions
Cylinder::Cylinder( float r, float h ) : Solid( 2.0 )
{
    radius = r;
    height = h;
}

//........................................................................
float Cylinder::get_volume()
{
    float pi = 3.141593;
    return pi*radius*radius*height;
}

Figure 118 - Cylinder Class Definition Using Inheritance and Virtual Functions

class Block : public Solid
{
private:
    float width, height, depth;

public:
    // CONSTRUCTOR
    Block( float w, float h, float d );

    // MEMBER FUNCTIONS
    float get_volume();
};

Figure 119 - Block Class Declaration Using Inheritance and Virtual Functions

Block::Block( float w, float h, float d ) : Solid( 1.0 )
{
    width = w;
    height = h;
    depth = d;
}

//........................................................................
float Block::get_volume()
{
    return width*height*depth;
}

Figure 120 - Block Class Declaration Using Inheritance and Virtual Functions
Appendix C  Responsibility-Driven Design

This appendix presents a review of the Responsibility-Driven Design process. The process is summarized from the text, “Designing Object-Oriented Software,” by Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener [Wirf90].

Responsibility-Driven Design breaks the design process down into two phases:

- **Exploratory Phase** - This phase provides an initial description for the system. A general set of classes, responsibilities, and collaborations is identified.
- **Analysis Phase** - This phase examines the output of the exploratory phase in an attempt to improve the design. Hierarchies, subsystems, and protocols are identified and optimized.

**C.1 Exploratory Phase**

The exploratory phase defines the system from an abstract point of view. The design specification if first read and clearly interpreted. The goal of the exploratory phase is to identify three things:

- **Classes** - abstractions useful in defining acting members of an application.
- **Responsibilities** - duties that a class must perform.
- **Collaborations** - connections between classes that are used to fulfill responsibilities.
C.1.1 Classes

Classes may be built by extracting the noun phrases from the specification. In general, the following guidelines may be used when initially identifying classes:

- **Model physical objects.** Objects such as an airplane, spring, or car are the easiest type of class to identify.

- **Model conceptual entities.** Items such as complex numbers or matrices often make appropriate classes.

- **Use a single term for each concept.** The vocabulary used to describe the system of classes is very important. Use only one name for each class. Try to form patterns for names, but make sure that a name chosen for a class accurately describes what the class is or does.

- **Be wary of the use of adjectives.** Adjectives can describe the characteristics or behavior of a class. The “blue wing” and “the red wing” are two objects of the same class with different parameters. The “main wing” and the “tail wing” are two different objects of different classes. These objects exhibit distinct behavior and are therefore not members of the same class.

- **Model categories of objects.** Defining classes such as car, truck, and motorcycle may also lead to the development of a vehicle class. This class acts as an abstraction mechanism for the classes in the vehicle category.

- **Model external interfaces.** Programs often need to interact with outside sources such as a network, a file system, or other programs. An interface to these external sources can be created by encapsulating a class around them.

- **Model the values of an object’s attributes.** An easy mistake to make is to create a class that really has no function. For instance, “the altitude of an aircraft” may be a noun phrase that was extracted from the specification. If altitude is defined as a real number, then perhaps “real” is the class that needs to be modeled rather than “altitude.”
At this stage a tentative list of classes has been identified. In general, as the design process continues, more classes will be added to this list than will be removed.

After the first set of classes has been identified, try to group the classes by category. This will identify the abstract classes - the categories. The goal is to identify as many abstract classes as possible. If an abstract class does not already exist identifying the category, create it. Abstract classes help manage common responsibilities of a group of classes and provide structure for a program.

C.1.2 Responsibilities

Defining responsibilities for classes helps to refine the classes. It also points out strengths and weaknesses of a design. The responsibilities of a class include:

- the knowledge maintained in the class
- the actions the class can perform

Responsibilities convey a sense of purpose for an object and help define its place in the system.

A useful mechanism for defining responsibilities is to perform a walk-through of the system. For each walk-through, note what each class is doing and record the responsibility. A good starting point is to look at the inputs to the system. Which class is responsible for handling the input? Where does control pass to next?

For assistance in assigning responsibilities, use the following guidelines:

- **Evenly distribute system intelligence.** The more evenly intelligence is distributed among classes, the less knowledge is required by each class. This results in classes
that are easier to implement and a system that as a whole is easier to maintain. The intelligence of a class is roughly equivalent to the number of responsibilities it has.

- **State responsibilities as generally as possible.** Leave the details for later in the design process. By stating responsibilities generally, it is possible to see the “big picture” of the design. It is also easier to organize the design.

- **Keep behavior with related information, if any.** If an object is responsible for maintaining some sort of information, then that object should also be responsible for performing any necessary operations on the information. Likewise, if a class is responsible for performing an operation on some information or needs information to perform some responsibility, then the class should also be responsible for the information.

- **Keep information about one thing in one place.** Try to keep the information pertaining to one thing in one specific place. Having two classes responsible for the same information leads to confusion and inconsistency in a design. There are three possible solutions to maintaining information that is needed by two classes:

  1. Create a new class whose sole responsibility is to maintain the information. Other classes that need the information will query this class for it.
  2. Reassign the responsibility to one of the classes. Have other objects requiring the information query it from this object.
  3. Collapse the different objects requiring the information into a single object.

- **Share responsibilities among related objects.** Some responsibilities cannot be assigned to a single object. Break these responsibilities into smaller chunks that can be handled by individual objects. It is okay to assign the group of responsibilities together to form a larger responsibility.
Additional responsibilities can be identified by examining the relationships between classes. The principle relationships between classes of concern at this point are:

- **the “is-kind-of” relationship** - a group of classes that are a kind of another thing indicates the presence of a subclass-superclass relationship. The superclass should have responsibilities that are common to all the subclasses. Assigning the responsibilities that are common to all the subclasses to the superclass makes creating the subclass easier.

- **the “is-analogous-to” relationship** - this relationship often indicates the presence of a missing superclass between two or more classes. Identifying this superclass may help to redefine responsibilities in the system.

- **this “is-part-of” relationship** - this relationship identifies a parent-child type relationship. The parent is responsible for maintaining one or more children.

### C.1.3 Collaborations

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. An object collaborates with another if, to fulfill a responsibility, it needs to send the other object any messages. It may take several collaborations to completely fulfill a single responsibility. The patterns of collaborations within an application reveal the flow of control and information during its execution. A collaboration without a responsibility indicates a missed responsibility.

As with identifying responsibilities, analyzing relationships between classes often reveals collaborations. The most pertinent relationships are:

- **the “is-part-of” relationship** - this parent-child relationship may not indicate collaborations. However, a parent class often fulfills its responsibilities by delegating them to one or more of its parts.
• **the “has-knowledge-of” relationship** - if an object has knowledge of something about another object, it is often indicative of the need for that knowledge. Attaining the knowledge implies a collaboration.

• **the “depends” on relationship** - if an object depends on another object, it will often undergo some transformation when the first object changes. The connection between these objects must be maintained with some collaboration.

### C.2 Analysis Phase

This phase of the design examines what has been created thus far. Several tools will be used to analyze what has been created and make modifications as necessary. The tools include:

• **hierarchy graphs**
  • inheritance hierarchies
  • Venn diagrams

• **contracts**
  • subsystems
  • collaborations graphs

#### C.2.1 Hierarchies

Hierarchy graphs are designed to illustrate the relationships between classes. They provide a form of responsibility grouping.

#### C.2.1.1 Inheritance Hierarchies

An example hierarchy graph is shown in Fig. 121. The figure shows two kinds of objects: concrete and abstract. Concrete classes are designed to be instantiated. Abstract classes are designed to be inherited from. Abstract classes are a main mechanism of
reuse in object-oriented design. Abstract classes exist solely to factor out behavior that is common to more than one subclass.

![Class Hierarchy Diagram]

**Figure 121 - Example Class Hierarchy**

Abstract classes are indicated in the hierarchy graph by filling in the upper left corner of the box.

**C.2.1.2 Venn Diagrams**

Venn diagrams can also be used in creating hierarchies. If a class is viewed as a set of responsibilities, then the intersection between two classes can be defined as a super class. A sample Venn diagram is shown in Fig. 122.
C.2.1.3 Guidelines for Building Hierarchies

The following guidelines can be used to evaluate the hierarchies:

- **Model a “kind-of” hierarchy.** Subclasses should inherit all the responsibilities of their superclasses. If a subclass does not support all the responsibilities of a superclass, then a new class needs to be created and the hierarchy rearranged. The new class should be an abstract class that supports the common responsibilities of the two previous classes. The previous classes should then inherit from the new class.

- **Factor common responsibilities as high as possible.** If a set of classes all support a common responsibility, then those classes should inherit that responsibility from a common superclass. This new superclass will probably be an abstract class. The more responsibilities that can be factored out of classes, the more robust a design will be. Abstract classes also provide convenient tie-in points for future modification of the design.

- **Make sure that abstract classes do not inherit from concrete classes.** Abstract classes support responsibilities in implementation-independent ways. Using polymorphism, responsibilities are satisfied by subclasses rather than in the actual
class itself. Concrete classes satisfy actual responsibilities. Inheriting an abstract class from a concrete class ties the hierarchy into a specific implementation.

- **Eliminate classes that do not add functionality.** If a class does not have any responsibilities, then it is not needed. Classes that do not define responsibilities, but do implement responsibilities inherited from abstract classes are needed.

C.2.2 Contracts

Grouping responsibilities into contracts is another form of abstraction that helps clarify a design. A contract defines a set of requests that a client can make of a server. A class can support more than one distinct contract. A contract differs from a responsibility in that a contract is a cohesive set of responsibilities that a client can depend on.

Each responsibility is part of at most one contract, but not all responsibilities are part of a contract. Some responsibilities represent behavior a class must have, but which cannot be requested by other objects. These responsibilities are referred to as private responsibilities.

Responsibilities can be partitioned into contracts using the following guidelines:

- **Group responsibilities that are used by the same clients.** A client class usually interacts with a server through one or two contracts. The contracts represent several responsibilities. If a client class interacts with all responsibilities of a server, it is possible for all the responsibilities to be grouped into one contract. A class may only support one contract.

- **Maximize the cohesiveness of classes.** Just as a contract should be composed of a cohesive set of responsibilities, a class should support a cohesive set of contracts. If a class defines a contact that has relatively little in common with the rest of the contracts defined by that class, it should be moved to a different class, usually a superclass or a subclass.
- **Minimize the number of contracts per class.** The fewer details a system has, the more comprehensible it is. In order to maximize reuse, a set of classes all supporting a common contract should inherit that contract from a common superclass. A contract should be defined by exactly one class, and supported by all of its subclasses.

To apply these guidelines, start by defining contracts for classes at the top of the inheritance hierarchies. New contracts should be defined only for classes that add significant new functionality. Examine the responsibilities added by each subclass and determine whether they represent new functionality, or whether they are just more specific ways of expressing inherited responsibilities, and are therefore part of the inherited contract.

**C.2.2.1 Subsystems**

The goal of analyzing a design using subsystems is to streamline the collaborations between classes. A subsystem is a group of classes that collaborate among themselves to support a set of contracts. From outside the subsystem, the group of classes can be viewed as working closely together to provide a clearly delimited unit of functionality. Subsystems are a concept used to simplify a design. It should be noted that subsystems are only conceptual entities; they do not exist during program execution.

The complexity of a large application can be dealt with by first identifying subsystems within it and treating those subsystems as classes. An application can be decomposed into subsystems and repeatedly decomposed until all required richness and detail have been modeled.

**C.2.2.2 Collaborations Graphs**

Collaborations, as previously discussed, represent a client-server connection between two classes. Servers fulfill requests from clients via contracts. A collaborations graph is like
a time-lapsed photograph of classes and the collaborations between them. All possible pathways are simultaneously shown.

The basic building block of a collaboration graph is the class as shown in Fig. 123. A class is represented by a block. Contracts that the class can provide are represented by half-circles and a unique number representing the contract.

![Figure 123 - Example Class and Contract](image)

Inheritance is represented by enclosing the subclass with the superclass as shown in Fig. 124. Any box enclosed within another supports all the contracts supported by the outer box. This is consistent with the concept of inheritance.

![Figure 124 - Collaborations Graph Inheritance Representation](image)

A collaboration represents one class using a contract of another. This is represented by a line from the client to the server as shown in Fig. 125.
Figure 125 - Example Collaboration

A complete collaboration graph displays all collaborations between all classes. As can be imagined, this can be quite large and complex. To simplify this display, subsystems are employed. A subsystem is displayed as shown in Fig. 126.

Figure 126 - Example Collaborations Graph

Some subsystems can be identified before constructing a collaborations graph. Others will be identified once the initial graph is created. Subsystems can be identified by frequent and complex collaborations. Use the following guidelines for identifying subsystems.
- Classes in a subsystem should collaborate to support a small and strongly cohesive set of responsibilities.
- Classes within a subsystem should be strongly interdependent.

Simplify the collaborations between and within subsystems by using the following guidelines:

- Minimize the number of collaborations a class has with other classes or subsystems.
- Minimize the number of classes and subsystems to which a subsystem delegates.
- Minimize the number of different contracts supported by a class or a subsystem.
Scott Anthony Woyak was born on June 6\textsuperscript{th}, 1969. He grew up knowing that he wanted to build things. Recognizing that computers were going to be an important part of the future, he chose to attend Virginia Tech because they required all engineering students to purchase one. Little did he know that that first exposure would provide a future career. For his masters, Scott created an object-oriented framework for building three-dimensional user-interfaces. That project was enough to convince Scott that programming was the way to go. Scott’s dissertation work applies his programming skills to building tools for engineers. In retrospect, Scott has realized that his interests have not really changed since he was a child. It is not the programming that draws him to computers, but the ability to create whatever he wants. Computers are after all, the ultimate medium for building things.