UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

CLAYTON KOSSOSKI

PROPOSTA DE UM MÉTODO DE TESTE PARA PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE USANDO O PARADIGMA ORIENTADO A NOTIFICAÇÕES

DISSERTAÇÃO

CURITIBA
2015

CLAYTON KOSSOSKI

PROPOSTA DE UM MÉTODO DE TESTE PARA PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE USANDO O PARADIGMA ORIENTADO A NOTIFICAÇÕES

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" – Área de Concentração: Engenharia de Computação.

Orientador: Prof. Dr. Paulo Cézar Stadzisz.

Coorientador: Prof. Dr. Jean Marcelo Simão.

CURITIBA
2015

Dados Internacionais de Catalogação na Publicação

K86p Kossoski, Clayton
2015 Proposta de um método de teste para processos de desenvolvimento de software usando o paradigma orientado a notificações / Clayton Kossoski.-- 2015.
268 f.: il.; 30 cm

Texto em português, com resumo em inglês.
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Curitiba, 2015.
Bibliografia: f. 193-202.

1. Paradigma orientado a notificações. 2. Software - Desenvolvimento. 3. Software - Testes. 4. Software - Controle de qualidade. 5. Software - Avaliação. 6. Métodos de simulação. 7. Engenharia de software. 8. Engenharia elétrica - Dissertações. I. Stadzisz, Paulo Cézar, orient. II. Simão, Jean Marcelo, coorient. III. Universidade Tecnológica Federal do Paraná - Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD 22 -- 621.3

Biblioteca Central da UTFPR, Câmpus Curitiba

UTFPR
CAMPUS CURITIBA

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Câmpus Curitiba

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL
CPGEI

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Título da Dissertação Nº 694

Proposta de um Método de Teste para Processos de Desenvolvimento de Software Usando o Paradigma Orientado a Notificação.

por

Clayton Kossoski

Orientador: Prof. Dr. Paulo César Stadzisz
Coorientador: Prof. Dr. Jean Marcelo Simão

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: **Engenharia de Computação** do Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR, às **14:00h** do dia **19 de agosto de 2015**. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores doutores:

Prof. Dr. Paulo César Stadzisz
(Presidente – UTFPR)

Prof. Dr. Marcos Antônio Quináia
(UNICENTRO)

Profa. Dra. Maria Claudia F. Pereira Emer
(UTFPR)

Visto da coordenação:

Prof. Dr. Emilio Carlos Gomes Wille
(Coordenador do CPGEI)

Dedico este trabalho à minha família e amigos.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela vida, saúde e perseverança concedida, elementos fundamentais que me permitiram concluir mais este trabalho.

Especialmente aos meus pais Anita e Demétrio, exemplos de trabalho, determinação e união, agradeço pela vida e a educação concedida a mim e ao meu irmão, Adriano. Além do apoio incondicional em todas as vezes que precisei.

Agradeço imensamente ao meu orientador Prof. Dr. Paulo Cezar Stadzisz e ao coorientador Prof. Dr. Jean Marcelo Simão, por todos os ensinamentos, direcionamentos, incentivos e pelo tempo despendido nas leituras e sugestões sobre os meus trabalhos. Também, agradeço pela confiança depositada em mim por ter me concedido a oportunidade de cumprir mais esta etapa da minha vida.

Agradeço ao meu amigo Galvão pelas palavras de incentivo durante os estudos realizados no CPGEI.

Agradeço aos colegas do grupo do PON e de outros grupos de estudos que fiz amizades.

Gostaria de deixar registrado também, o meu reconhecimento a outros entes da família e amigos, que me incentivaram a continuar os estudos.

Agradeço ao CPGEI, pela cooperação.

Agradeço também a CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro e à UTFPR/CPGEI/LSIP (Laboratório de Sistemas Inteligentes de Produção) por todo o suporte oferecido durante ao desenvolvimento deste trabalho.

Enfim, a todos os que por algum motivo contribuíram para a realização deste trabalho, seja por atos ou pensamentos.

Aventura é fazer algo por impulso, sem planejamento. Audácia é quando pensamos em todas as possibilidades de erros, acertos e planejamos nossas atitudes.

Mário Sergio Cortella

RESUMO

KOSSOSKI, Clayton. **Proposta de um método de teste para processos de desenvolvimento de software usando o Paradigma Orientado a Notificações**. 2015. [268 f]. Dissertação (Mestrado em Engenharia de Computação) – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2015.

O Paradigma Orientado a Notificações (PON) é uma alternativa para o desenvolvimento de aplicações em software e propõe resolver certos problemas existentes nos paradigmas usuais de programação, nomeadamente o Paradigma Declarativo (PD) e o Paradigma Imperativo (PI). Na verdade, o PON unifica as principais vantagens do PD e do PI, ao mesmo tempo que resolve (em termos de modelo) várias de suas deficiências e inconvenientes relativas ao cálculo lógico-causal em aplicações de software, supostamente desde ambientes monoprocessados a completamente multiprocessados. O PON tem sido materializado em termos de programação e modelagem, mas ainda não possuía um método formalizado para orientar os desenvolvedores na elaboração de teste de software. Esta dissertação propõe um método de teste para projetos de software que empregam o PON no seu desenvolvimento. O método de teste de software proposto foi desenvolvido para ser aplicado nas fases de teste unitário e teste de integração. O teste unitário considera as menores entidades testáveis do PON e requer critérios de teste específicos. O teste de integração considera o funcionamento das entidades PON em conjunto para realização de casos de uso e pode ser realizado em duas etapas: (1) teste sobre as funcionalidades descritas nos requisitos e no caso de uso e (2) teste que exercitem diretamente as entidades PON que compõem o caso de uso (como *Premisses*, *Conditions* e *Rules*). Esse método de teste foi aplicado em um caso de estudo que envolve a modelagem e desenvolvimento de um software de combate aéreo e os resultados desta pesquisa mostram que o método proposto possui grande importância no teste de programas PON.

Palavras-chave: Paradigma Orientado a Notificações. Teste de Software. Teste Unitário. Teste de Integração.

ABSTRACT

KOSSOSKI, Clayton. **A proposal of a test method for software development processes using the Notification Oriented Paradigm**. 2015. [268 f]. Dissertação (Mestrado em Engenharia de Computação) – Graduate Program in Electrical and Computer Engineering, Federal University of Technology – Parana. Curitiba, 2015.

The Notification Oriented Paradigm (NOP) is an alternative to the development of software applications and proposes to solve certain problems in the usual programming paradigms, including the Declarative Paradigm (DP) and Imperative Paradigm (IP). Indeed, the NOP unifies the main advantages of DP and IP while solving (in terms of model) several of its deficiencies and inconveniences related to logical-causal calculation, apparently from both mono and multiprocessor environments. The NOP has been materialized in terms of programming and modeling, but still did not have a formalized method to guide developers in designing and software testing activity. This dissertation proposes a test method for software projects that use the NOP in its development. The proposed software testing method was developed for use in the phases of unit testing and integration testing. The unit testing considers the smallest testable entities of the NOP and requires specific techniques for generating test cases. The integration testing considers the operation of the PON entities together to carry out use cases and can be accomplished in two steps: (1) test on the features described in the requirements and use case and (2) test that directly exercise the NOP entities that make up the use case (as *Premisses*, *Conditions* and *Rules*). The test method was applied in a case study involving the modeling and development of a simple air combat and the results of this research show that the proposed method has great importance in testing NOP programs in both unit and integration testing.

Keywords: Notification Oriented Paradigm. Software Testing. Unit Testing. Integration Testing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama de classes conceitual do PON e suas entidades 36
Figura 2 – Cadeia de notificação das entidades do PON...37
Figura 3 – Exemplo de uma *Rule* e entidades relacionadas 38
Figura 4 – Modelo centralizado de resolução de conflitos...39
Figura 5 – Mecanismo de extensão da UML 44
Figura 6 – Estrutura do *framework* PON Otimizado 44
Figura 7 – Estrutura do pacote *Core* 45
Figura 8 – *NOP Profile* pacote *Core*...46
Figura 9 – *NOP Profile Application* pacote *Application*..47
Figura 10 – DON contextualizado no RUP...48
Figura 11 – Resumo dos ciclos do método DON 50
Figura 12 – Elementos básicos que compõem um diagrama de casos de uso.........55
Figura 13 – Exemplo de grafo de fluxo de controle de um programa *P* 58
Figura 14 – Grafo de fluxo de controle para o método runGame 61
Figura 15 – Processo genérico do teste de mutação 66
Figura 16 – Tela do software em execução 71
Figura 17 – Comparação entre plano cartesiano e o plano de pixels no Allegro.......72
Figura 18 – Diagrama de casos de uso do software 76
Figura 19 – Diagrama de classes resumido do software..77
Figura 20 – Diagrama de componentes (passo 1): *Rule* com nome, *Condition* e *Method* 81
Figura 21 – Diagrama de componentes (passo 2): *Rule* e suas interfaces 84
Figura 22 – Diagrama de componentes (passo 2): Operadores lógicos 85
Figura 23 – Diagrama de componentes (passo 3): *Rule*, interfaces e componentes externos 86
Figura 24 – Diagrama de sequência do caso de uso Controlar Avião (rlAirplaneMovingLeft) 88
Figura 25 – Diagrama de sequência do caso de uso Controlar Avião (rlAirplaneMovingRight)..89
Figura 26 – Diagrama de sequência do caso de uso Controlar Avião (rlAirplaneShoots)..90
Figura 27 – Elementos do novo diagrama de objetos PON 92
Figura 28 – *Rule* rlAirplaneMovingRight, suas entidades e relacionamentos representados com o diagrama de objetos PON..92
Figura 29 – Fases e atividades do método de teste de software em PON................96
Figura 30 – Visão expandida da fase de teste unitário...98
Figura 31 – Classes de equivalência e análise de valores limite para prAirplaneLimRightScreen...102
Figura 32 – Visão expandida da fase de teste de integração.................................123
Figura 33 – Fluxo básico e fluxos alternativos de eventos em um caso de uso......125

Figura 34 – Diagrama de objetos do caso de uso Controlar Avião 137
Figura 35 – Visão interna do pacote rlAllegroDrawBullet 138
Figura 36 – Visão interna do pacote rlAllegroMoveTheBulletUp 138
Figura 37 – Diagrama de fluxo de notificações do caso de uso Controlar Avião.....145
Figura 38 – Diagrama de objetos do caso de uso Realizar Leitura de Teclado 165
Figura 39 – Diagrama de objetos do caso de uso Controlar Helicóptero 169
Figura 40 – Diagrama de objetos do caso de uso Controlar Apresentação 173
Figura 41 – Diagrama de objetos do caso de uso Pausar e Continuar Jogo 176
Figura 42 – Diagrama de objetos do caso de uso Parar Jogo...............................179
Figura 43 – Diagrama de objetos do caso de uso Detectar Colisão de Projétil.......183
Figura 44 – Falta de conexões entre objetos 252
Figura 45 – Excesso de conexões entre objetos..253
Figura 46 – Ausência de *Attribute* em *FBE* 253
Figura 47 – *Method* que não altera *Attribute* 254
Figura 48 – *Attribute* que nunca é alterado 254
Figura 49 – *Premise* compartilhada entre *Rules* 255
Figura 50 – *Method* compartilhado entre *Rules*..255
Figura 51 – *Master Rule* compartilhada 256
Figura 52 – *Attribute* compartilhado entre *Premisses*...256
Figura 53 – *Attribute* alterado por mais de um *Method* 257
Figura 54 – Objetos que participam da implementação de vários casos de uso.....258
Figura 55 – Representação esquemática para obtenção da rede de alcançabilidade PON 259
Figura 56 – Interseção para visualização de *Premisses* redundantes. 268

LISTA DE TABELAS

Tabela 1 – Exemplo de uma tabela de decisão para um caso de uso qualquer 57
Tabela 2 – Cobertura de caminhos no método runGame 61
Tabela 3 – Requisitos funcionais, não funcionais e de *design* do software...............74
Tabela 4 – Descrição sucinta de cada caso de uso 76
Tabela 5 – Relação entre os casos de uso e seus requisitos....................................78
Tabela 6 – *Rule* 1: Movimentar o cenário para baixo 79
Tabela 7 – *Rule* 2: Atualizar o progresso do jogador 79
Tabela 8 – *Rule* 3: Manter a posição do Avião...80
Tabela 9 – *Rule* 4: Movimentar Avião à esquerda..80
Tabela 10 – *Rules* identificadas para o software..81
Tabela 11 – *Rules*, *Premisses* e *Instigations* identificados..82
Tabela 12 – Plano de teste para uma *Premise* 100
Tabela 13 – Exemplo de descrição de caso de teste no plano de teste..................100
Tabela 14 – Estrutura da *Premise* prAirplaneLimRightScreen 101
Tabela 15 – Classes de equivalência e análise de valor limite para a *Premise* prAirplaneLimRightScreen...102
Tabela 16 – Exemplo de descrição de caso de teste para prAirplaneLimRightScreen 103
Tabela 17 – Casos de teste para a *Premise* prAirplaneLimRightScreen.................103
Tabela 18 – Classes de equivalência e análise de valor limite para a *Premise* prBulletXGreaterX1Helicopter 104
Tabela 19 – Casos de teste para a *Premise* prBulletXGreaterX1Helicopter 104
Tabela 20 – Estrutura de uma *Condition* que avalia quatro *Premisses*...................106
Tabela 21 – Caso de teste para a *Condition* da *Rule* rlAirplaneMovingRight..........106
Tabela 22 – Casos de teste para a *Condition* da *Rule* rlAirplaneMovingRight 107
Tabela 23 – *Condition* que avalia uma disjunção de duas *Premisses*.....................107
Tabela 24 – Casos de teste para a *Condition* da *Rule* rlAllegroBlit que avalia uma disjunção de duas *Premisses*...107
Tabela 25 – *Condition* que avalia conjunção de duas *SubConditions*.....................108
Tabela 26 – Casos de teste para a *SubCondition 1* 108
Tabela 27 – Casos de teste para a *SubCondition 2* 108
Tabela 28 – Casos de teste para uma a *Condition* que avalia duas *SubConditions* 108
Tabela 29 – *Condition* que avalia uma *Premise* com *Attribute* impertinente 109
Tabela 30 – Execução de casos de teste para uma *Condition* que avalia uma *Premise* com *Attribute* impertinente 109
Tabela 31 – *Rule* que avalia apenas uma *Condition* 110
Tabela 32 – Caso de teste para a *Rule* rlAirplaneMovingRight..............................110
Tabela 33 – Execução de casos de teste para a *Rule* rlAirplaneMovingRight 111
Tabela 34 – Plano de teste para o *Method* mtAirplaneAddNewBullet.....................116

Tabela 35 – Classes de equivalência e análise de valor limite para exercitar mtAirplaneAddNewBullet....118
Tabela 36 – Caso de teste para mtAirplaneAddNewBullet....119
Tabela 37 – Casos de teste que exercitam valores limite do *Method* 119
Tabela 38 – Casos de teste para o *Method* mtAirplaneAddNewBullet....120
Tabela 39 – Determinação de número de casos de teste....120
Tabela 40 – Plano de Teste para o caso de uso Controlar Avião 127
Tabela 41 – Descrição do caso de uso Controlar Avião....129
Tabela 42 – Matriz de identificação de cenários do caso de uso Controlar Avião...131
Tabela 43 – Matriz de Casos de Teste do caso de uso Controlar Avião....132
Tabela 44 – Classes de equivalência e análise de valores limite para as variáveis operacionais....133
Tabela 45 – Matriz de Casos de Teste com valores (caso de uso Controlar Avião) 134
Tabela 46 – Variáveis operacionais e *Attributes* correspondentes....139
Tabela 47 – Matriz de Casos de Teste para o Caso de Uso Controlar Avião 140
Tabela 48 – Matriz de Casos de Teste para o Caso de Uso Controlar Avião 142
Tabela 49 – Determinação do número de casos de teste possíveis com análise de valores limite 143
Tabela 50 – Notação do grafo de fluxo de notificações....145
Tabela 51 – Alguns casos de teste para exercitar o fluxo de notificações do caso de uso Controlar Avião....146
Tabela 52 – Estrutura da *Premise* prAirplaneLimRightScreen 149
Tabela 53 – Estrutura de uma *Condition* que avalia quatro *Premisses*....149
Tabela 54 – *Rule* rlAirplaneMovingRight que avalia apenas uma *Condition*....150
Tabela 55 – *Premisses* agrupadas por características semelhantes 154
Tabela 56 – Classes de equivalência e análise de valor limite para a *Premise* prPauseButtonTrue 155
Tabela 57 – Casos de teste para a *Premise* prPauseButtonTrue 155
Tabela 58 – Classes de equivalência e análise de valor limite para a *Premise* prBulletXGreaterX1Airplane 155
Tabela 59 – Casos de teste para a *Premise* prBulletXGreaterX1Airplane 156
Tabela 60 – Classes de equivalência e análise de valor limite para a *Premise* prAirplaneLimLeftScreen....156
Tabela 61 – Casos de teste para a *Premise* prAirplaneLimLeftScreen....156
Tabela 62 – Classes de equivalência e análise de valor limite para a *Premise* prHelicopterLifePointsZero....157
Tabela 63 – Casos de teste para a *Premise* prAirplaneLimLeftScreen....157
Tabela 64 – *Conditions* e *SubConditions* agrupadas por características semelhantes 158
Tabela 65 – *Condition* pertencente a *Rule* rlAirplaneMovingRight 158
Tabela 66 – Alguns casos de teste para *Condition* que pertence a *Rule* rlAiplaneMovingRight 159

Tabela 67 – *Condition* pertencente a *Rule* rlScenarioDrawing 159
Tabela 68 – Casos de teste para a *Condition* da *Rule* rlScenarioDrawing 159
Tabela 69 – *Rule* que avalia apenas uma *Condition* 160
Tabela 70 – Casos de teste para a *Rule* rlAirplaneMovingRight 160
Tabela 71 – *Methods* do *FBE* Airplane...160
Tabela 72 – *Methods* do *FBE* Helicopter...161
Tabela 73 – *Methods* do *FBE* Stage 1 161
Tabela 74 – *Methods* do *FBE* Allegro Bullet..161
Tabela 75 – Caso de teste do *Method* mtHelicopterAddNewBullet 162
Tabela 76 – Classes de equivalência e análise de valor limite para exercitar mtHelicopterAddNewBullet...162
Tabela 77 – Casos de teste para o *Method* mtHelicopterAddNewBullet 162
Tabela 78 – Descrição do caso de uso Realizar Leitura de Teclado.......................163
Tabela 79 – Classes de equivalência e análise de valores limite para o caso de uso Realizar Leitura de Teclado...164
Tabela 80 – Matriz de Casos de Teste com valores (caso de uso Realizar Leitura de Teclado) 164
Tabela 81 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Realizar Leitura de Teclado...165
Tabela 82 – Matriz de Casos de Teste para o Caso de Uso Realizar Leitura de Teclado 166
Tabela 83 – Descrição do caso de uso Controlar Helicóptero.................................167
Tabela 84 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Controlar Helicóptero 167
Tabela 85 – Matriz de Casos de Teste com valores (caso de uso Controlar Helicóptero)...168
Tabela 86 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Controlar Helicóptero 170
Tabela 87 – Matriz de Casos de Teste para o Caso de Uso Controlar Helicóptero 170
Tabela 88 – Determinação do número de casos de teste possíveis com análise de valores limite 170
Tabela 89 – Descrição do caso de uso Controlar Apresentação.............................171
Tabela 90 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Controlar Apresentação 172
Tabela 91 – Matriz de Casos de Teste com valores (caso de uso Controlar Apresentação)..172
Tabela 92 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Controlar Apresentação 174
Tabela 93 – Matriz de Casos de Teste para o Caso de Uso Controlar Apresentação 174
Tabela 94 – Determinação do número de casos de teste possíveis com análise de valores limite 174
Tabela 95 – Descrição do caso de uso Pausar e Continuar Jogo............................175

Tabela 96 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Pausar e Continuar Jogo 175
Tabela 97 – Matriz de Casos de Teste com valores (caso de uso Pausar e Continuar Jogo) 176
Tabela 98 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Pausar e Continuar Jogo....................................176
Tabela 99 – Matriz de Casos de Teste para o Caso de Uso Pausar e Continuar Jogo 177
Tabela 100 – Determinação do número de casos de teste possíveis com análise de valores limite 177
Tabela 101 – Descrição do caso de uso Parar Jogo....................................178
Tabela 102 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Parar Jogo 178
Tabela 103 – Matriz de Casos de Teste com valores (caso de uso Parar Jogo) 178
Tabela 104 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Parar Jogo 179
Tabela 105 – Matriz de Casos de Teste para o Caso de Uso Parar Jogo 179
Tabela 106 – Determinação do número de casos de teste possíveis com análise de valores limite 179
Tabela 107 – Descrição do caso de uso Detectar Colisão de Projétil.....................180
Tabela 108 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Detectar Colisão de Projétil 181
Tabela 109 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Detectar Colisão de Projétil 183
Tabela 110 – Matriz de Casos de Teste com valores (caso de uso Detectar Colisão de Projétil) 184
Tabela 111 – Matriz de Casos de Teste para o Caso de Uso Detectar Colisão de Projétil 184
Tabela 112 – Descrição do caso de uso Iniciar Jogo 185
Tabela 113 – *Rule* 5: Movimentar Avião à direita....................................204
Tabela 114 – *Rule* 6: Disparar projétil contra o Helicóptero 204
Tabela 115 – *Rule* 7: Disparar projétil contra o Avião 204
Tabela 116 – *Rule* 8: Apresentar o Helicóptero no tela....................................205
Tabela 117 – *Rule* 9: Decrementar pontos de vida do Avião 205
Tabela 118 – *Rule* 10: Decrementar pontos de vida do Helicóptero 205
Tabela 119 – *Rule* 11: Eliminar o Helicóptero 206
Tabela 120 – *Rule* 12: Eliminar o jogador 206
Tabela 121 – *Rule* 13: Pausar o jogo 206
Tabela 122 – *Rule* 14: Continuar o jogo....................................206
Tabela 123 – *Rule* 15: Parar o jogo....................................207
Tabela 124 – *Rule* 16: Limpar a tela (Allegro)....................................207
Tabela 125 – *Rule* 17: Desenhar todos os elementos (Allegro).............................207
Tabela 126 – *Rule* 18: Comandos do teclado (Allegro)....................................208
Tabela 127 – *Rule* 19: Desenhar o Avião (Allegro) 208

Tabela 128 – *Rule* 20: Desenhar o cenário de fundo 208
Tabela 129 – *Rule* 21: Desenhar projétil 208
Tabela 130 – *Rule* 22: Detectar colisão com Helicóptero (Allegro) 209
Tabela 131 – *Rule* 23: Detectar colisão de projétil com Avião (Allegro)..................209
Tabela 132 – *Rule* 24: Apresentar a trajetória ascendente de um projétil disparado 209
Tabela 133 – *Rule* 25: Movimentar Helicóptero na tela...210
Tabela 134 – *Rule* 26: Apresentar a trajetória descendente de um projétil disparado 210
Tabela 135 – *Attributes* e suas lista de inscrições de *Premisses*............................263
Tabela 136 – *Premisses* e suas listas de inscrições de *Conditions* 263
Tabela 137 - *Rules* e suas listas de execução de *Methods* 264
Tabela 138 – *Attributes* alterados pelos *Methods* e o impacto nas *Premisses* 264
Tabela 139 - Exemplo de uso predicativo em *Premise* 266
Tabela 140 - Exemplo de uso predicativo em *Rule* 266

LISTA DE ABREVIATURAS

i.e.	id est (isto é, ou seja)
v.g.	verbi gratia (por exemplo)
e.g.	exempli gratia (por exemplo)
op.	operador

LISTA DE SIGLAS

2D	2 Dimensões
API	*Application Programming Interface*
ARQPON	Arquitetura PON
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
COMTEL	*Congreso Internacional de Computación y Telecomunicaciones*
CPGEI	Programa de pós-graduação em Engenharia Elétrica e Informática Industrial
DA	Diagrama de Atividades
DFN	Diagrama de Fluxo de Notificações
DON	Desenvolvimento Orientado a Notificações
ES	Engenharia de Software
FBE	*Fact Base Element*
FIFO	*First In, First Out*
FPGA	*Field Programmable Gate Array*
GFC	Grafo de Fluxo de Controle
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronic Engineers*
LIFO	*Last In, First Out*
LingPON	Linguagem PON
OO	Orientado a Objetos
PD	Paradigma Declarativo
PF	Paradigma Funcional
PI	Paradigma Imperativo
PL	Paradigma Lógico
POE	Programação Orientada a Eventos
PON	Paradigma Orientado a Notificações
POO	Paradigma Orientado a Objetos
RNA	Rede Neural Artificial
SBR	Sistema Baseado em Regras
UC	*Use Case*
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO..20
1.1 CONTEXTO DO TRABALHO E PESQUISA..20
1.2 PROBLEMA ABORDADO 23
1.3 OBJETIVOS..24
1.3.1 Objetivo Geral..24
1.3.2 Objetivos Específicos..25
1.4 MOTIVAÇÃO E JUSTIFICATIVA 25
1.5 MÉTODO 26
1.6 ESTRUTURA DO TRABALHO 27
2 FUNDAMENTAÇÃO TEÓRICA...29
2.1 PARADIGMAS E LINGUAGENS DE PROGRAMAÇÃO..................................29
2.1.1 Considerações Sobre Paradigmas de Desenvolvimento...............................29
2.1.2 Dificuldades da Programação Imperativa 30
2.1.2.1 Redundância 31
2.1.2.2 Acoplamento...32
2.1.2.3 Dificuldade de distribuição..32
2.1.3 Dificuldades da Programação Declarativa 33
2.1.4 Considerações Sobre Outras Abordagens de Programação..........................34
2.2 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)...................................35
2.2.1 Mecanismo de Notificações e Funcionamento 35
2.2.2 Resolução de Conflitos no PON 39
2.2.3 Breve Histórico das Implementações do PON...40
2.3 DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES (DON) 42
2.3.1 Perfil UML para o PON – *NOP Profile* 43
2.3.2 Processo de Desenvolvimento Orientado a Notificações 47
2.4 TESTE DE SOFTWARE 50
2.4.1 Origem, Objetivos e Terminologias Básicas do Teste de Software 50
2.4.2 Fases do Teste 52
2.4.3 Técnicas e Critérios de Teste 52
2.4.3.1 Técnica funcional...53
2.4.3.2 Técnica estrutural 57
2.4.3.3 Técnica baseada em defeitos...63
2.4.4 Principais Limitações da Atividade de Teste..67
2.4.5 Ferramentas de Teste..68
2.5 CONSIDERAÇÕES SOBRE A FUNDAMENTAÇÃO TEÓRICA 69
3 DESENVOLVIMENTO DE UMA APLICAÇÃO PON 70
3.1 APRESENTAÇÃO DO SOFTWARE DESENVOLVIDO....................................70
3.2 DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES (DON) 72
3.2.1 Especificação de Requisitos 74

3.2.2 Modelo de Casos de Uso....75
3.2.3 Modelo de Classes....77
3.2.4 Levantamento das Regras do Software....78
3.2.5 Modelo de Componentes....80
3.2.6 Modelo de Sequência....87
3.2.7 Modelo de Objetos....91
3.3 CONSIDERAÇÕES SOBRE O DESENVOLVIMENTO EM PON....93
4 PROPOSTA DE UM MÉTODO DE TESTE DE SOFTWARE PARA O PON....95
4.1 APRESENTAÇÃO DO MÉTODO PROPOSTO....95
4.2 TESTE UNITÁRIO EM PON....96
4.2.1 Visão Geral do Teste Unitário....97
4.2.2 Abordagens de Testes Unitários....99
4.2.2.1 Documento de plano de teste unitário....99
4.2.2.2 Estratégia de geração de casos de teste para Premisses....101
4.2.2.3 Estratégia de geração de casos de teste para Conditions e SubConditions105
4.2.2.4 Estratégia de geração de casos de teste para Rules....110
4.2.2.5 Estratégia de geração de casos de teste para FBEs....111
4.2.3 Avaliação dos Resultados dos Casos de Teste Unitários....120
4.2.4 Considerações Sobre o Teste Unitário proposto....121
4.3 TESTE DE INTEGRAÇÃO EM PON....122
4.3.1 Visão Geral do Teste de Integração....122
4.3.2 Estratégia de Teste com Caso de Uso....124
4.3.3 Teste de Integração em PON Utilizando Casos de Uso....125
4.3.3.1 Testes que exercitem as informações da documentação do caso de uso (funcionalidades)....126
4.3.3.2 Testes diretamente sobre as entidades que implementam o caso de uso.135
4.3.4 Avaliação dos Resultados dos Casos de Teste de Integração....148
4.3.5 Considerações Sobre o Teste de Integração Proposto....151
4.4 CONSIDERAÇÕES SOBRE O MÉTODO DE TESTE PROPOSTO....151
5 CASO DE ESTUDO: APLICAÇÃO DO MÉTODO DE TESTE DE SOFTWARE EM PON....153
5.1 PLANEJAMENTO E GERAÇÃO DE CASOS DE TESTE....153
5.1.1 Teste unitário....153
5.1.1.1 Testes unitários em Premisses....153
5.1.1.2 Testes unitários em SubConditions e Conditions....157
5.1.1.3 Testes unitários em Rules....159
5.1.1.4 Testes unitários em FBEs....160
5.1.2 Testes de Integração....163
5.1.2.1 Testes de integração no caso de uso Realizar Leitura de Teclado....163
5.1.2.2 Testes de integração no caso de uso Controlar Avião....166
5.1.2.3 Testes de integração no caso de uso Controlar Helicóptero....166
5.1.2.4 Testes de integração no caso de uso Controlar Apresentação....170

5.1.2.5 Testes de integração no caso de uso Pausar e Continuar Jogo 174
5.1.2.6 Testes de integração no caso de uso Parar Jogo 177
5.1.2.7 Testes de integração no caso de uso Detectar Colisão de Projétil.............180
5.1.2.8 Testes de integração no caso de uso Iniciar Jogo......................................185
5.2 CONSIDERAÇÕES SOBRE O CASO DE ESTUDO 186
6 CONCLUSÃO E TRABALHOS FUTUROS...187
6.1 CONCLUSÃO 187
6.2 TRABALHOS FUTUROS...190
REFERÊNCIAS...193
APÊNDICE A – DIAGRAMA DE CLASSES COMPLETO 203
APÊNDICE B – *RULES* DESENVOLVIDAS PARA O SOFTWARE 204
APÊNDICE C - DIAGRAMAS DE COMPONENTES 211
APÊNDICE D – DIAGRAMAS DE SEQUÊNCIA 236
APÊNDICE E – CÓDIGO FONTE EM FRAMEWORK PON OTIMIZADO DAS PREMISSES E RULES DO SOFTWARE DESENVOLVIDO..................................245
APÊNDICE F – PROBLEMAS IDENTIFICÁVEIS NO DESENVOLVIMENTO EM PON 252
APÊNDICE G – OUTRAS INFORMAÇÕES E TÉCNICA EXPERIMENTAL PARA O TESTE DE SOFTWARE EM PON 259

1 INTRODUÇÃO

Neste capítulo introdutório, a Seção 1.1 apresenta o contexto do trabalho e da pesquisa destacando os paradigmas de desenvolvimento de software e introduzindo o PON – Paradigma Orientado a Notificações. A Seção 1.2 apresenta o problema abordado e a necessidade de um método para teste de software para o PON. A Seção 1.3 apresenta os objetivos pretendidos com este trabalho. A Seção 1.4 apresenta a motivação e justificativa. A Seção 1.5 apresenta o método. Concluindo este capítulo, a Seção 1.6 apresenta a estrutura desta dissertação.

1.1 CONTEXTO DO TRABALHO E PESQUISA

Cada vez mais, a sociedade depende de software para praticamente todas as suas demandas. Estruturas governamentais e toda a diversidade de serviços públicos são controlados por sistemas computacionais. A indústria, o comércio de bens e serviços, o sistema financeiro e a própria área de cultura e entretenimento também utilizam software intensamente (SOMMERVILLE, 2011).

Em geral, os sistemas de software são artefatos complexos, pois eles reúnem, na forma de modelos e códigos, uma lógica de realização de processos ou funcionalidades não triviais. Um software de controle de estoque, por exemplo, deve incluir em sua lógica um grande elenco de funcionalidades para cobrir todas as necessidades que uma empresa possui para garantir controle sobre os materiais que mantêm armazenados. Isso pode levar a um conjunto de dezenas de milhares de linhas de código para traduzir em software a lógica e os processos envolvidos. Em decorrência desta complexidade, a atividade de desenvolvimento de software é, também, extremamente complexa, exigindo muito tempo e investimento para sua elaboração (SOMMERVILLE, 2011).

Nas últimas décadas, ocorreram grandes avanços no desenvolvimento de software. Um grande número de processos e ferramentas de desenvolvimento, abordagens de projeto, *frameworks,* técnicas e linguagens de programação foram e têm sido propostos pelas empresas, associações e academia (PRESSMAN, 2010).

Em termos de modelos de desenvolvimento de software, as linguagens de programação, em geral, podem ser agrupadas em alguns paradigmas fundamentais (também denominados modelos ou categorias): Paradigma Imperativo (PI) – que engloba o Paradigma Orientado a Objetos (POO) e o Paradigma Procedimental (PP) – Paradigma Declarativo (PD), Paradigma Funcional (PF) e Paradigma Lógico (PL) (SEBESTA, 2012).

Em uma linguagem de programação declarativa é necessário definir o que o programa deve fazer sem especificar como que isto ocorrerá. Existem dois principais estilos populares de programação declarativa (ROY; HARIDI, 2004): funcional e lógica. Um programa funcional depende da aplicação de instruções simples em que o resultado da computação (i.e., dados) é imediatamente utilizado. Assim, aborda a computação como uma avaliação de funções lógico-matemáticas e evita estados ou dados mutáveis. Por sua vez, a linguagem de programação lógica é um exemplo de linguagem que envolve a declaração de regras (ou linguagem baseada em regras) (SEBESTA, 2012). Neste paradigma, as regras não são especificadas em uma ordem particular e durante a execução do software, escolhe-se uma ordem em que as regras serão usadas para produzir um resultado desejado. Desta forma, o programa básico formará conclusões imediatas a partir de uma lista de premissas. Em contraste, em uma linguagem de programação imperativa é necessário descrever como a computação ocorrerá. O PI é fundamentado na lógica da arquitetura de Von Newman que consiste em sequências de instruções, que são armazenadas em memória e executadas ordenadamente. Durante a execução das instruções, dados podem ser acessados e escritos de/para posições específicas na memória do computador (BROOKSHEAR, 2011).

Em geral, cada linguagem de programação pode ser agrupada em um desses paradigmas. Como exemplo, a linguagem Fortran é considerada a primeira linguagem imperativa de alto nível. Posteriormente, a linguagem C forneceu importantes recursos como operadores aritméticos e de atribuição e estrutura de blocos. Isso permitiu a manipulação simplificada do ambiente, além de fornecer passagem de funções como parâmetros e ponteiros (GABBRIELLI; MARTINI, 2010). As linguagens Java, C++, .NET e Smalltalk são baseadas no modelo orientado a objetos. Haskell e Standard ML são baseados no modelo funcional. Prolog e Mercury são baseados no modelo lógico. Erlang é funcional, concorrente e mais tolerante a falhas em programação distribuída (ROY; HARIDI, 2004).

Apesar das evoluções na área de software, especificamente paradigmas e linguagens de programação, a utilização da capacidade plena de cada processador nem sempre é devidamente aproveitada em função de limitações das técnicas de programação usualmente empregadas (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009) (GABBRIELLI; MARTINI, 2010) (SIMÃO et al., 2012) (RONSZCKA, 2012). Na verdade, tecnologias de programação baseadas no estado da arte e da técnica, como o POO, sofrem de limitações intrínsecas de seus paradigmas (BANASZEWSKI, 2009) (SIMÃO et al., 2012).

Os paradigmas clássicos de desenvolvimento, como o PD e o PI comumente levam a um forte acoplamento de expressões causais e a processamento desnecessário por motivos como redundâncias em avaliações causais e utilização de custosas estruturas de dados para tal. Tais limitações frequentemente comprometem o desempenho pleno das aplicações. Neste âmbito, existem motivações para busca de alternativas ao PD e PI com o objetivo de eliminar ou diminuir as desvantagens desses paradigmas (ROY; HARIDI, 2004) (BANASZEWSKI, 2009) (SIMÃO et al., 2012) (GABBRIELLI; MARTINI, 2010).

Décadas de pesquisas apontam que a programação paralela não pode ser completamente ocultada do programador. Não é possível automaticamente transformar um programa qualquer em um programa paralelo empregando os paradigmas vigentes. As linguagens de programação tradicionais, como Java e C++, são fracamente equipadas para isto, justamente porque herdam limitações advindas de seus paradigmas como dificuldade implícita de alcançar desacoplamento entre módulos (ROY, 2009) (TANENBAUM, 2007).

Uma alternativa para resolver os problemas abordados é o emergente Paradigma Orientado a Notificações (PON). O PON surgiu como uma solução para desenvolvimento de software que unifica os melhores conceitos do Paradigma Declarativo (e.g. representação do conhecimento em regras) e do Paradigma Imperativo (e.g. flexibilidade de expressão e nível apropriado de abstração). Ao mesmo tempo, o PON objetiva resolver parte das principais deficiências do PD e PI no tocante ao cálculo lógico-causal.

Ao que tudo tem indicado, o PON resolveria, em termos de modelo, várias inconveniências em aplicações de software (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009) (SIMÃO et al., 2012). Neste sentido, várias pesquisas estão sendo feitas na área do PON como o desenvolvimento de uma linguagem nativa de

programação e respectivo compilador (RONSZCKA et al., 2013), arquitetura de hardware específica, coprocessador (PETERS, 2012), arquitetura paralela com distribuição de carga de software (i.e. *multicore*) (BELMONTE; SIMAO; STADZISZ, 2012), processador nativo em PON ARQPON (LINHARES; SIMÃO; STADZISZ, 2014) entre outros. Com isso, o PON está evoluindo em várias frentes a fim de se consolidar como um novo paradigma que forneça todo o aparato necessário para o desenvolvimento de software, desde a especificação, projeto, desenvolvimento, testes e manutenção.

1.2 PROBLEMA ABORDADO

O processo de desenvolvimento de software envolve intenso esforço na sua condução, com o estabelecimento de requisitos, projeto e modelagem, programação e teste. Em todas essas etapas do desenvolvimento de um software há a possibilidade de ocorrerem erros ainda que boas técnicas de desenvolvimento sejam aplicadas (DELAMARO; MALDONADO; JINO, 2007).

A atividade de teste, em especial, é muito importante, independentemente das facilidades de desenvolvimento do paradigma escolhido. Isso se deve à necessidade de se produzir software com qualidade, atendimento das expectativas do cliente, cumprimento de prazos e custos, entre outros. O teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe (MYERS et al., 2004), buscando reduzir a possibilidade dos erros ocorrerem futuramente no ambiente de produção ou, pior, quando entregue ao usuário final.

Na literatura, no tocante a outros paradigmas vigentes de desenvolvimento e/ou programação, vários trabalhos investigam o teste funcional (HOWDEN, 1980) (BEIZER, 1995), o teste estrutural (RAPPS; WEYUKER, 1985) (RAPPS; WEYUKER, 1982) (MALDONADO, 1991) (MCCABE, 1976) e o teste baseado em defeitos (AGRAWAL et al., 1989) (DEMILLO; LIPTON; SAYWARD, 1978) (OFFUTT, 1992) (OFFUTT; HAYES, 1996) (MA; KWON; OFFUTT, 2002). Também, constatou-se que a atividade de teste de software é dividida em fases, possui técnicas e critérios de teste, além de uma ampla variedade de ferramentas que auxiliam sua realização.

Como em qualquer outro paradigma de desenvolvimento de software, a atividade de teste também possui grande impacto em PON. Entretanto, no contexto do PON, esses e outros tipos de testes citados ainda não têm sido explorados suficientemente. Basicamente, os testes têm sido aplicados de maneira elementar e intuitiva, sem considerar as especificidades do PON, como sua organização em forma de regras.

Desta forma, o problema abordado nesta pesquisa é a falta de um método específico para a condução da atividade de teste dentro dos processos de desenvolvimento de software no paradigma PON. A solução deste problema contribuirá para aumentar a aplicabilidade do paradigma PON no desenvolvimento profissional de software.

Assim, este trabalho busca responder as seguintes perguntas:

- Como são os testes unitários em PON?
- Como são os testes de integração em PON?
- Como é o teste funcional e o teste estrutural em PON?
- É possível testar o PON por meio de modelos ou diagramas?

1.3 OBJETIVOS

Esta subseção apresenta o objetivo geral, bem como os objetivos específicos abordados nesta pesquisa.

1.3.1 Objetivo Geral

Esse trabalho de pesquisa se insere no contexto dos estudos feitos no CPGEI/UTFPR na área de Engenharia de Computação e, particularmente, na área de pesquisa do PON – Paradigma de Orientação a Notificações.

O objetivo geral do trabalho é propor um método de teste de software para o Paradigma Orientado a Notificações, visando contribuir para o estabelecimento de processos integrais de desenvolvimento de software PON.

1.3.2 Objetivos Específicos

A partir do objetivo geral desta pesquisa, foram definidos os seguintes objetivos específicos:

- Determinar quais são as técnicas de teste mais comuns e representativas na área de software;
- Adaptar os critérios de testes unitários padrões a critérios de teste unitário para software PON;
- Adaptar os critérios de testes de integração padrões a critérios de teste de integração para software PON;
- Avaliar a aplicação dos critérios de teste propostos em uma aplicação PON.

1.4 MOTIVAÇÃO E JUSTIFICATIVA

Desde sua proposição embrionária a partir dos trabalhos de Simão (2001), o PON evoluiu em diversas frentes, incluindo sua compreensão como paradigma, o desenvolvimento de materializações como *framework*s (em três versões), linguagem, compilador e hardwares, o Desenvolvimento Orientado a Notificações (DON) (WIECHETECK, 2011), comparações de desempenho entre PON e POO (BANASZEWSKI, 2009), padrões de projeto que foram incorporados ao *framework*[1] e utilizados na concepção de software PON (RONSZCKA, 2012), entre outros. Entretanto, nenhum destes trabalhos enfatizou a atividade de teste das aplicações produzidas. Em geral, as verificações das aplicações se limitaram a ensaios de execução e depuração de erros lógicos evidentes.

No trabalho de pesquisa descrito nesta dissertação, busca-se estudar como conduzir a tarefa de teste dentro de um processo de desenvolvimento de software PON. A motivação para esta pesquisa é contribuir para o desenvolvimento de processos de desenvolvimento de software PON por meio da proposição de um método de teste apropriado ao paradigma. Como consequência, este método

[1] Atualmente, o estado da técnica do PON é um *framework* (denominado *Framework* PON Otimizado) em C++, enquanto uma linguagem e um compilador especializados estão em desenvolvimento (RONSZCKA, 2012) (VALENÇA, 2012).

permitirá aumentar os níveis de confiança na qualidade dos softwares desenvolvidos em PON.

1.5 MÉTODO

Para realização desta pesquisa foram realizados diversos trabalhos que podem ser agrupados nas seguintes etapas: (i) Levantamento bibliográfico, (ii) Projeto e construção de um software PON, (iii) Estudo das técnicas de teste funcional, estrutural e baseada em defeitos, (iv) Estudo e proposição de um método de teste de software para o PON, (v) Realização de um caso de estudo aplicando o método de teste apresentado e (vi) Elaboração dessa dissertação e artigo sobre a pesquisa. Detalha-se a seguir estas etapas:

(i) Levantamento bibliográfico

Para realização do levantamento bibliográfico inicial, foram pesquisados trabalhos sobre o PON usando a palavra-chave "notification oriented paradigm" nos repositórios de Periódicos CAPES (busca retornou sete resultados), IEEE Xplore (busca retornou dois resultados) e Google Scholar (busca retornou 32 resultados).

Sobre teste, não se utilizou busca sistemática, pois não havia interesse em descobrir quais são as pesquisas na área envolvendo novos conceitos, técnicas ou critério. Buscou-se uma literatura clássica nos livros a respeito de métodos convencionais e padrões de teste. Foram utilizados 10 livros, dentre as referências clássicas incluindo: Introdução ao Teste de Software (DELAMARO; MALDONADO; JINO, 2007), Testing Object-oriented Systems: Models, Patterns, and Tools (BINDER, 1999), Art of Software Testing (MYERS et al., 2004), Black-box testing: techniques for functional testing of software and systems (BEIZER, 1995). Também, foram estudados trabalhos bastante conhecidos na area de testes como Data Flow Analysis Techniques for Test Data Selection (RAPPS; WEYUKER, 1982), An Applicable Family of Data Flow Testing Criteria (FRANKL; WEYUKER, 1988), Selecting Software Test Data Using Data Flow Information (RAPPS; WEYUKER, 1985), Functional Program Testing (HOWDEN, 1980), *Design* of Mutant Operators for the C Programming Language (AGRAWAL et al., 1989), Hints on Test Data Selection: Help for the Practicing Programmer (DEMILLO; LIPTON; SAYWARD,

1978), A Semantic Model of Program Faults (OFFUTT; HAYES, 1996), entre outros, além da tese Critérios potenciais usos: Uma contribuição ao teste estrutural de software (MALDONADO, 1991).

(ii) Projeto e construção de um software PON

Nesta etapa se buscou adquirir conhecimento sobre os aspectos conceituais e práticos do PON, por meio do desenvolvimento de um software com dimensão até maior que outros estudos anteriores. Esse software é um jogo de combate aéreo em duas dimensões que foi modelado com o DON e implementado com o (mais recente) *framework* otimizado PON.

(iii) Estudo das técnicas de teste funcional, estrutural e baseado em defeitos

Nesta etapa, foram estudados os principais trabalhos sobre o teste funcional, o teste estrutural e o teste baseado em defeitos com o objetivo de fazer um balanço das técnicas convencionais de teste que poderiam ser aplicadas no desenvolvimento de software PON.

(iv) Estudo e proposição de um método de teste de software para o PON

Esta etapa considerou o conhecimento sobre as técnicas de teste de software mais conhecidas (funcional, estrutural e baseada em erros) para propor um método de teste que consiste na aplicação dessas técnicas e adaptação de critérios de teste para o PON.

(v) Realização de um caso de estudo aplicando o método de teste apresentado

Nesta etapa, o software desenvolvido foi testado com todas os critérios de teste propostos na Seção (iv).

(vi) Elaboração da dissertação e artigo sobre a pesquisa.

A última etapa foi o registro do conhecimento produzido na forma da dissertação de mestrado que compõe este documento. Foi submetido e aceito um artigo no COMTEL 2014, titulado “Introdução ao teste funcional de software no Paradigma Orientado a Notificações” (KOSSOSKI; SIMÃO; STADZISZ, 2014).

1.6 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta a Fundamentação Teórica estudada para elaboração desta dissertação. O Capítulo 3 apresenta o Desenvolvimento de uma Aplicação

PON utilizando os atuais estados da arte e da técnica desse paradigma. O Capítulo 4 apresenta a Proposta de um Método de Teste de Software para o PON. O Capítulo 5 apresenta um Caso de Estudo que consiste na aplicação do método de teste proposto no Capítulo 4. Finalizando esta dissertação, o Capítulo 6 apresenta a Conclusão e os Trabalhos Futuros que podem ser desenvolvidos a partir desse trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos que fundamentaram a elaboração dessa pesquisa de mestrado. A Seção 2.1 apresenta uma visão geral sobre o atual estado da arte dos paradigmas e linguagens de programação usuais na computação. A Seção 2.2 apresenta uma visão do estado da arte e da técnica do PON. A Seção 2.3 apresenta, brevemente, o Desenvolvimento Orientado a Notificações (DON). A Seção 2.4 apresenta uma visão geral sobre as principais fases e técnicas de teste de software existentes. Concluindo este capítulo, a Seção 2.5 apresenta as considerações sobre a fundamentação teórica.

Salienta-se que a Seções 2.1, 2.2 e 2.3 baseiam-se inclusive na adaptação e mesmo resumo de textos do grupo do PON, destacando aqui os trabalhos de Banaszewski (2009), Ronszcka (2012), Valença (2012), Wiecheteck (2011) Xavier (2014), Linhares (2014) entre outros. Detalhamentos sobre paradigmas e linguagens de programação podem ser encontrados em trabalhos dos grupos de Banaszewski, Ronszcka, Xavier e também na literatura pertinente como Roy (2004), Scott (2009), Gabbrielli e Martini (2010) e Sebesta (2012).

2.1 PARADIGMAS E LINGUAGENS DE PROGRAMAÇÃO

Uma vez que esta pesquisa está focada no Paradigma Orientado a Notificações (PON), esta seção faz uma revisão geral dos paradigmas de desenvolvimento de software de forma a facilitar a compreensão das características do PON, discutidas a seguir.

2.1.1 Considerações Sobre Paradigmas de Desenvolvimento

A maioria das linguagens de programação mais utilizadas pertence ao Paradigma Imperativo (PI) (SEBESTA, 2012). O PI, em geral, funciona sobre pesquisas orientadas a laços de repetições sobre elementos passivos (dados e comandos de decisão), relacionando os dados às expressões causais.

Normalmente, estas características impactam negativamente nas aplicações, devido a sua estrutura monolítica, prolixa e acoplada que gera execução de código não otimizado, redundante e interdependente (SIMÃO et al., 2012). Esse paradigma exige que o desenvolvedor descreva “como fazer” todos os passos que o programa deve seguir para processar determinado resultado, isto é, obrigando o programador a ter total controle sobre a aplicação (BOOKSHEAR, 2006) (BANASZEWSKI et al., 2007) (SIMÃO et al., 2012) (GABBRIELLI; MARTINI, 2010) (RONSZCKA, 2012).

O Paradigma Declarativo (PD) é uma alternativa ao PI, proporciona um nível maior de abstração e mais facilidades de programação. Um exemplo de tal facilidade é permitir que o desenvolvedor descreva “o que fazer”, para que o software processe determinado resultado, isto é, o programador não precisa se preocupar em descrever minunciosamente como o programa deve se comportar (KAISLER, 2005) (GABBRIELLI; MARTINI, 2010). Também, algumas linguagens declarativas podem evitar muitas das redundâncias de execução, como os Sistemas Baseados em Regras (SBR) baseados em algoritmos de inferência otimizados (FORGY, 1982) (CHENG; CHEN, 2000) (LEE; CHENG, 2002) (KANG; CHENG, 2004).

No entanto, programas construídos em linguagens de programação comuns do PI ou do PD, mesmo que baseadas em soluções otimizadas, também apresentam problemas (BANASZEWSKI et al., 2007) (SIMÃO; STADZISZ, 2008) (SIMÃO et al., 2012) (RONSZCKA, 2012). Deste modo, as subseções a seguir, irão apresentar um pouco mais sobre as principais limitações desses paradigmas.

2.1.2 Dificuldades da Programação Imperativa

As principais dificuldades ou desvantagens da programação imperativa são redundância de código, acoplamento e distribuição em multiprocessadores (ou *multicore* em idioma inglês) (SIMÃO; STADZISZ, 2009). A primeira afeta principalmente o tempo de processamento, a segunda, o processo de desacoplamento (reaproveitamento de módulos e/ou partes) e a terceira afeta a facilidade em distribuir o código em múltiplos processadores (SIMÃO et al., 2012).

2.1.2.1 *Redundância*

Na programação imperativa, incluindo a programação orientada a objetos, a presença de código redundante é resultado da maneira com que as expressões causais são organizadas e avaliadas (SIMÃO; STADZISZ, 2008) (SIMÃO et al., 2012). No exemplo do Algoritmo 1 é possível observar que o laço de repetição força a avaliação de todas as condições de maneira sequencial. Entretanto, a maioria das avaliações é desnecessária quando apenas alguns atributos podem apresentar mudança de valor em cada iteração. Se for considerado um sistema maior, integrando muitas partes como essa, pode-se ter uma grande diferença de desempenho como um todo. Ainda, este tipo de código apresenta problemas denominados "redundância temporal e redundância estrutural" (PAN; SOUZA; KAK, 1998) (SIMÃO; STADZISZ, 2008) (SIMÃO; STADZISZ, 2009) (SIMÃO et al., 2012).

```
1    . . .
2    while (true) do
3      if ((object_1.Attribute_1 = 1) and
4          (object_2.Attribute_2 = 1) and
5          (object_3.Attribute_3 = 1))
6      then
7           object_1.Method_1();
8           object_2.Method_1();
9           object_3.Method_1();
10     end_if
11     . . .
12     if ((object_1.Attribute_1 = 1) and
13         (object_2.Attribute_n = n) and
14         (object_3.Attribute_n = n))
15     then
16          object_1.Method_n();
17          object_2.Method_n();
18          object_3.Method_n();
19     end_if
20   end_while
21   . . .
```

Algoritmo 1 – Exemplo de código redundante na Programação Imperativa
Fonte: Simão et al. (2012)

A redundância temporal ocorre quando uma avaliação lógico-causal é realizada repetidas vezes sobre um elemento já avaliado e inalterado. A redundância estrutural, por sua vez, ocorre quando uma expressão lógica não é compartilhada entre outras expressões causais relacionadas, causando reavaliações

desnecessárias (SIMÃO; STADZISZ, 2008) (SIMÃO; STADZISZ, 2009) (BANASZEWSKI, 2009).

2.1.2.2 Acoplamento

O acoplamento entre códigos, classes ou subsistemas refere-se ao número de interconexões e/ou dependências com diversas outras partes do código. O acoplamento fraco significa que as classes relacionadas contém poucas ou apenas uma conexão com outras classes. Por sua vez, o acoplamento forte significa que as classes relacionadas precisam conhecer detalhes internos umas das outras, as alterações se propagam pelo sistema tornando potencialmente mais difícil a atividade de desenvolvimento, manutenção e teste (GAMMA et al., 1995).

O acoplamento aumenta também a complexidade na escrita de instruções, dificultando o reaproveitamento de código, distribuição e paralelização em sistemas distribuídos (SIMÃO; STADZISZ, 2009) (SIMÃO et al., 2012).

A programação na linguagem imperativa naturalmente causa acoplamento durante o desenvolvimento do software, pois mistura relações causais com dados (LINHARES; SIMÃO; STADZISZ, 2014). As avaliações de expressões causais são realizadas sequencialmente em um programa ou, pelo menos, em linhas de execução presentes em *threads*, normalmente guiadas por meio de um laço de repetição. Essas expressões causais não conduzem ativamente sua própria execução, sendo, portanto, passivos e interdependentes (SIMÃO; STADZISZ, 2009) (SIMÃO et al., 2012).

2.1.2.3 Dificuldade de distribuição

A dificuldade de distribuição é um problema, uma vez que existem situações em que a distribuição é realmente necessária como softwares para meteorologia (MARETIS, 1990), sistema de controle de planta nuclear (DÍAZ et al., 2007), sistema de manufatura inteligente (DEEN, 2003) (SIMÃO, 2005) (TIANFIELD, 2007) e controle cooperativo (KUMAR; LEONARD; MORSE, 2004) (SIMÃO et al., 2012) (RONSZCKA, 2012).

A execução concorrente (i.e. distribuída) é difícil de ser desenvolvida e aplicada (TANENBAUM, 2007). Na linguagem imperativa, o programador necessita fazer uma divisão estrutural do programa em partes concorrentes, que são escritas como tarefas, e a execução delas deve ser sincronizada (ROY, 2009). Por isso, linguagens de programação tradicionais como Java ou C++ estão mal equipadas para funcionar com *multicore* (ROY, 2009) (RONSZCKA, 2012).

2.1.3 Dificuldades da Programação Declarativa

As principais limitações ou desvantagens da programação declarativa estão relacionadas às pesadas estruturas de dados computacionais e também ao acoplamento (SIMÃO et al., 2012).

Na programação declarativa, os estados das variáveis são tratados em uma Base de Fatos e o conhecimento causal em uma Base Causal (ou Base de Regras, na programação em SBR), que são automaticamente combinadas por meio de um Motor de Inferência. Além disso, alguns algoritmos de inferência (i.e. RETE, TREAT, LEAPS e HAL) evitam a maioria das redundâncias temporais e estruturais (BANASZEWSKI, 2009). No entanto, as estruturas de dados utilizadas para resolver esses problemas ainda implicam em muito consumo de capacidade de processamento (FORGY, 1982) (SIMÃO et al., 2012) (RONSZCKA, 2012).

Além da sobrecarga de processamento, a programação declarativa também apresenta acoplamento em seu código, de maneira similar à programação imperativa. O motor de inferência é responsável por analisar todos os dados passivos (base de fatos) e inferir, a partir do estado desses, as expressões causais (regras) afetadas por tais estados (SIMÃO; STADZISZ, 2009) (SIMÃO et al., 2012) (RONSZCKA, 2012).

O uso da programação declarativa é mais motivador quando o software em desenvolvimento apresenta numerosas redundâncias e pouca variação de dados. Além disso, em geral, um motor de inferência relacionado a uma determinada linguagem declarativa limita a criatividade do desenvolvedor, dificultando algumas otimizações algorítmicas e o acesso ao hardware, o que pode ser inadequado em determinados contextos (SCOTT, 2000) (WATT, 2004) (BOOKSHEAR, 2006) (SIMÃO et al., 2012).

2.1.4 Considerações Sobre Outras Abordagens de Programação

Foram propostas diversas melhorias no contexto da programação imperativa e da programação declarativa, tais como a Programação Orientada a Eventos (POE) e a Programação Funcional (PF) (RUSSEL; NORVIG, 2003) (FAISON, 2006) (BANASZEWSKI, 2009). Essencialmente, na POE e na PF, cada evento (e.g. um botão pressionado, uma interrupção de hardware ou uma mensagem recebida) desencadeia uma dada execução (i.e. procedimento, processo ou método), geralmente em um tipo determinado de módulo (i.e. bloco, objeto ou até mesmo agente), ao invés de análises sucessivas e repetidas das expressões relacionais para a sua execução. No entanto, a PF diferencia-se da POE porque as chamadas de funções ocorrem por meio de outras funções, em substituição aos eventos (SIMÃO et al., 2012) (RONSZCKA, 2012). Tanto a POE quando a PF são fundamentadas no PI e PD, herdando, portanto desvantagens desses paradigmas ainda que diminuam os efeitos da redundância temporal.

Outra abordagem alternativa é a Programação Orientada a Fluxo de Dados (JOHNSTON; HANNA; MILLAR, 2004), que permite a execução do programa orientada por dados, ao invés de uma linha de execução com base na pesquisa sobre os dados. Portanto, isso facilitaria o desacoplamento e a distribuição (JOHNSTON; HANNA; MILLAR, 2004) (SIMÃO et al., 2012) (RONSZCKA, 2012). Porém, o processamento lógico causal é realizado por intermédio de avançados motores de inferência, tais como RETE (GAUDIOT; SOHN, 1990) (TUTTLE; EICK, 1992) (SIMÃO et al., 2012). Conforme discutido previamente, os motores de inferência atuais tentam alcançar uma abordagem orientada a fluxo de dados. No entanto, o processo de inferência ainda se baseia em pesquisas sobre elementos passivos (SIMÃO et al., 2012) (RONSZCKA, 2012).

Como pode ser observado, existem vários problemas que dificultam o desenvolvimento de software em relação à composição de código otimizado, distribuído e simplificado na escrita de instruções, o que são questões relacionadas ao uso mais racional dos recursos computacionais. Nesse contexto, um novo paradigma de programação, chamado Paradigma Orientado a Notificações (PON), foi proposto para resolver parte, ou ao menos minimizar, conforme o ponto de vista, alguns desses problemas destacados.

2.2 PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

O Paradigma Orientado a Notificações (PON) surgiu como uma solução que usa os melhores conceitos do Paradigma Imperativo e Paradigma Declarativo, ao mesmo tempo em que objetiva resolver suas deficiências no tocante ao cálculo lógico-causal. O PON propõe que toda a inferência ocorra por meio de entidades notificantes mínimas e colaborativas que tratam de processamento factual e lógico-causal, evitando a má utilização de processamento e problemas de acoplamento dos paradigmas mais populares, ainda buscando simplificar a tarefa de escrita de código (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009) (RONSZCKA, 2012).

Com isso, o PON apresentaria resposta a vários problemas desses paradigmas, como repetição de expressões lógicas e reavaliações desnecessárias (i.e. redundâncias temporais e estruturais) e, particularmente, o acoplamento forte de entidades com relação às avaliações ou cálculo lógico-causal (SIMÃO et al., 2012).

Atualmente, o estado da técnica do PON, no tocante o desenvolvimento, é um *framework* em C++, enquanto uma linguagem, um compilador e, mesmo, um microprocessador especializado estão em franco desenvolvimento prototipal (XAVIER, 2014) (FERREIRA, 2014) (LINHARES; SIMÃO; STADZISZ, 2014). O *Framework PON* foi projetado para fornecer uma *Application Programming Interface* (*API*) e estruturas que facilitam o desenvolvimento de software segundo a orientação do PON (RONSZCKA, 2012) (VALENÇA, 2012).

2.2.1 Mecanismo de Notificações e Funcionamento

Uma aplicação desenvolvida em PON é constituída por um conjunto de Regras (*Rules*) e Entidades Factuais (*FBE – Fact Base Element*). O fluxo de execução é construído de maneira implícita às declarações das *Rules* e *FBEs*, graças à maneira emergente que são construídas as conexões para execução da cadeia de notificações pontuais entre as entidades PON. Desta forma, a definição da inferência em PON ocorre de maneira diferente do fluxo de execução encontrado em aplicações do PI, como o POO, nas quais o desenvolvedor informa de maneira

explícita a sequência ou laço de iteração por meio de comandos como *while* e *for* (SIMÃO et al., 2012).

A dinâmica da execução da aplicação PON se dá pela mudança de estado dos *FBEs* que provocam um ciclo de notificações até as *Rules* por meio de condições que levam o sistema a reagir e alcançar novos estados. A Figura 1 esboça o diagrama de classes conceitual do funcionamento do PON.

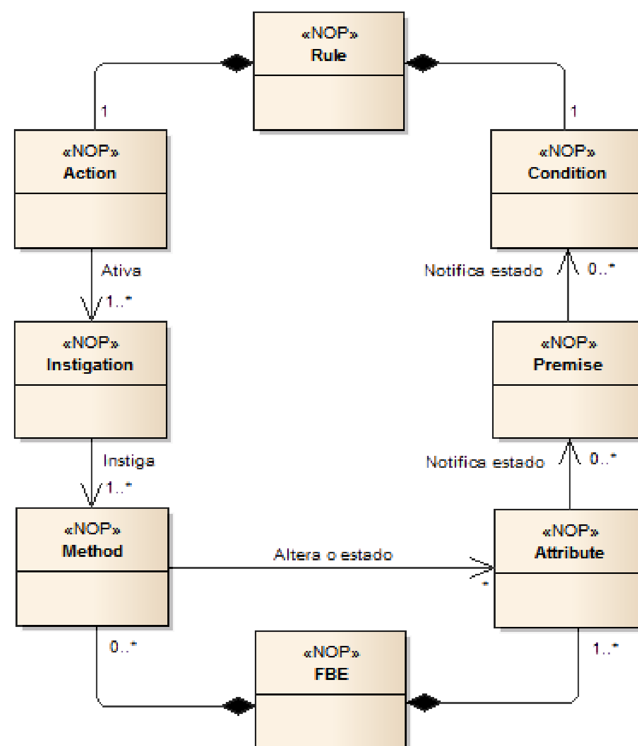


Figura 1 – Diagrama de classes conceitual do PON e suas entidades
Fonte: Simão, Tacla et al. (2012)

A classe *FBE* representa entidades do sistema que juntas representam seu estado e parte de comportamento do sistema, respectivamente por meio de seu conjunto de *Attributes* e seu conjunto de *Methods*. Quando há uma mudança de estado no sistema, um objeto *Attribute* de um determinado *FBE* sofrerá mudança de seu valor (estado). Neste momento ele notifica as *Premisses* especificamente a ele relacionadas, para que estas reavaliem seus estados lógicos. Se o valor lógico da *Premise* é alterado, a *Premise* colabora com a avaliação lógica de uma ou de um conjunto de *Conditions* relacionadas, o que ocorre por meio da notificação sobre a mudança de seu estado lógico às *Conditions* (BANASZEWSKI, 2009).

Na sequência, cada *Condition* notificada avalia o seu valor lógico de acordo com as notificações da *Premise* e com o operador lógico (de conjunção ou

disjunção) utilizado. Então, no caso de uma conjunção, quando todas as *Premisses* que integram uma *Condition* são satisfeitas, a *Condition* também é satisfeita, resultando na aprovação da sua respectiva *Rule* para a execução (BANASZEWSKI, 2009) (RONSZCKA, 2012).

Quando uma *Rule* aprovada está pronta para executar, a sua *Action* é ativada. Uma *Action* é conectada a uma ou várias *Instigations*. As *Instigations* notificadas pelas *Actions* acionam a execução de algum serviço de um objeto *FBE* por meio dos *Methods*. Normalmente, as chamadas para os *Methods* também alteram os estados dos *Attributes* e o ciclo de notificação pode recomeçar (BANASZEWSKI, 2009).

Por exemplo, a Figura 2 ilustra uma aplicação PON contendo dois *FBEs* (à esquerda) e duas *Rules* (à direita). Os dois *FBEs* contém *Attributes* que, quando têm seus valores alterados, notificam as respectivas *Premisses* e, consecutivamente, as *Conditions* das *Rules*. Quando ativadas, as *Actions* das *Rules* notificam às *Instigations* que chamam a execução dos *Methods* dos *FBEs* (SIMÃO et al., 2012). Esse exemplo permite melhor compreender como se dá o ciclo de notificações.

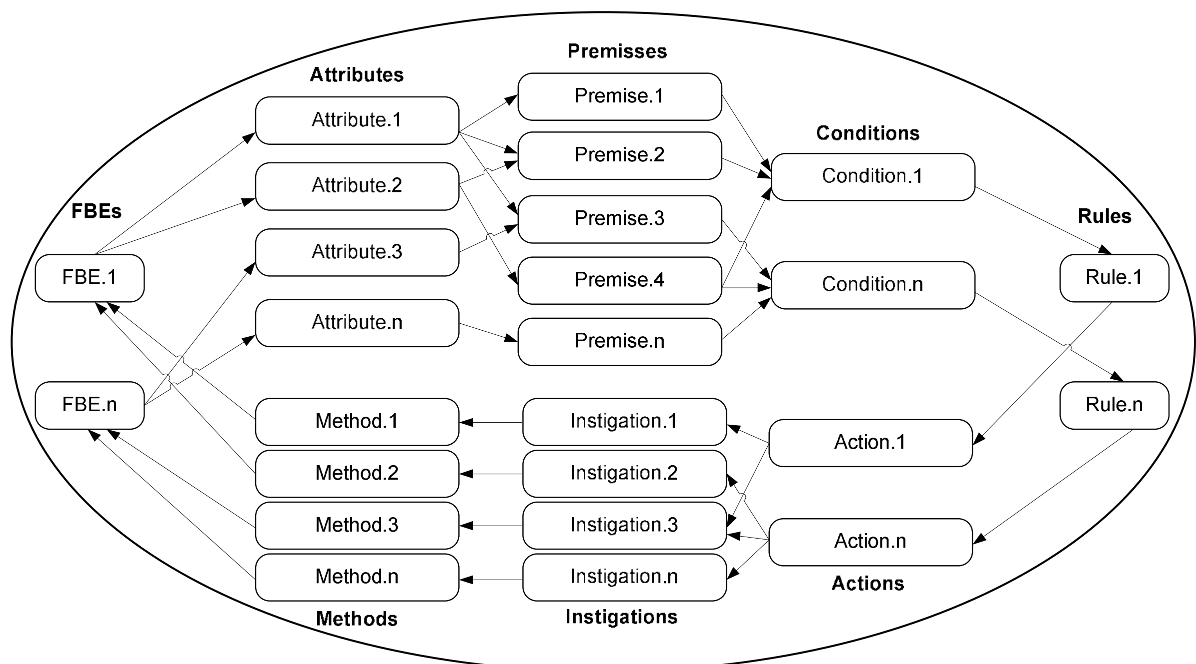


Figura 2 – Cadeia de notificação das entidades do PON
Fonte: Adaptado de Simão, Tacla et al. (2012)

A Figura 3 apresenta um exemplo de *Rule*, na forma de uma regra causal. Cada *Rule* é uma entidade computacional composta por outras entidades (Figura 2),

que podem ser observadas como objetos e/ou classes. Por exemplo, essa *Rule* apresentada é composta por um objeto *Condition* e um objeto *Action*. A *Condition* trata da decisão da *Rule*, enquanto a *Action* trata da execução das ações da *Rule*. Assim, *Condition* e *Action* trabalham juntas para realizar o conhecimento lógico e causal da *Rule* (SIMÃO et al., 2012).

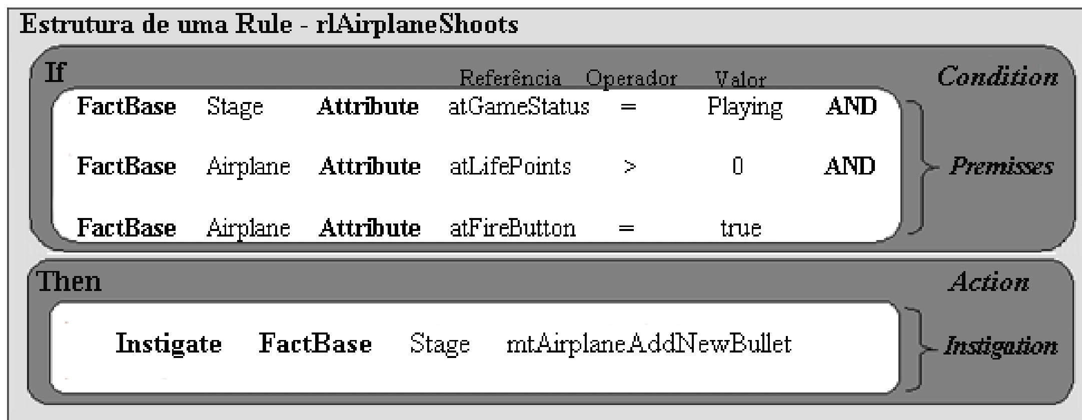


Figura 3 – Exemplo de uma *Rule* e entidades relacionadas

A *Rule* apresentada na Figura 3 controla o disparo de um projétil de um avião (v.g. jogador) em um jogo de combate aéreo desenvolvido em 2D para esta dissertação (será apresentado no Capítulo 3). A *Condition* dessa *Rule* contém uma conjunção que requer três *Premisses* verdadeiras. As *Premisses* fazem as seguintes avaliações sobre o Attributes das *FBEs*: a) Stage está em estado “jogando” (Playing)? b) o avião (Airplane) possui pontos de vida (atLifePoints) para que possa disparar? c) foi pressionado o botão (atFireButton) de disparo? Quando a *Condition* estiver satisfeita, a *Rule* executará a *Instigation*, que por sua vez, neste caso, executa um *Method* para adicionar um novo projétil na tela do jogo.

Assim, é possível perceber que, na essência, a computação no PON está organizada e distribuída entre entidades autônomas e reativas que colaboram por meio de notificações pontuais. Desta maneira, o PON leva a uma nova maneira de desenvolver software, na qual os fluxos de execução ocorrem de maneira colaborativa entre as entidades (RONSZCKA et al., 2011).

2.2.2 Resolução de Conflitos no PON

Um conflito ocorre quando duas ou mais *Rules* referenciam um mesmo *FBE* e demandam exclusividade de acesso a este *FBE*. Deste modo, as *Rules* concorrem para adquirir acesso exclusivo a esse *FBE*, sendo que somente uma delas pode executar por vez, ou seja, aquela que obteve o acesso exclusivo. Assim, para definir as questões de resolução de conflitos entre as *Rules*, basicamente, o fluxo de sua execução é determinado segundo uma estratégia pré-estabelecida. Essa estratégia pode variar para alcançar o fluxo de execução pretendido pelo desenvolvedor e também pode variar em função do ambiente monoprocessado ou multiprocessado (RONSZCKA, 2012).

Em ambientes monoprocessados é empregado um escalonador de *Rules* formado por uma estrutura de dados do tipo linear (e.g. pilha, fila ou lista) (BANASZEWSKI, 2009). Essas estruturas guardam referências para as *Rules* aprovadas, conforme ilustra a Figura 4. Assim, elas recebem as *Rules* na ordem em que elas são aprovadas, podendo reorganizá-las de acordo com cada estratégia adotada (BANASZEWSKI, 2009) (RONSZCKA, 2012).

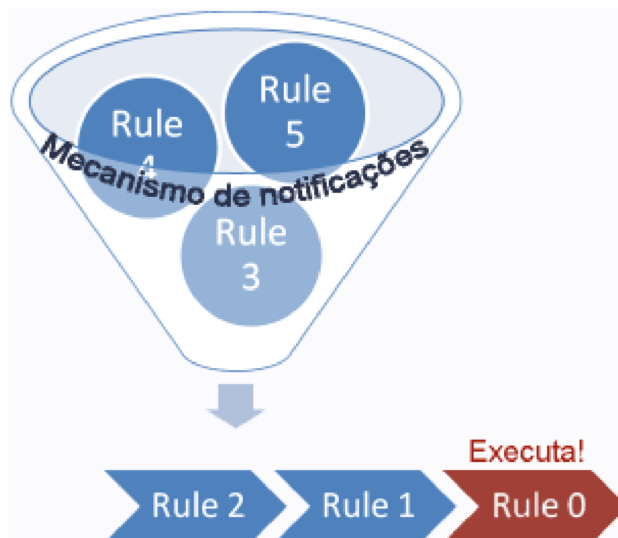


Figura 4 – Modelo centralizado de resolução de conflitos
Fonte: Adaptado de Banaszewski (2009)

Desta forma, conforme a estratégia de resolução de conflitos predeterminada pelo desenvolvedor as *Rules* em questão serão efetivamente executadas. Neste âmbito, os modelos de resolução de conflitos empregados para o

Framework PON Otimizado em ambientes monoprocessados são (RONSZCKA, 2012):

- *BREADTH*: se baseia no escalonamento *F*irst *I*n, *F*irst *O*ut (*FIFO*), ou seja, refere-se à execução de entidades *Rule* seguindo uma estrutura de dados do tipo fila;
- *DEPTH*: se baseia no escalonamento *L*ast *I*n, *F*irst *O*ut (*LIFO*), ou seja, refere-se à execução de entidades *Rule* seguindo uma estrutura de dados do tipo pilha;
- *PRIORITY*: organiza as entidades *Rule* de acordo com as prioridades definidas nas mesmas; e
- *NO_ONE*: quando nenhuma estratégia for definida pelo desenvolvedor, esta estratégia não envia as *Rules* ao escalonador e permite que as mesmas sejam aprovadas e executadas imediatamente.

As soluções para evitar os conflitos apresentados são particularmente aplicáveis a soluções PON monoprocessadas, ainda que até possam ser úteis em soluções multiprocessadas e distribuídas, mas se caracterizando como elemento acoplante (BANASZEWSKI, 2009).

Nos trabalhos de Simão (2005), Banaszewski (2009), Simão e Stadzisz (2008), Simão e Stadzisz (2009) e Simão e Stadzisz (2010) encontram-se soluções úteis ao PON para resolução de conflitos em aplicações distribuídas, bem como soluções correlatas para a garantia do chamado determinismo de evolução nessa classe de aplicações (RONSZCKA, 2012). Entretanto, sistemas distribuídos para o PON não são particularmente tratados nesta dissertação.

2.2.3 Breve Histórico das Implementações do PON

Desde a materialização original (metamodelo monoprocessado) proposto como *framework* em linguagem de programação C++ (SIMÃO, 2005), o PON recebeu várias melhorias na estrutura e comportamento visando se estabelecer como um paradigma de programação efetivo e de fácil aplicação.

A primeira implementação do PON, denominada *Framework* Original, foi comparada em tempos de execução com implementações PI/POO e PD/SBR. No caso de PI/POO houve comparações com programas C++/OO usuais

(BANASZEWSKI, 2009) (SIMÃO et al., 2012). No caso de PD/SBR, houve comparações com dois *shells*, CLIPS e *RuleWorks*, que usam o eficiente motor de inferência RETE. Essas comparações apresentaram resultados a favor do PON, ainda que sobre *toy problems* (BANASZEWSKI, 2009). Também, houve comparações qualitativas e assintóticas favoráveis ao PON em relação a motores de inferência como RETE, TREAT, LEAPS e HAL (BANASZEWSKI, 2009) (XAVIER, 2014).

Outros testes sobre aplicações reais se fizeram necessários para verificar a eficiência e eficácia do PON, em termos de desempenho, particularmente em relação ao dominante POO. Tais testes mostraram que a implementação do PON é melhor que POO quando há muitas relações causais e variáveis com baixa ou média variação de estados (BANASZEWSKI, 2009) (LINHARES et al., 2011) (BATISTA et al., 2011) (RONSZCKA et al., 2011) (VALENÇA et al., 2011) (SIMÃO et al., 2012) (SIMÃO et al., 2012) (SIMÃO et al., 2012).

A comparação entre o *Framework* PON Prototipal e o *Framework* Original feito por Banaszewski (2009) concluiu que quanto melhor for a implementação do PON (e.g. em termos de otimização de estruturas de dados), melhores serão os resultados de desempenho. Similarmente, Ronszcka (2012) e Valença (2012) compararam o *Framework* Original e a nova versão denominada *Framework* PON Otimizado obtendo resultado favorável para o *Framework* Otimizado (VALENÇA et al., 2011) (SIMÃO et al., 2012) (SIMÃO et al., 2012) (XAVIER, 2014).

No estado da técnica, e mesmo da arte em PON, há também pesquisas em desenvolvimento como linguagem nativa de programação e respectivo compilador (RONSZCKA et al., 2013), arquitetura de hardware específica, coprocessador PON (PETERS, 2012), arquitetura paralela com distribuição de carga de software (i.e. *multicore*) (BELMONTE; SIMAO; STADZISZ, 2012) e o processador nativo em PON ARQPON (LINHARES; SIMÃO; STADZISZ, 2014). Ainda, acerca da máquina de inferência que realiza o cálculo lógico-causal do PON, há estudos utilizando Fuzzy e Redes Neurais (MELO, 2013). Em termos de processo de engenharia de software, o estado da arte específico para o PON é o DON – Desenvolvimento Orientado a Notificações e seu inerente Perfil para UML (WIECHETECK, 2011) (WIECHETECK; STADZISZ; SIMÃO, 2011) (XAVIER, 2014). O DON, que será apresentado brevemente na próxima seção, inclusive foi empregado para a elaboração do Capítulo 3 desta dissertação.

O perfil UML denominado Perfil PON define os principais conceitos do PON por meio da utilização de mecanismos de extensão da UML (e.g. estereótipos) (WIECHETECK; STADZISZ; SIMÃO, 2011). O DON é organizado em oito passos: “Capturar requisitos”, “Criar Modelo de Caso de Uso”; “Criar Modelo de Classes”, “Criar Modelo de Atividades de Alto Nível”, “Criar Modelo de Componentes”, “Criar Modelo de Sequência”, “Criar Modelo de Comunicação”, “Criar Modelo de Redes de Petri”. Os dois primeiros passos fazem parte da fase de requisitos, enquanto os outros seis passos representam a fase de projeto de software (WIECHETECK, 2011) (XAVIER, 2014).

Ainda, o trabalho de Xavier (2014) apresenta uma comparação entre Paradigma Orientado a Eventos (POE) e o PON. Como conceito, ambos podem ter inspirações semelhantes para seu surgimento, como facilitar construção de software por meio de estruturas reativas. No entanto, eles são díspares em suas características, como na forma de resolver problemas ao construir software. Portanto, um dos principais objetivos daquele estudo foi comparar os paradigmas POE e PON segundo os critérios de suas características estruturais, forma de construir software e medições durante a execução, buscando detectar as vantagens e desvantagens relativas de cada paradigma. Ao construir software em ambos os paradigmas, ficou evidente que o PON tem maior expressividade em programação. Propôs-se que o PON é uma alternativa para tratamento de eventos em software, sendo oportunamente integrável, por exemplo, entre *Framework* C++ e OO C++ (XAVIER, 2014). Também, o trabalho de Linhares (2014) propôs uma arquitetura de computação (processador) mais adequada à execução de programas PON do que o modelo de Von Neumann – o ARQPON.

2.3 DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES (DON)

Em termos de processo de engenharia de software, o estado da arte específico para o PON é o DON – Desenvolvimento Orientado a Notificações e seu inerente Perfil para UML, denominado *NOP Profile* (WIECHETECK, 2011) (WIECHETECK; STADZISZ; SIMÃO, 2011) (XAVIER, 2014).

2.3.1 Perfil UML para o PON – *NOP Profile*

O *NOP Profile* define os principais conceitos do PON por meio da utilização de mecanismos de extensão da UML (e.g. estereótipos) que permite uma nova sintaxe e semântica aos seus elementos, voltado para modelagem dos elementos que compõe uma aplicação PON (WIECHETECK; STADZISZ; SIMÃO, 2011). O Perfil UML é o conjunto de sintaxe e semântica, agrupados dentro de um pacote.

Esse perfil foi baseado no método de criação de perfis UML proposto pela literatura pertinente (WIECHETECK, 2011). A definição de um perfil UML envolve a criação de dois artefatos: modelo de domínio (ou metamodelo) e o perfil UML propriamente dito. O modelo de domínio descreve o *framework* de domínio da aplicação que serve de base para a criação do perfil UML, permitindo identificar seus estereótipos, valores etiquetados e restrições.

Segundo Lima (2008), os mecanismos que permitem extensão de novos elementos de modelagem ao metamodelo da UML são (RONSZCKA, 2012):

- Estereótipo: permite a classificação de um elemento de modelo de acordo com um elemento de modelo base já existente na UML. Estereótipos devem possuir restrições e valores etiquetados para adicionar informações necessárias aos novos elementos.
- Valor etiquetado: permite explicitar uma propriedade de um elemento, sendo que essas informações podem ser adicionadas a qualquer elemento ao modelo. No metamodelo, os valores etiquetados são representados como atributos da classe que define um estereótipo, mas no modelo são apresentadas em um novo compartimento dos elementos, denominado “tags”.
- Restrição: é uma informação semântica anexada a um ou mais elementos do modelo para expressar uma condição que o sistema deve satisfazer. Tal especificação é escrita em uma determinada linguagem de restrições.

A Figura 5 apresenta a identificação de Estereótipo, Valor Etiquetado e Restrição.

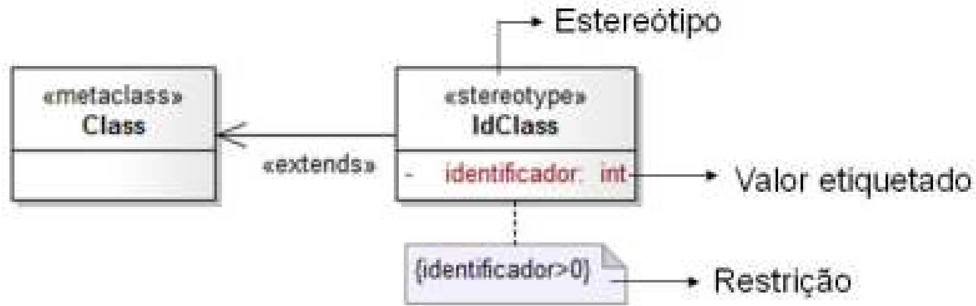


Figura 5 – Mecanismo de extensão da UML
Fonte: Wiecheteck, Stadzisz e Simão (2011)

O mecanismo de extensão da UML foi atribuído às entidades participantes do modelo de domínio de uma aplicação PON. Um modelo de domínio pode ser construído por meio dos elementos de modelagem da UML como classes, pacotes e associações (WIECHETECK; STADZISZ; SIMÃO, 2011). Assim, para a concepção de aplicações PON, as entidades participantes do modelo de domínio pertencem, basicamente, às classes que fazem parte dos pacotes *Application* e *Scheduler* esboçado pela Figura 6 e do pacote *Core* esboçado pela Figura 7 (RONSZCKA, 2012).

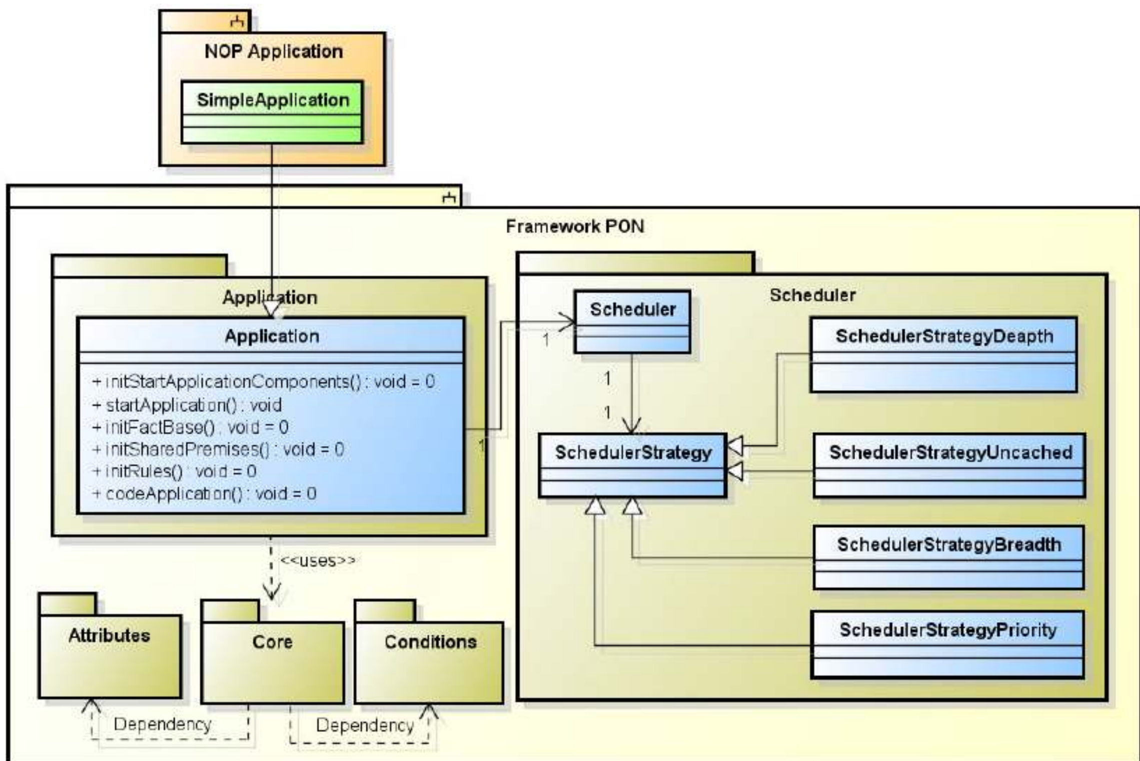


Figura 6 – Estrutura do *framework* PON Otimizado
Fonte: Ronszcka (2012)

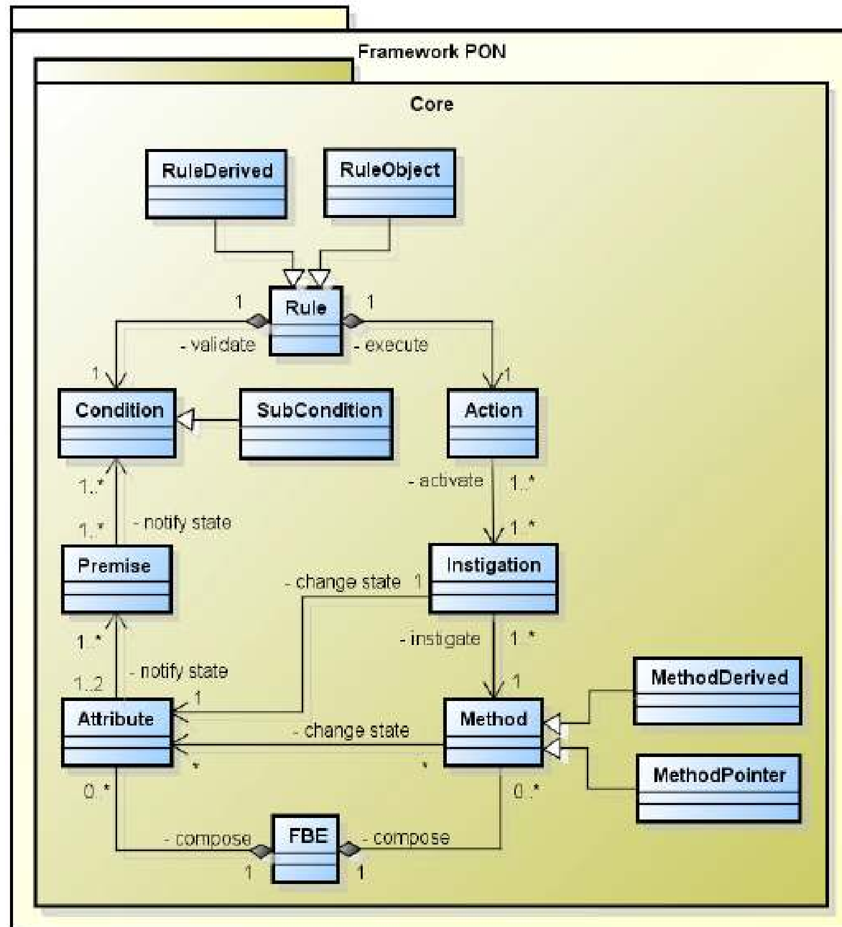


Figura 7 – Estrutura do pacote *Core*
Fonte: Ronszcka (2012)

Após a identificação dos modelos do domínio do PON, o próximo passo foi a criação do perfil UML para o PON, denominado o *NOP Profile*. O *NOP Profile* é composto por um pacote UML estereotipado (<<profile>>) com mesmo nome do perfil, que compreende outros dois pacotes – *NOP Profile Core* e *NOP Profile Application* – um para cada pacote de classes do *Framework* do PON (WIECHETECK; STADZISZ; SIMÃO, 2011).

O *NOP Profile Core* é composto por elementos de extensão da UML que representam e descrevem os objetos participantes do mecanismo de notificações do PON. Esses elementos de extensão foram obtidos a partir da análise do modelo do domínio da Figura 7 (que ilustra o diagrama de classes do *Framework* PON do pacote *Core*). Primeiramente, foi criado um estereótipo para cada elemento relevante definido no modelo do domínio e, em seguida, foram associados esses estereótipos aos elementos do metamodelo da UML por meio do relacionamento de extensão (<<extends>>). A Figura 8 exibe o perfil *NOP Profile Core* criado (WIECHETECK; STADZISZ; SIMÃO, 2011).

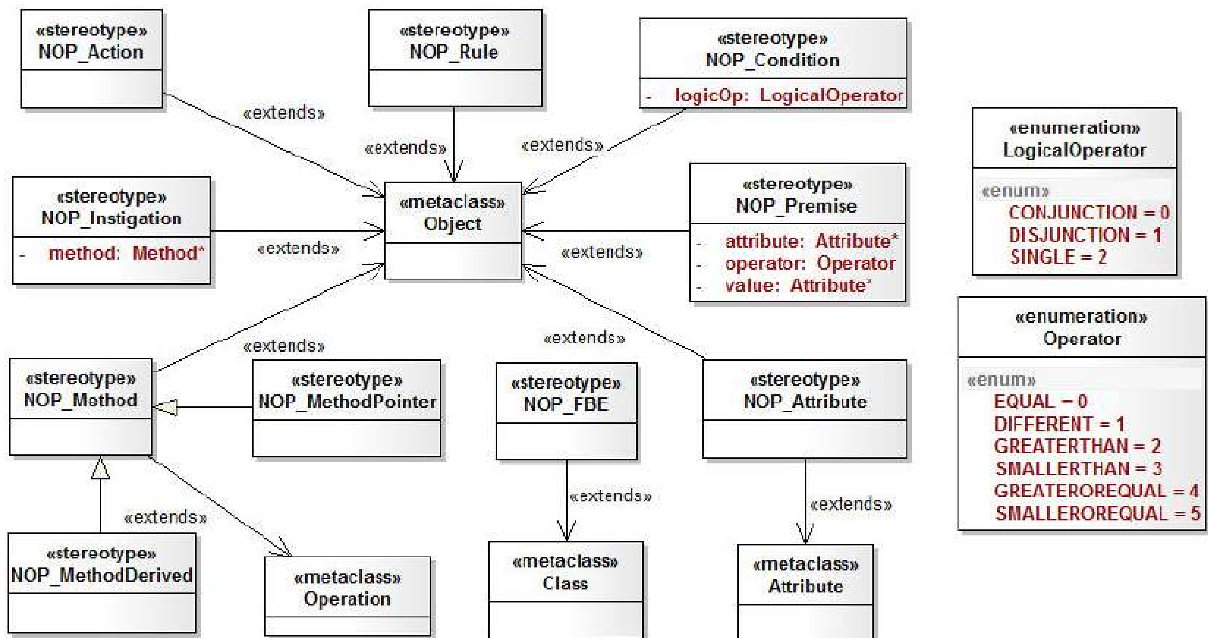


Figura 8 – *NOP Profile* pacote *Core*
Fonte: Wiecheteck, Stadzisz e Simão (2011)

Seguindo os mesmos passos para a criação do *NOP Profile Core*, foi construído o *NOP Profile Application* analisando-se o modelo de domínio da Figura 8. Nesse perfil, a classe *Application* do modelo do domínio foi transformada em um estereótipo – NOP_*Application* – que estende a metaclasse Class do metamodelo da UML, uma vez que uma nova aplicação no PON é representada como uma subclasse da classe *Application*. Já os métodos da classe *Application* do *Framework* PON foram transformados em estereótipos que estendem a metaclasse *Operation* da UML, sendo eles: initFactBase e init*Rules*. Também foi criada uma enumeração – *Scheduler*Strategy – que define as estratégias de resolução de conflitos entre regras existentes no PON, que podem ser: BREADTH, PRIORITY, DEPTH, UNCACHED e NO_ONE. A Figura 9 ilustra o pacote de perfil *NOP Profile Application* criado (WIECHETECK; STADZISZ; SIMÃO, 2011).

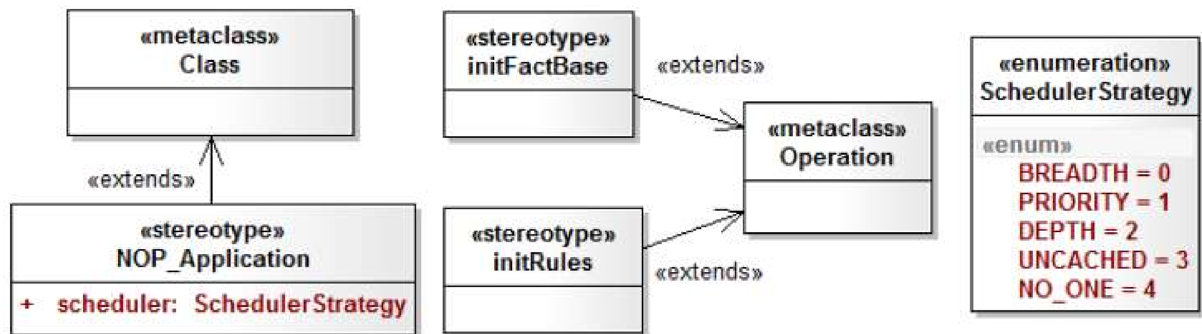


Figura 9 – *NOP Profile Application* pacote *Application*
Fonte: Wiecheteck, Stadzisz e Simão (2011)

2.3.2 Processo de Desenvolvimento Orientado a Notificações

Após a elaboração do perfil UML para o PON (*NOP Profile*), o arquiteto de software de aplicações do PON poderá usufruí-lo dentro do processo de Desenvolvimento Orientado a Notificações (DON) (RONSZCKA, 2012).

O Desenvolvimento Orientado a Notificações (DON) é um método que compreende as fases de requisitos e projeto de um processo de software (WIECHETECK, 2011). Quando comparado ao processo RUP – IBM *Rational Unified Process* (IBM, 2015), o DON compreende apenas as disciplinas de “Requisitos” e “Análise e Projeto”, conforme ilustrado na Figura 10, pois ele está focado unicamente nestas disciplinas.

O DON é organizado em oito passos: “Capturar requisitos”, “Criar Modelo de Caso de Uso”; “Criar Modelo de Classes”, “Criar Modelo de Atividades de Alto Nível”, “Criar Modelo de Componentes”, “Criar Modelo de Sequência”, “Criar Modelo de Comunicação”, “Criar Modelo de Redes de Petri”. Os dois primeiros passos fazem parte da fase de requisitos, enquanto os outros seis passos representam a fase de projeto de software (WIECHETECK, 2011) (XAVIER, 2014).

A modelagem estrutural é realizada por meio do Modelo de Classes, Modelo de Componentes e Modelo de Objetos (um novo Modelo de Objetos para o PON foi proposto nesta dissertação, diferente do modelo de objetos da UML). Por sua vez, a modelagem comportamental é realizada por meio do Modelo de Casos de Uso, Modelo de Estados de Alto Nível, Modelo de Sequência, Modelo de Comunicação e Modelo de redes de Petri (WIECHETECK, 2011).

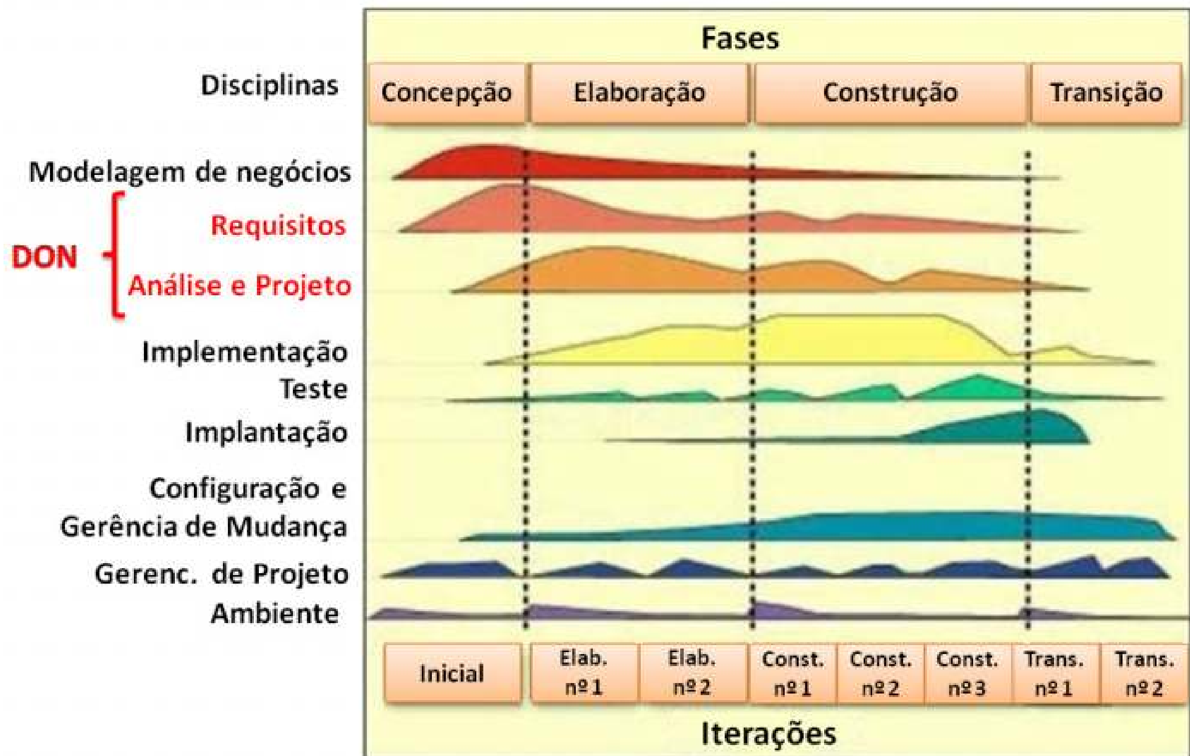


Figura 10 – DON contextualizado no RUP
Fonte: Wiecheteck (2011)

A etapa de Capturar Requisitos tem por objetivo obter os requisitos do sistema, que por sua vez, são fundamentais para o desenvolvimento da etapa Casos de Uso. Os diagramas de casos de uso proporcionam uma visão geral do sistema, sem se preocupar com particularidades de implementação. Essas duas etapas não sofreram mudanças com relação às mesmas correspondentes na UML.

O Modelo de Classes é baseado no diagrama de classes da UML. No DON, esse modelo requer a definição adicional da classe aplicação PON. Neste modelo, é definida a classe que representa a aplicação PON e são identificados os elementos *FBE* do sistema, *Attributes*, *Methods* e relacionamentos. Este diagrama utiliza estereótipos desenvolvidos para o Perfil UML para o PON. Nessa etapa é possível identificar os *FBEs* que irão compor o sistema.

Segundo Wiecheteck (2011), os Modelos de Atividades de Alto Nível podem auxiliar na identificação das *Rules* do sistema e podem ser representados pelo diagrama de máquinas de estados ou diagrama de atividades da UML. Esses diagramas acompanham as mudanças sofridas nos estados de uma instância de uma classe e outras instâncias relacionadas.

Os Modelos de Componentes são responsáveis por modelar a composição estática das *Rules* do sistema que já devem possuir *Premisses* e *Instigations* (identificados nos modelos de estados de alto nível). Nessa etapa, *FBE*, *Rules*, *Premisses* e *Instigations* são apresentados em detalhes, incluindo a comunicação entre esses elementos. Basicamente, é necessário cumprir três passos: definição das *Rules*, definição das *Premisses* e *Instigations*, e união das *Rules* aos *FBEs* (WIECHETECK, 2011).

O Modelo de Sequência é fundamentado no diagrama de sequência da UML, e permite identificar a ordem de execução das regras por meio da modelagem das notificações disparadas entre os objetos colaboradores. A fim de facilitar a modelagem do sistema, este modelo pode ser criado com um nível de abstração maior, no qual os *Attributes*, *Methods*, *Premise* e *Instigation* são suprimidos (WIECHETECK, 2011).

O Modelo de Comunicação é baseado no diagrama de comunicação da UML e apresenta as mesmas informações do modelo de sequência, entretanto, não se preocupa com a cronologia do processo, mas com os objetos vinculados mensagens trocadas entre eles.

O Modelo de redes de Petri apresenta a interação dinâmica entre os objetos do sistema PON e permite fazer a validação do modelo por meio da sua simulação ou análise (qualidade inerente às redes de Petri). Esse modelo também permite uma visão dinâmica global do sistema. As redes de Petri permitem a modelagem de concorrência, sincronização e compartilhamento de recursos do sistema, além da análise comportamental.

O DON é composto por três ciclos de desenvolvimento básicos (WIECHETECK, 2011). O ciclo 1 propõe a execução das fases de captura de requisitos e criação/refinamento do modelo de casos de uso. Quando essas duas fases estiverem refinadas é possível iniciar o ciclo 2. O ciclo 2 propõe o desenvolvimento dos modelos de classes, modelo de estados de alto nível, modelo de componentes, modelo de sequência, modelo de comunicação e modelo de redes de Petri. Por fim, o ciclo 3 propõe refinamentos nos modelos elaborados na fase 2. A Figura 11 ilustra eses ciclos.

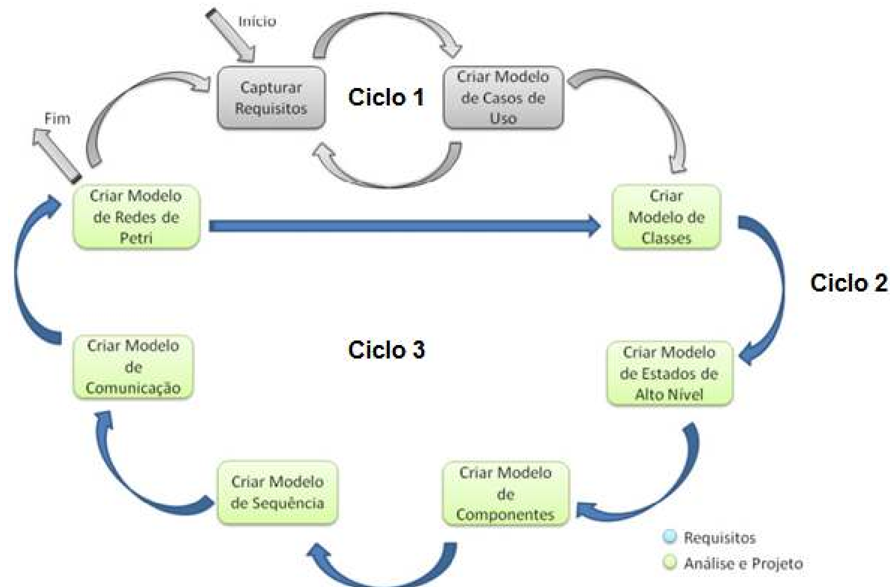


Figura 11 – Resumo dos ciclos do método DON
Fonte: Adaptado de Wiecheteck (2011)

Segundo Wiecheteck (2011), através do método DON com a utilização do *NOP Profile* é possível conceber os artefatos relacionados às fases de levantamento de requisitos e análise/projeto de software, as quais compõem as fases de concepção e elaboração do modelo RUP.

2.4 TESTE DE SOFTWARE

Nesta subseção são apresentados conceitos fundamentais sobre teste de software, incluindo as fases envolvidas, técnicas, critérios de teste e ferramentas. Esses conceitos não só são importantes para contextualizar esta pesquisa, mas serão também empregados na proposição do método de teste para PON.

2.4.1 Origem, Objetivos e Terminologias Básicas do Teste de Software

Até o início da década de 1980, o processo de testes era a última etapa no desenvolvimento de software, possuía o intuito de comprovar o funcionamento do produto e era conduzido eventualmente pelos próprios desenvolvedores. Essa visão começou a mudar com o trabalho de Myers (1979), que afirmava ser função dos testes descobrir erros, e não provar que o software estava funcionando. Gelperin e

Hetzel (1988) descreveram uma (re) evolução dos testes, propondo o documento de plano de testes, que deveria ser escrito a partir dos requisitos do software. O teste de software, que era realizado de maneira esporádica, passou a partir de então a ser incorporado no processo de desenvolvimento de software.

Com o amadurecimento sobre a importância da atividade de teste de software foram definidas algumas terminologias básicas (DELAMARO; MALDONADO; JINO, 2007):

- Defeito (*fault*) – passo, processo ou definição de dados incorretos. Por exemplo: instrução escrita incorretamente.
- Engano (*mistake*) – ação humana que produz um resultado incorreto. Por exemplo: erros cometidos pelo programador.
- Erro (*error*) – diferença entre o valor obtido e o valor esperado. Por exemplo: o resultado de uma adição não corresponde ao valor esperado com os valores informados. Normalmente, o erro é causador de uma ou mais falhas.
- Falha (*failure*) – produção de uma saída incorreta com relação à especificação.
- Domínio: conjunto de todos os valores que podem ser utilizados para executar um programa P;
- Dado de Teste: um elemento do domínio de entrada de um programa P;
- Caso de Teste: um par formado por um dado de teste mais o resultado esperado para a execução do programa com aquele dado;
- Conjunto de Teste: casos de teste usados durante uma determinada atividade de teste;
- Oráculo: Quem determina se o resultado obtido com o teste está correto (v.g. testador ou outro mecanismo).

Em geral, os erros são classificados em erros computacionais e erros de domínio. Os erros computacionais provocam uma computação incorreta, mas o caminho executado (sequência de instruções) é igual ao caminho esperado. Os erros de domínio executam um caminho diferente do caminho esperado, ou seja, quando um caminho errado for selecionado durante a execução de um caso de teste (DELAMARO; MALDONADO; JINO, 2007).

2.4.2 Fases do Teste

Resumidamente, quatro fases ou tipos de testes são efetuados durante o processo de desenvolvimento do software (MYERS et al., 2004):

- Teste de Unidade (ou Teste Unitário): testa a menor unidade do software (e.g. trecho de código, função, método de classe ou mesmo a própria classe), examinando a estrutura de dados e identificando erros de lógica e cálculo.
- Teste de Integração: testa a integração entre as unidades de software, busca identificar erros de interface entre os mesmos.
- Teste de Validação: testa se o software está em conformidade com os requisitos (e.g. funcionais, informativos, comportamentais, de manutenibilidade e configuração entre outros).
- Teste de Sistema: é uma série de diferentes testes cuja finalidade primária é exercitar totalmente o sistema; são testados aspectos com relação à recuperação, segurança, esforço, desempenho e inclusive elementos e sistemas externos.

Técnicas e critérios de testes têm sido propostos para nortear a atividade de teste, auxiliando a seleção e avaliação de conjuntos de casos de teste, visando determinar aqueles que aumentam as chances de revelar defeitos (MYERS et al., 2004). Muito embora haja novas proposições no tocante a testes no chamado estado da arte, esse trabalho considera principalmente as técnicas e abordagens mais consolidadas até então, o chamado “estado da técnica”.

2.4.3 Técnicas e Critérios de Teste

As técnicas mais conhecidas para avaliar a qualidade da atividade de teste de software são: a *técnica funcional*, que usa a especificação funcional do software para derivar casos de teste; a *técnica estrutural*, que deriva casos de teste a partir do código fonte do software; e a *técnica baseada em defeitos*, que deriva casos de teste para mostrar a presença ou ausência de defeitos comuns que podem acontecer na elaboração de um programa (DELAMARO; MALDONADO; JINO,

2007). As subseções a seguir, descrevem essas três principais técnicas de teste e alguns critérios de teste pertencentes a elas.

2.4.3.1 Técnica funcional

O teste funcional concentra-se na busca de circunstâncias em que o programa executável não apresenta comportamento de acordo com as especificações, ou seja, não está relacionado com a análise do comportamento interno ou estrutura do software via verificação de código. Essa técnica propõe a derivação de casos de teste por meio do fornecimento de condições de entrada que exercitem completamente todos os requisitos de um software (MYERS et al., 2004).

Os critérios de testes funcionais mais comuns que podem revelar defeitos são (MYERS et al., 2004): particionamento em classes de equivalência, análise de valor limite, grafo causa efeito e suposição de erro.

- Particionamento em classes[2] de equivalência

  O domínio de entrada de um programa é identificado na especificação e dividido em classes de equivalência válidas e inválidas, sendo que são selecionados casos de teste a partir de classes geradas. É testado de um pequeno subconjunto de todas possíveis entradas. Testa-se um pequeno subconjunto porque se presume que é inviável testar todas as possibilidades, que exigiriam muito tempo e esforços computacionais para sua realização.

- Análise de valores-limite

  Considerando o particionamento em classes de equivalência, selecionam-se os casos de teste que estão nas fronteiras das classes, porque nesses pontos pode estar concentrado o maior número de defeitos. Com isto, o primeiro, o intermediário e o último elemento do conjunto de valores são testados.

[2] O termo “classe”, nesse caso, está relacionado a um intervalo de valores que tem a mesma importância. Ou seja, qualquer valor escolhido seria adequado e apresentaria o mesmo impacto no teste. Assim, este termo difere substancialmente do significado de classe na orientação a objetos.

- Grafo causa-efeito

  Causas (condições de entrada) e efeitos (ações) são relacionados em um módulo em teste. Desenvolve-se um grafo que é convertido em uma tabela de decisão e esta é transformada em casos de teste.

- Suposição de erro

  A intuição e experiência do testador são consideradas para elaborar testes que explorem erros prováveis.

Por meio dessa técnica, é possível revelar defeitos relacionados a funções incorretas, erros de interface, estrutura de dados ou no acesso externo a bases de dados, desempenho, inicialização ou finalização. Assim, para este tipo de teste, é essencial uma especificação bem elaborada e de acordo com as necessidades do cliente. Dado que a maioria desses critérios de teste foram adaptados para geração de casos de teste para o PON serão apresentadas com mais profundidade e exemplificados no Capítulo 4.

2.4.3.1.1 Técnica de teste funcional com casos de uso

Na UML, o modelo de casos de uso possui os elementos chave Ator, Fronteira do Sistema e Instância do Caso de Uso que caracterizam o diagrama (BOOCH; RUMBAUGH; JACOBSON, 2005). Ator: qualquer pessoa ou sistema externo submetendo ou recebendo informações do sistema sob teste. Fronteira de sistema: uma abstração de interface que aceita e transporta entradas e produz saídas do sistema. Instância de caso de uso: um caso de uso com valores específicos determinados por parâmetros passados ao sistema sob teste, que define os resultados esperados com valores de entrada específicos. A Figura 12 apresenta os elementos básicos que fazem parte de um diagrama de casos de uso.

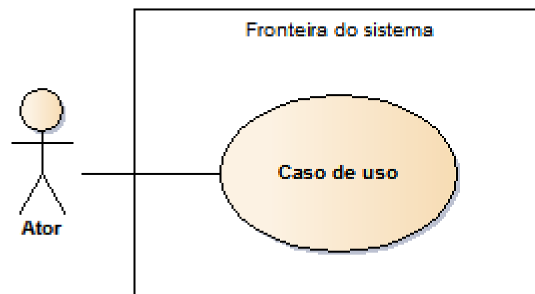


Figura 12 – Elementos básicos que compõem um diagrama de casos de uso
Fonte: Adaptado de Booch, Rumbaugh e Jacobson (2005)

Casos de uso são compostos por operações. Uma operação é uma sequência particular de mensagens trocadas entre objetos, inicializada por uma entrada externa e causa um caminho particular a ser percorrido, que pode ser apresentado por meio de um diagrama de sequência ou outro diagrama dinâmico (BOOCH; RUMBAUGH; JACOBSON, 2005) (BINDER, 1999). Os casos de uso apresentam muitas outras informações de um sistema como:

- Requisitos funcionais;
- Alocação de funcionalidade de classes (objetos);
- Interação entre objetos;
- Interfaces de usuário;
- Documentação para o usuário.

Além disso, um sistema especificado com casos de uso provê alguns recursos que possibilitam a realização de testes e também podem ser aplicados para analisar problemas de projeto como (JACOBSON, 1995):

- Determinação de fronteiras de sistema;
- Particionamento arquitetural;
- Determinação de incrementos no desenvolvimento;
- Rastreabilidade;
- Conceituação e prototipagem.

Jacobson (1992) lista quatro tipos gerais de testes que podem ser derivados a partir de casos de uso:

1) Teste de caminho básico ou caminhos esperados no fluxo de eventos;

2) Teste de caminhos alternativos ou todos os outros caminhos possíveis;
3) Teste de qualquer item de requisito rastreável em cada caso de uso;
4) Teste de características descritas na documentação rastreáveis para cada caso de uso.

O diagrama de casos de uso permite a realização de testes de integração e testes de sistema. O teste de integração exercita a operabilidade básica entre todas as colaborações de componentes e cenários que realizam um caso de uso. O teste de sistema exercita os requisitos do software. Considerando isto, qualquer falha que possa existir precisa ser exercitada por combinações de entradas e estados. Segundo Myers (2004), testar combinações de condições e valores de fronteiras é a abordagem mais efetiva para provocar possíveis falhas. Para isto, Binder (1999) sugere um procedimento genérico para realização de casos de teste com casos de uso:

1) Identificação de variáveis operacionais

As variáveis operacionais são todas as variáveis que fazem parte da realização do caso de uso, inclusive estados de objetos. Podem ser identificadas e analisadas no modelo de relacionamentos de entradas e/ou saídas em uma tabela de decisão (BINDER, 1999). São fatores que variam de um cenário para outro e determinam diferentes respostas do sistema como:

- Entradas e saídas explícitas;
- Condições ambientais que resultam em diferentes comportamentos do ator;
- Abstrações do estado do sistema sob teste (e.g., o estado do jogo precisa ser “jogando”, o jogador precisa ter mais que zero de pontos de vida, e assim por diante).

2) Definição de domínios das variáveis operacionais e relações operacionais

Os domínios são identificados pela definição do conjunto de valores válidos e inválidos para cada variável operacional em uma tabela de decisão. Tabela de decisão é uma técnica que auxilia na criação de casos de teste, sendo que modela os testes baseados nas regras de negócio e todas as relações operacionais prováveis. Com isso, verifica se as possíveis combinações do sistema estão sendo

manipuladas de acordo com o previsto. Em uma tabela de decisão, cada linha é denominada uma variante de teste (BRIAND; LABICHE, 2001). A Tabela 1 apresenta um exemplo de tabela de decisão.

Tabela 1 – Exemplo de uma tabela de decisão para um caso de uso qualquer

Variantes (caso de uso V)	Condições			Ações	
	A	B	C	I	II
V_1	Sim	Sim	Sim	Sim	Sim
V_2	Sim	Sim	Não	Sim	Não
V_3	Sim	Não	Não	Sim	Não
V_4	Não	Não	Não	Não	Não
V_5	Não	Não	Sim	Não	Sim
V_6	Não	Sim	Sim	Não	Não

Fonte: Adaptado de Briand e Labiche (2001)

Por exemplo, a variante V_1 necessita que as condições A, B e C sejam verdadeiras para que possa executar as ações I e II. A variante V_2 necessita que as condições A e B sejam verdadeiras e que a condição C seja falsa para que possa executar a ação I, porém, com essas condições não executa a ação II.

3) Projetar e executar casos de teste

Cada variante, ou seja, cada caso de teste identificado com a tabela de decisão precisa ser testada. São requeridos, no mínimo, dois casos de teste para cada variante: (1) um caso de teste verdadeiro é um conjunto de valores que satisfazem todas as condições em um variante, (2) um caso de teste falso é uma mudança nos valores de entrada ou estado em que pelo menos uma condição será falsa. Casos de teste podem cobrir todas as possíveis variações de condições (variáveis operacionais) de entrada verdadeiras e falsas que devem resultar em diferentes comportamentos do sistema (BRIAND; LABICHE, 2001).

2.4.3.2 Técnica estrutural

Esta técnica considera o aspecto de que um programa pode ser decomposto em um conjunto de blocos disjuntos de comandos, sendo que a execução de um

implica a execução de outros comandos subsequentes a este (MALDONADO, 1991).

A representação mais comum do fluxo de execução de um sistema é um grafo de fluxo de controle (GFC) $G = (N, E, s)$, onde *N* representa o conjunto de nós, *E* o conjunto de arcos e s o nó de entrada. Um caminho é uma sequência finita de nós (n_1, n_2,..., n_k), k ≥ 2, tal que existe um arco de n_i para n_{i+1}, para i = 1, 2,..., k-1. Um caminho é um *caminho simples* se todos os nós que compõem este caminho, exceto possivelmente o primeiro e o último, são distintos. Se todos os nós são distintos é tido que este *caminho é livre de laço*. Um *caminho completo* é um caminho no qual o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo *G* (MALDONADO, 1991). Para exemplificar, um grafo de fluxo de controle é apresentado na Figura 13 e seu respectivo código em linguagem C++ no Algoritmo 2.

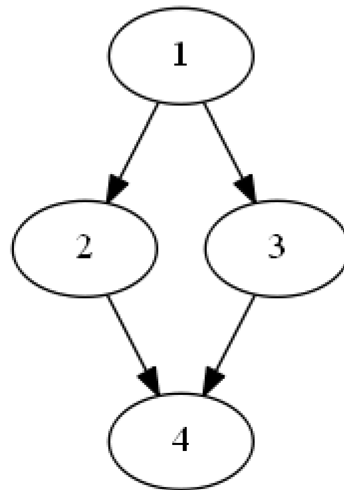


Figura 13 – Exemplo de grafo de fluxo de controle de um programa *P*

```
              include <stdio.h>
              int main()
/* 1 */       {
/* 1 */          int n;
/* 1 */          printf("informe um numero");
/* 1 */          scanf("%d",&n);
/* 1 */          if((n%2)==0);
/* 2 */          {
/* 2 */             printf("par");
/* 2 */          }
/* 3 */          else
/* 3 */          {
/* 3 */             printf("impar");
/* 3 */          }
/* 4 */          return 0;
              }
```

Algoritmo 2 – Código fonte de um programa *P* dividido por blocos de instruções. Este código refere-se ao grafo apresentado na Figura 13

Geralmente, fazem parte do teste estrutural os critérios baseados em fluxo de controle, os critérios baseados em fluxo de dados e critérios baseados em complexidade. Assim, essa técnica de teste preocupa-se com o comportamento interno do código: instruções que são ou não executadas, desvios, sequências, estados que desencadeiam diferentes caminhos, etc. (MYERS et al., 2004).

2.4.3.2.1 Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam informações de fluxo de controle (decisões) para determinar quais caminhos devem ser exercitados. Os critérios mais conhecidos são (MYERS et al., 2004):

- Todos-Nós: exige que seja executado cada vértice do GFC, pelo menos uma vez.
- Todas-Arestas (ou Todos-Arcos ou Todos-Ramos): exige que sejam executados todos os desvios do GFC, pelo menos uma vez.
- Todos-Caminhos: exige que todos os caminhos possíveis do programa sejam executados.

Além destes critérios, existem vários outros, entretanto, foram citados os mais conhecidos.

Para exemplificar a análise de fluxo de controle, o Algoritmo 3 apresenta um método em C++ dividido em seguimentos de código e a Figura 14 apresenta o Grafo de Fluxo de Controle (GFC) deste programa.

Linha	Instruções	Seguimento
/*1*/	public void runGame(){	A
/*2*/	for(int i = 0; | i < lstChar.getSize(); | i++){	B, C
/*3*/	if(lstChar[i].lifePoints>0 | && lstChar[i].active == true) {	D, E
/*4*/	lstChar[i].draw();	F
/*5*/	lstChar[i].movement();	
/*6*/	lstChar[i].attack();	
/*7*/	}	
/*8*/	if(lstChar[i].collision == true){	G
/*9*/	lstChar[i].decreaseLifePoints();	H
/*10*/	}	
/*11*/	}	I

Algoritmo 3 – Seguimentos de código no método runGame
Fonte: Adaptado de Binder (1999)

No GFC da Figura 14, um seguimento é representado por um nodo (círculo/elipse), um ramo é representado por um arco de saída, uma extremidade dirigida (seta) mostra qual outro seguimento poderá ser alcançado e um caminho é composto por seguimentos conectados por setas. A ocorrência de operações lógico-relacionais é apresentada como um quadrado envolvendo um ou mais nodos (que apresentem uma condição/predicado) e contendo uma letra que a identifica (BINDER, 1999).

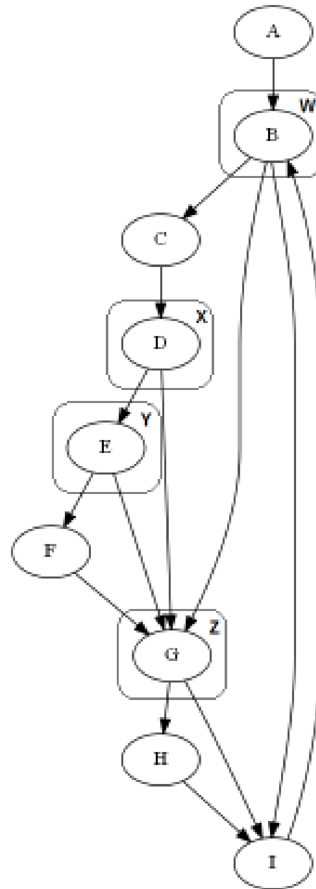


Figura 14 – Grafo de fluxo de controle para o método runGame
Fonte: Adaptado de Binder (1999)

A Tabela 2 apresenta quais caminhos serão cobertos pela execução do programa. Os *loops* são representados por seguimentos entre parênteses, seguidos de um asterisco indicando que este grupo pode iterar de zero a *n* vezes.

Tabela 2 – Cobertura de caminhos no método runGame

			Caminho de condições por nodo			
			B	**D**	**E**	**G**
	Caminho de Entrada/Saída	**Cobertura de ramo**	C1	C2	C3	C4
1	ABI	WI	F	DC	DC	DC
2	A(BCDEFGHI)*	WCXYFZHI	T	T	T	T
3	A(BCDEGHI)*	WCDYGHI	T	T	F	T
4	A(BCDEGI)*	WCDYGI	T	T	F	F
5	A(BGHI)*	WZHI	F	DC	DC	T
6	A(BGI)*	WZI	F	DC	DC	F
Legenda: T = Condição precisa ser verdadeira, F = Condição precisa ser falsa, DC = Caminho inalcançável. C1: i<lstChar.getSize() / C2: lstChar[i].lifePoints>0 / C3: lstChar[i].active==true / C4: lstChar[i].collision==true						

Fonte: Adaptado de Binder (1999)

2.4.3.2.2 Critérios baseados em fluxo de dados

Os critérios baseados em fluxo de dados utilizam informações de fluxo de dados para determinar os caminhos que devem ser exercitados no teste. A seguir, são apresentadas dois grupos de critérios baseados em fluxo de dados: critérios propostos por Rapps e Weyuker (RAPPS; WEYUKER, 1985) (RAPPS; WEYUKER, 1982) e critérios propostos por Maldonado (MALDONADO, 1991):

a) Critérios de Rapps e Weyuker

Dentre os critérios de Rapps e Weyuker, os principais são:

Todas-definições: requer que cada definição de variável seja exercitada pelo menos uma vez, por um c-uso (uso computacional, v.g. atribuição de variável) ou p-uso (uso predicativo, v.g. avaliação de variável em expressões relacionais – *if*, *then*, *else*).

Todos-usos: requer que todas as associações entre uma definição de variável e seus subsequentes usos (c-usos ou p-usos) sejam exercitadas pelos casos de teste, através de um caminho em que a variável não é redefinida.

Todos-DU-caminhos: requer que toda associação entre uma definição de variável e seus subsequentes (c-usos ou p-usos) sejam exercitados por todos os caminhos livres de definição e livres de laço de repetição que cubram esta associação.

b) Critérios de Maldonado – critérios de potenciais usos

Este critério requer associações independentemente da ocorrência implícita de uma referência a uma determinada definição: se um uso desta definição puder existir – um potencial uso – a potencial associação é requerida. Portanto, propõe que sejam explorados todos os possíveis efeitos a partir de uma mudança de estado do programa em teste.

2.4.3.2.3 Critérios baseados na complexidade

Os critérios baseados na complexidade utilizam informações de complexidade[3] do programa para derivar requisitos de teste. Um dos critérios mais conhecidos desta classe é o Critério de McCabe (MCCABE, 1976), que utiliza a complexidade ciclomática do grafo de programa para derivar requisitos de teste. Essencialmente, este critério requer que um conjunto de caminhos linearmente independentes do grafo de programa sejam executados (DELAMARO; MALDONADO; JINO, 2007).

2.4.3.3 Técnica baseada em defeitos

O teste baseado em defeitos procura mostrar quais defeitos conhecidos não estão presentes no programa, produzindo um conjunto de dados de teste que diferenciem o programa original de programas similares. Esses programas similares são gerados a partir de modificações no programa original, de acordo com defeitos previamente conhecidos. São critérios de teste baseados em defeitos a Semeadura de Erros (DEMILLO; LIPTON; SAYWARD, 1978) e a Análise de Mutantes (BUDD, 1980).

2.4.3.3.1 Semeadura de erros

A partir de um programa que já foi objeto de teste e cujos erros são conhecidos, é introduzida uma quantidade definida de erros artificiais (novos). Os casos de teste que já foram realizados sobre o programa anteriormente são repetidos para o programa modificado e verifica-se a quantidade de novos erros que foram identificados. A proporção entre o total de novos erros identificados pelo total de novos erros introduzidos permite fazer uma estimativa sobre a quantidade de erros remanescentes para o conjunto de teste definidos para o programa. Se os casos de teste foram capazes de identificar 60% dos novos erros pode-se supor que

[3] De modo geral, a complexidade é o número máximo de casos de teste requeridos por um critério, no pior caso.

isto se verifique para o conjunto total dos erros do programa original. Assim, é possível comparar a estimativa de erros remanescentes com a confiabilidade esperada para o software (DEMILLO; LIPTON; SAYWARD, 1978).

Essa técnica pode ser confundida com o teste de mutação (que será apresentado na subseção a seguir), entretanto são abordagens diferentes. Por meio da revisão de literatura pertinente, é possível observar que teste de mutação foi objeto de estudo em muito mais trabalhos que o teste semeadura de erros, possivelmente, por apresentar uma evolução, de certo ponto de vista, sobre o teste de semeadura de erros (DEMILLO et al., 1987) (OFFUTT; HAYES, 1996) (OFFUTT et al., 1996).

2.4.3.3.2 Teste de mutação

O Teste de Mutação ou Análise de Mutantes é um critério de teste baseado em defeitos, que considera duas suposições (DEMILLO; LIPTON; SAYWARD, 1978). A primeira é a suposição do programador competente, na qual todo programa criado por um programador competente está correto ou próximo ao correto. A segunda suposição é o efeito do *acoplamento*[4], na qual testes projetados para revelar defeitos simples podem revelar defeitos complexos ou em outras palavras, defeitos complexos estão associados a defeitos simples.

Deste modo, para revelar tais defeitos, a análise de mutantes identifica os desvios sintáticos mais comuns cometidos pelos programadores (como trocar nome de variável, instruções incorretas como troca de operadores lógicos) e aplica pequenas transformações (por meio de operadores de mutação) sobre o programa em teste (programa original) que conduzem a um programa incorreto (AGRAWAL et al., 1989).

Offutt e Hayes (1996) definem que um mutante deve ser um programa cuja diferença semântica seja pequena em relação ao original. Assim, no teste de mutação, defeitos típicos de software são utilizados para derivar requisitos de teste.

---

[4] É possível observar que a denotação de acoplamento em teste de mutação difere completamente em relação à engenharia de software, outra área que utiliza a mesma expressão. O acoplamento em ES significa o quanto uma classe depende de (as) outra (s) para funcionar. Considera-se que quanto maior for esta dependência, maior será o acoplamento.

O programa que está sendo testado é alterado diversas vezes, em geral, com uma alteração de cada vez, gerando um conjunto de "mutantes".

Para exemplificar, considere que o programador deveria escrever a seguinte instrução apresentada no Algoritmo 4 (KOSCIANSKI; SOARES, 2007):

```
if (a == b)
```

Algoritmo 4 – Declaração de instrução utilizando o operador de igualdade
Fonte: Koscianski e Soares (2007)

Para exercitar os caminhos criados nesse código são necessários dois casos de teste: um em que as variáveis sejam iguais e outro em que sejam diferentes. Porém, caso o programador tenha cometido um erro e escreveu a seguinte instrução apresentada no Algoritmo 5:

```
if (a >= b)
```

Algoritmo 5 – Declaração de instrução utilizando operador de maior ou igual
Fonte: Koscianski e Soares (2007)

O caso de teste {a=1; b=2} forneceria o mesmo resultado tanto na versão correta quanto na versão incorreta do programa, ou seja, é incapaz de revelar um defeito. Para que fosse possível revelar um defeito, caso de teste poderia ser {a=2; b=1}.

Então, para modelar os desvios sintáticos mais comuns (como troca de operador lógico), são utilizados os operadores de mutação, que são aplicados a um programa *P*, transformando *P* em um programa mutante Mi (DELAMARO; MALDONADO; JINO, 2007). A Figura 15 ilustra o processo genérico do teste de mutação, envolvendo os seguintes passos:

- Após gerar os mutantes, o programa original *P* é executado com o conjunto de casos de teste *T* selecionados e é verificado se o comportamento do programa *P* é o esperado.
- Se não for, ou seja, se o programa apresenta resultados incorretos para algum caso de teste, então um erro foi detectado.

- Caso contrário, o teste continua com a execução dos mutantes *Mi* usando o conjunto de casos de teste *T*.
- Se um mutante *Mi* apresenta resultado diferente de *P*, isso significa que os casos de teste conseguiram expor a diferença entre *P* e *Mi;* neste caso, *Mi* será considerado “morto” será descartado.
- Por outro lado, se *Mi* apresenta comportamento igual a *P*, diz-se que *Mi* continua vivo. Isso pode significar uma fraqueza em *T*, pois não conseguiu diferenciar *P* de *Mi,* ou que *P* e *Mi* são equivalentes.

O objetivo é que todos os mutantes sejam mortos por *T*, porque quando algum mutante permanece vivo, significa que o conjunto de teste *T* é incapaz de revelar o erro causado pelo defeito no ponto onde houve a mutação.

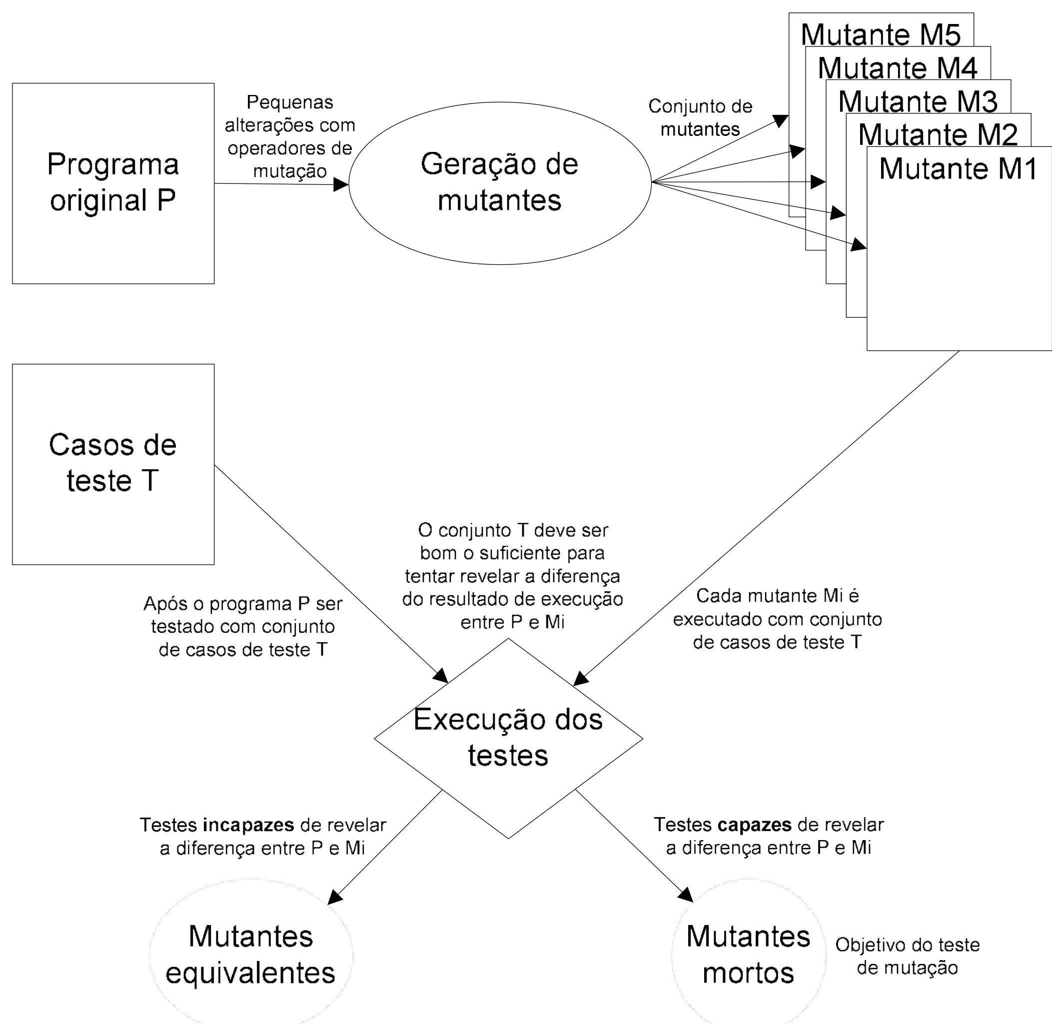


Figura 15 – Processo genérico do teste de mutação
Fonte: Adaptado de Thakur (2014)

Com a execução dos mutantes é possível ter uma ideia da adequação dos casos de teste utilizados, por meio do “escore de mutação” (*mutation score*). O es*core* de mutação varia entre 0 e 1 e fornece uma medida objetiva de quanto o conjunto de casos de teste analisado aproxima-se da adequação (valor mais próximo a 1). Dado o programa *P* e o conjunto de casos de teste *T*, calcula-se o es*core* de mutação $\mathrm{ms}\,(\mathrm{P,T})$ de acordo com a equação (1) (DELAMARO; MALDONADO; JINO, 2007):

$$\mathrm{ms}(\mathrm{P,T}) = \frac{\mathrm{DM}\ (\mathrm{P,T})}{\mathrm{M(P)} - \ \mathrm{EM(P)}} \quad (1)$$

$DM(P,T)$: número de mutantes mortos pelo conjunto de casos de teste

$M(P)$: número total de mutantes gerados a partir do programa P

$EM(P)$: número de mutantes gerados que são equivalentes a P

Assim, o escore de mutação pode ser obtido pelo cálculo da razão entre o número de mutantes efetivamente mortos por *T* e o número de mutantes que se pode matar, dado pelo número total de mutantes gerados subtraído do número de mutantes equivalentes. Embora seja um critério voltado para o teste de unidade, é possível adaptar o teste de mutação para outras fases do processo de software (DELAMARO; MALDONADO; JINO, 2007).

2.4.4 Principais Limitações da Atividade de Teste

Em geral, a atividade de teste possui algumas limitações (MALDONADO, 1991):

- Correção coincidente: quando um programa que contenha um defeito e, mesmo que seja executado, apresenta um resultado correto;
- Caminhos não executáveis: quando não há combinações possíveis de valores de entrada, parâmetros e variáveis globais que causem a execução de um determinado caminho no programa;
- Caminhos ausentes: quando uma funcionalidade que deveria existir, mas não foi implementada no programa;

- Mutantes equivalentes: quando dois programas diferentes computam a mesma função e produzem o mesmo resultado.

Logo, os critérios de teste podem ser utilizados em conjunto, pois incidem em problemas diferentes e são considerados complementares entre si.

2.4.5 Ferramentas de Teste

A realização da atividade de teste de software, em geral, é inviável na prática sem o suporte de ferramentas que a automatize, mesmo para programas pouco complexos (MYERS et al., 2004). A seguir são citadas algumas ferramentas que podem auxiliar essa atividade.

- JUnit: foi desenvolvido por Kent Beck e Erich Gamma em 1995 e, desde então, é o *framework* mais popular para teste de unidade em aplicações Java (TAHCHIEV et al., 2010).
- LDRA *Tool Suite*: é um conjunto de ferramentas que auxilia o desenvolvimento de software durante todo o ciclo de vida. Possui conformidade com os padrões de projeto, teste de software e as ferramentas de verificação. Apresenta integração com o rastreamento do ciclo de vida do software, análise estática e dinâmica, teste de unidade e sistema em qualquer plataforma alvo (LDRA, 2014).
- PokeTool: a ferramenta PokeTool (*Potential Uses Criteria Tool for Program Testing*) apoia a aplicação dos critérios de potenciais usos em programas escritos em C, Pascal, Fortran e Cobol (CHAIM, 1991);
- Mujava: apresenta vários operadores de mutação voltados ao teste de classes em Java (MA; KWON; OFFUTT, 2002).
- Mothra: utilizada para aplicação do critério de análise de mutantes em programas Fortran (DEMILLO et al., 1988);
- Depuradores: permitem a observação da execução de um programa a cada instrução e seu conteúdo na memória. Normalmente, estão associados a testes rápidos de fluxos de execução que podem ser realizados durante o desenvolvimento de software. Os depuradores atuais possibilitam também a modificação do conteúdo das variáveis durante a execução, para que o programador possa testar hipóteses de causas de defeitos. Atualmente, os

depuradores estão integrados a ambientes de desenvolvimento como Eclipse e Visual Studio (KOSCIANSKI; SOARES, 2007).

2.5 CONSIDERAÇÕES SOBRE A FUNDAMENTAÇÃO TEÓRICA

Esse capítulo apresentou os conceitos fundamentais utilizados para estruturar esta dissertação.

A Seção 2.1 apresentou uma visão geral sobre paradigmas e linguagens de programação, conceitos essenciais para situar o leitor sobre a classificação, as linguagens de programação mais representativas e as principais características dos paradigmas utilizados na atualidade. Ainda, foram apresentadas as principais limitações desses paradigmas no tocante à facilidade de desenvolvimento, distribuição em multiprocessadores, leitura de código, manutenção, entre outras.

A Seção 2.2 introduziu, resumidamente, o Paradigma Orientado a Notificações e seus principais conceitos. Foi apresentada a história e a fundamentação do PON para contextualizar esse paradigma. Também, foi apresentado o mecanismo de notificações do PON juntamente com detalhes de sua implementação (*framework*), estado da arte e da técnica, e principais vantagens em relação aos paradigmas dominantes.

A Seção 2.4 apresentou alguns conceitos, fases, técnicas, critérios e ferramentas mais utilizadas para a atividade de teste de software. Foi apresentada uma visão geral sobre as principais técnicas de teste: funcional, estrutural e teste baseado em erros, a fim de elucidar sobre a atividade de teste, que é uma das mais importantes em qualquer metodologia de desenvolvimento de software. Esses conceitos são fundamentais para o desenvolvimento da proposta de um método de teste de software para o PON que será apresentado no Capítulo 4.

3 DESENVOLVIMENTO DE UMA APLICAÇÃO PON

Este capítulo apresenta um caso de estudo referente à implementação simplificada de um software (jogo) de combate aéreo com interface gráfica em 2D (duas dimensões) que foi desenvolvido com auxílio do *Framework* PON Otimizado e da biblioteca gráfica externa Allegro C++. Esse software foi desenvolvido para auxiliar na compreensão do PON como paradigma de desenvolvimento. Além disso, no Capítulo 5, é apresentado um caso de estudo que aplica o método de teste proposto em todo esse software. A Seção 3.1 apresenta brevemente o software e a biblioteca gráfica utilizada, a Seção 3.2 apresenta a modelagem e implementação do software seguindo o DON. Concluindo este capítulo, a Seção 3.3 apresenta as considerações sobre o desenvolvimento em PON.

3.1 APRESENTAÇÃO DO SOFTWARE DESENVOLVIDO

O software desenvolvido é um jogo de combate aéreo, sendo o equivalente a uma implementação simplificada do jogo clássico “River Raid” (ATARI2600, 2015). Esse jogo foi escolhido por ser maior que estudos anteriores, mas ainda dentro de um tamanho razoável para um estudo prospectivo e ainda possuindo caráter lúdico e seu propósito é de fácil entendimento.

Esse software possui uma interface gráfica em duas dimensões que apresenta o movimento de um avião controlado pelo jogador que enfrenta inimigos que aparecem em posições aleatórias e podem atacá-lo. Essa aeronave move-se para frente e, também, pode atacar. O usuário do software (i.e. o jogador) pode controlar a aeronave da seguinte forma:

- Movimentação lateral, para a esquerda e para a direita, usando setas do teclado, para permitir alinhar a aeronave com um alvo inimigo à sua frente.
- Disparo de projétil a frente da aeronave, usando a barra de espaço do teclado.
- Pausar e continuar o jogo usando as teclas *Enter* e *Shift*, respectivamente.
- Encerrar o jogo usando a tecla *Esc*.

A aeronave e os inimigos possuem pontos de quantidade de vida que são decrementados a cada colisão com projétil. O objetivo do jogador com sua aeronave é terminar uma fase vencendo o maior número de inimigos e se esquivando de ataques. Para a implementação do software foi utilizado o *Framework* PON Otimizado e uma biblioteca gráfica externa.

A Figura 16 apresenta a tela principal de jogo. A biblioteca gráfica utilizada no desenvolvimento do software denomina-se Allegro (LIBALLEG, 2014). Allegro é uma biblioteca de código fonte aberto e multiplataforma para o desenvolvimento de *video games*. Suporta diversos recursos nativamente, como gráficos 2D, 3D, entrada de dados pelo teclado e mouse. Possui uma *API* voltada para iniciantes, facilitando o desenvolvimento de interfaces gráficas.

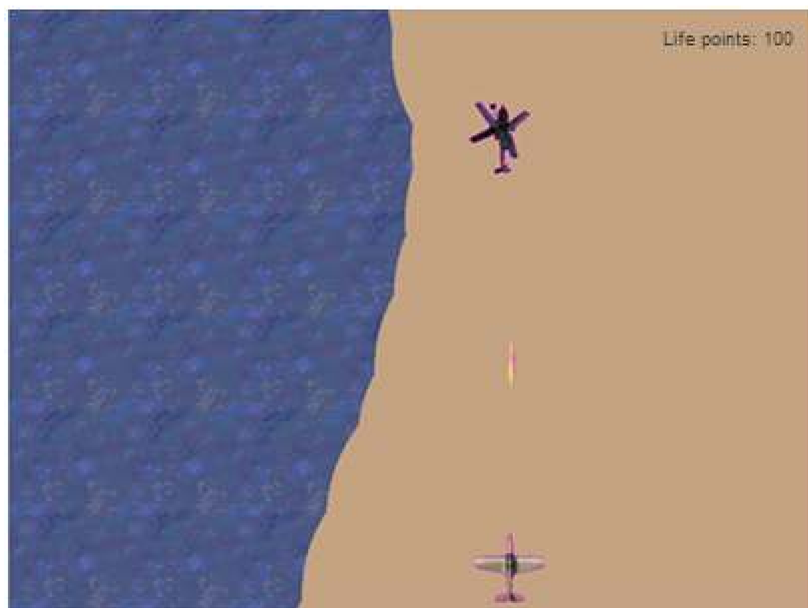


Figura 16 – Tela do software em execução

Alguns dos recursos básicos do Allegro, que foram utilizados no desenvolvimento deste software, são:

- Bitmap: uma matriz de pontos desenhada pelo Allegro definida nos eixos x e y indicando sua área ocupada. Os dois pares de coordenadas (inicial e final) determinam a posição e dimensão do retângulo que contém o bitmap (Figura 17).
- Draw_sprite: instrução que permite que os Bitmaps sejam desenhados e também trata imagens transparentes.

- Clear e Blit: Clear limpa toda a tela e Blit plota os Bitmaps. Quando os dois atuam em conjunto, as imagens estáticas são apagadas, reposicionadas e redesenhadas, podendo ser em uma nova posição, simulando movimento.
- Double buffering: constrói uma imagem em um *buffer* fora da tela para desenhar todos os Bitmaps em conjunto permitindo aumentar o desempenho em relação a uma solução mono *buffering* (HARBOUR, 2006).

Esses recursos foram considerados para o desenvolvimento de entidades PON (*Premisses*, *Conditions*, *Rules e Methods)*, conforme será visto nas seções de projeto do DON.

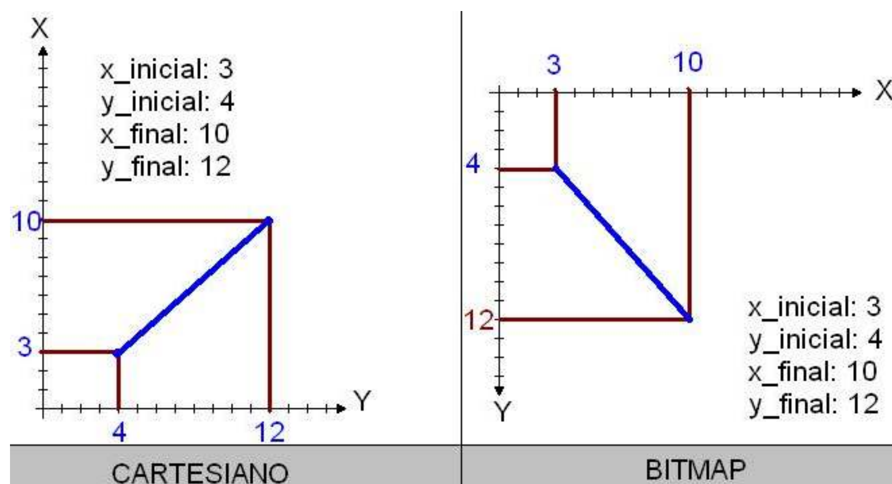


Figura 17 – Comparação entre plano cartesiano e o plano de pixels no Allegro
Fonte: Weissengeist (2014)

3.2 DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES (DON)

A elaboração do software utilizando o Desenvolvimento Orientado a Notificações (DON) resultou na criação de diversos artefatos:

1) Especificação de requisitos
2) Diagrama de casos de uso
3) Diagrama de classes
4) Levantamento de regras do sistema (modificado)
5) Diagrama de componentes
6) Diagrama de sequência
7) Diagrama de objetos (novo)

Seguindo o DON, a quarta etapa do método (Levantamento de regras do sistema), originalmente apresentada por Wiecheteck (2011), propõe a criação de vários diagramas de estados de alto nível para tentar auxiliar na atividade de identificação de *Rules* (e outras entidades PON) necessárias para o sistema. Porém, para essa dissertação, optou-se por não utilizar exatamente esta abordagem para o levantamento de regras, optou-se por modificá-la. O uso de diagrama de estados apresentado por Wiecheteck (2011) foi demonstrado de maneira muito simplificada, praticamente referindo-se a um sistema muito pequeno que fora desenvolvido, que não oferecia maiores complexidades e, principalmente, não apresenta uma técnica consolidada para extração de *Rules* a partir dos diagramas de estados. Se fosse utilizada a abordagem original, para o software apresentado nessa dissertação, seria necessário produzir grandes diagramas de estados que certamente dificultariam o processo de levantamento de regras. Por isso, optou-se pela simplicidade, por meio de uma análise sobre os requisitos e os casos de uso para identificar as regras necessárias para o software, abordagem essa que se apresentou muito eficaz. Do mesmo modo, aqui não foi apresentada uma técnica para o levantamento de regras, apenas buscou-se responder às três perguntas propostas por Wiecheteck (2011), porém com uma abordagem diferente. Nessa dissertação, considera-se que o levantamento de regras é uma atividade criativa e está relacionada à experiência e capacidade do analista em identificar regras (*Rules)* – e outras entidades – necessárias para o sistema em PON.

Também, optou-se por não utilizar os diagramas de comunicação e de redes de Petri. Segundo Wiecheteck (2011), os diagramas de comunicação são opcionais, apresentam as mesmas informações do diagrama de sequência, mas com enfoque diferente, que não causaria impacto no desenvolvimento do método de teste proposto, principal objeto de estudo nesta dissertação. Por sua vez, a modelagem de uma arquitetura de software PON em redes de Petri, embora intuitiva até certo ponto, ainda exigirá o estabelecimento de um método validado para poder ser feita para todos os casos. Wiecheteck (2011) propôs técnicas parciais para esta modelagem, mas elas não foram concluídas. Como não é propósito dessa dissertação desenvolver tais técnicas ou método, optou-se por não incluí-los neste estudo.

Adicionalmente, nessa dissertação, foi desenvolvido um diagrama de objetos para apresentar o conjunto de instâncias de regras e seus elementos que compõem

o software PON. Para tal, propôs-se uma notação gráfica particular, mais intuitiva do que a notação UML convencional. Esse diagrama mostrou-se muito útil para visualização e compreensão do ciclo de notificações e interações entre elementos. Além disso, facilita a visualização da rede de alcançabilidade existente no PON, conforme será visto no Capítulo 4.

3.2.1 Especificação de Requisitos

Com base na observação de jogos de combate aéreo similares, foi realizado um estudo para levantar os requisitos básicos de um jogo deste tipo. A Tabela 3 apresenta requisitos funcionais, requisitos não funcionais e os requisitos de *design* definidos para este software.

Tabela 3 – Requisitos funcionais, não funcionais e de *design* do software

#	Requisitos Funcionais
RF001	O sistema deverá oferecer pelo menos uma fase para o jogador.
RF002	O sistema deverá fazer o Avião disparar projétil quando a tecla de espaço for pressionada.
RF003	O sistema deverá mover o terreno para baixo, simulando movimentação do Avião, quando o sistema estiver com estado jogando.
RF004	O sistema deverá movimentar os inimigos.
RF005	O sistema deverá fazer os inimigos dispararem um projétil a cada dois segundos.
RF006	O sistema deverá decrementar os pontos de vida de qualquer Personagem quando algum projétil acertá-lo.
RF007	O sistema fechará o jogo quando o Avião estiver com zero ou menos de pontos de vida.
RF008	O sistema deverá permitir que o jogador controle a movimentação do Avião para direita, esquerda ou que se mantenha na posição.
RF009	O sistema deverá pausar o jogo quando o jogador pressionar a tecla Enter durante a execução de uma fase e deverá continuar o jogo quando o jogador pressionar a tecla Shift.
RF010	O sistema deverá eliminar qualquer personagem quando o valor de pontos de vida seja zero ou menos.
RF011	O sistema apresentará as informações da quantidade de pontos de vida do Avião.
RF012	O sistema deverá apresentar a movimentação e o sentido de qualquer projétil que seja disparado pelos personagens.
RF013	O sistema deverá apresentar todos os personagens que possuam a quantidade de pontos de vida superior à zero.
RF014	O sistema deverá parar o jogo (finalizar) quando o jogador pressionar a tecla Esc durante a execução de uma fase.
RF015	O sistema deverá controlar a apresentação (desenho) dos personagens, projéteis, cenário.
RF016	O sistema deverá interpretar comandos de teclado enviados pelo jogador.
RF017	O sistema deverá fornecer um menu com a opção “Iniciar Jogo”.

Requisitos funcionais, não funcionais e de *design* para o software (continuação)

#	Requisitos não funcionais	Tipo
RNF001	O Avião não poderá extrapolar a delimitação da tela do sistema (800x600).	Usabilidade
RNF002	Os pontos de vida do Avião serão apresentados no canto superior direito da tela.	Interface com o usuário
RNF003	O sistema deverá ter um Menu Principal que terá uma imagem de fundo.	Interface com o usuário
RNF004	O sistema limitará a apresentação do Avião apenas na região inferior da tela.	Interface com o usuário
RNF005	O sistema permitirá a movimentação do Avião apenas na horizontal.	Usabilidade
RNF006	O sistema deverá oferecer, no mínimo, um ou mais inimigos do tipo helicóptero.	Interface com o usuário
#	**Requisitos de projeto**	
RD001	O sistema deverá ser desenvolvido em *Framework* PON Otimizado.	
RD002	O sistema deverá ser desenvolvido em ambiente Windows 7.	
RD003	O sistema deverá ser desenvolvido com a biblioteca gráfica Allegro.	

3.2.2 Modelo de Casos de Uso

A segunda etapa do desenvolvimento consistiu na elaboração do diagrama de casos de uso que proporciona uma visão geral do software e é fundamentado nos requisitos levantados. A Figura 18 apresenta o diagrama de casos de uso elaborado para este software.

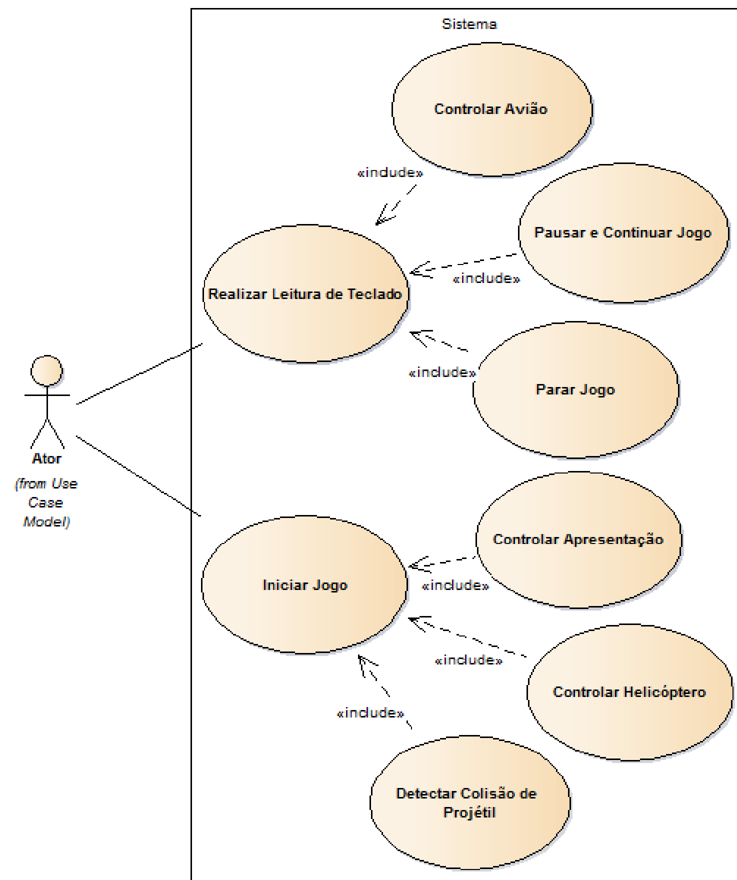


Figura 18 – Diagrama de casos de uso do software

A Tabela 4 apresenta uma descrição sucinta sobre as funcionalidades que cada caso de uso realiza. As descrições completas serão apresentadas nos capítulos seguintes.

Tabela 4 – Descrição sucinta de cada caso de uso

Caso de uso	Descrição sucinta de sua funcionalidade
Iniciar Jogo	Responsável por iniciar ou reiniciar o jogo.
Controlar Apresentação	Concentra as funcionalidades que são realizadas com auxílio da biblioteca gráfica Allegro, como desenho de personagem e desenho de cenário.
Controlar Avião	O principal caso de uso do jogo. Por meio dele o jogador pode movimentar e atacar com o avião.
Controlar Helicóptero	Responsável pela movimentação e o ataque do personagem helicóptero.
Realizar Leitura de Teclado	Responsável por realizar a leitura do teclado e interpretar os comandos que devem ser realizados durante a execução do jogo.
Detectar Colisão de Projétil	Responsável por detectar colisão entre projéteis e personagens (avião ou helicóptero) e decrementar pontos de vida quando ocorrem as colisões.
Pausar e Continuar Jogo	O jogador pode pausar ou continuar o jogo pausado.
Parar Jogo	O jogador pode parar o jogo. Quando isto ocorre, o jogo é encerrado.

O modelo de casos de uso tem papel importante nesta dissertação. Alguns critérios do método de teste de software proposto para o PON consideraram esse diagrama, conforme será visto no Capítulo 4.

3.2.3 Modelo de Classes

No modelo de classes proposto pelo DON é definida uma classe que representa a aplicação PON e são identificados os elementos *FBE*, *Attribute* e *Methods*. A Figura 19 apresenta o diagrama de classes resumido desenvolvido para esse software. O diagrama de classes completo com os *Attributes* e *Methods* está disponível no APÊNDICE A.

Figura 19 – Diagrama de classes resumido do software

A aplicação deve iniciar estendendo a classe *Application* do *framework*. No exemplo, essa classe é chamada SimpleFlightCombatSimulator e recebe o estereótipo <<NOP_Application>> definido no pacote *Core-Class* do *NOP Profile*. Quando aplicado este estereótipo, a classe automaticamente herda o valor etiquetado ”*scheduler*”. Neste exemplo, *scheduler* foi definido como “BREADTH”. Complementarmente, foram adicionados os estereótipos de operação *codeApplication*, *initFactBase*, *initRules*, *initSharedEntities* e *initStartApplicationComponents*.

No modelo de classes da Figura 19 também são apresentados os *FBEs* identificados: Character e as classes derivadas Allegro_Character, Airplane, Enemy e Helicopter; Stage e as classes derivadas Allegro_Stage e Stage1; Bullet e a sua única classe derivada, Allegro_Bullet; e Allegro_Keyboard, que não possui classes derivadas.

3.2.4 Levantamento das Regras do Software

Conforme descrito anteriormente, o levantamento das Regras ou *Rules* necessárias para compor o software utilizado nessa dissertação é uma tarefa criativa do analista e realizada por meio da análise dos casos de uso e requisitos definidos. Assim, o primeiro passo é relacionar os requisitos com cada caso de uso, conforme apresentado na Tabela 5.

Tabela 5 – Relação entre os casos de uso e seus requisitos

Casos de uso	Requisitos relacionados
Controlar Avião	RF002, RF008, RNF004, RNF005, RF014, RNF001
Pausar e Continuar Jogo	RF009
Parar Jogo	RF007
Realizar Leitura de Teclado	RF002, RF008, RF009, RF016
Controlar Helicóptero	RF004, RF005, RNF006
Detectar Colisão de Projétil	RF006, RF010
Controlar Apresentação	RF001, RNF002, RF003, RF011, RF012, RF013, RNF001, RNF002, RNF003, RNF004, RNF005, RF014, RF015
Iniciar Jogo	RF017

Na sequência, é realizada uma investigação a fim de se levantar o conjunto de *Rules* necessárias para realizar cada caso de uso. Não há, atualmente, uma técnica consolidada para guiar o desenvolvedor nesta atividade de concepção. Por

isso, neste estudo optou-se pela utilização parcial da abordagem empírica proposta por Wiecheteck (2011) que procura responder as seguintes questões:

- Qual é o propósito da *Rule*?
- Quais fatores precisam ocorrer para que a *Rule* seja executada?
- Quais as consequências da execução da *Rule*?

As partir da Tabela 6 é apresentado o conjunto de regras identificadas e as respostas a essas questões, conforme será visto a seguir.

A Tabela 6 apresenta a descrição da *Rule* que deve movimentar o cenário do jogo. Essa *Rule* deve implementar, parcialmente, o caso de uso “Controlar Apresentação” por meio do requisito RF003.

Tabela 6 – *Rule* 1: Movimentar o cenário para baixo

Passo	Resposta
1 – Propósito	Movimentar o cenário para baixo
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero
3 – Consequência	O cenário movimenta-se para baixo

A Tabela 7 apresenta a descrição da *Rule* que deverá atualizar as informações sobre o estado do jogador. Essa *Rule* implementará, parcialmente, o caso de uso “Controlar Apresentação” por meio dos requisitos RF011 e RNF002.

Tabela 7 – *Rule* 2: Atualizar o progresso do jogador

Passo	Resposta
1 – Propósito	Atualizar o estado do progresso do jogador
2 – Fatores	O sistema precisa estar com estado jogando ou pausado
3 – Consequência	O estado do progresso do jogador é atualizado e aparece no canto superior direito da tela

A Tabela 8 apresenta a descrição da *Rule* que manterá a posição do avião na tela de jogo. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Avião” por meio dos requisitos RF008 e RNF001.

Tabela 8 – *Rule* 3: Manter a posição do Avião

Passo	Resposta
1 – Propósito	Manter a posição do Avião
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero O jogador não pode ter pressionado o botão para movimentar o Avião para direita ou para esquerda
3 – Consequência	O Avião mantém a posição na tela

A Tabela 9 apresenta a descrição da *Rule* que movimentará o avião para a esquerda. Essa *Rule* implementará, parcialmente, o caso de uso “Controlar Avião” por meio dos requisitos RF008, RNF004 e RNF005.

Tabela 9 – *Rule* 4: Movimentar Avião à esquerda

Passo	Resposta
1 – Propósito	Movimentar Avião à esquerda
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero O Avião não pode estar no limite da borda esquerda da tela Jogador precisa ter pressionado o botão esquerdo
3 – Consequência	O Avião move-se para a esquerda quando o botão esquerdo for pressionado

Todas as outras *Rules* desenvolvidas para o software estão disponíveis no APÊNDICE B.

3.2.5 Modelo de Componentes

O modelo de componentes é responsável pela composição estática das *Rules* do sistema. Para isto, é usado o *NOP Profile* (WIECHETECK, 2011). Basicamente, existem três passos a serem seguidos para o cumprimento desta etapa: 1) definição de nome para as *Rules*, 2) definição das *Premisses* e *Instigations* das *Rules* e, 3) união das *Rules* aos *FBEs*. Esses passos serão aplicados no decorrer da seção.

Passo 1) definição das *Rules*

O primeiro passo consiste na atribuição de nomes para as *Rules* que foram identificadas anteriormente, conforme apresenta a Tabela 10.

Tabela 10 – *Rules* identificadas para o software

Rule	Nome		*Rule*	Nome
1	rlScenarioMoving		14	rlGameUnpause
2	rlProgressUpdating		15	rlGameStop
3	rlAirplaneMoving		16	rlAllegroClear
4	rlAirplaneMovingLeft		17	rlAllegroBlit
5	rlAirplaneMovingRight		18	rlAllegroKeyboard
6	rlAirplaneShoots		19	rlAllegroDrawAirplane
7	rlHelicopterShoots		20	rlAllegroDrawTheScenario
8	rlAllegroDrawHelicopter		21	rlAllegroDrawBullet
9	rlAirplaneDecreaseLifePoints		22	rlAllegroHelicopterCollision
10	rlHelicopterDecreaseLifePoints		23	rlAllegroAirplaneCollision
11	rlHelicopterDies		24	rlAllegroMoveTheBulletUp
12	rlPlayerGameOver		25	rlHelicopterMoving
13	rlGamePause		26	rlAllegroMoveTheBulletDown

Para cada uma dessas *Rules,* é necessário adicionar um componente <<NOP_*Rule*>>, um <<NOP_*Action*>> e um <<NOP_*Condition*>>. O relacionamento entre os componentes é representado pelos estereótipos <<*Rule*Notifies*Action*>> e <<*Condition*Notifies*Rule*>>, definido no pacote *Core-Assembly* para o Perfil PON, o qual expressa a semântica e as restrições entre estes componentes. A Figura 20 apresenta um exemplo para representação da *Rule* rlAirplaneMovingRight.

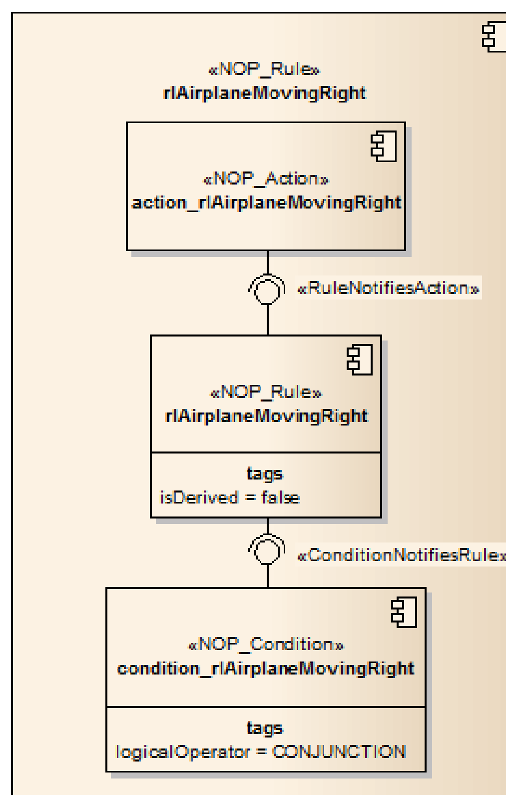


Figura 20 – Diagrama de componentes (passo 1): *Rule* com nome, *Condition* e *Method*

Quando identificadas, as *Premisses* e *Instigations* podem ser representadas por meio de um modelo de componentes de acordo com o nível de abstração que se queira representar. Em um alto nível de abstração, as *Premisses* e *Instigations* são representadas como interfaces de componentes, em um baixo nível de abstração, a representação pode ser feita apenas por componentes.

Passo 2) Definição das *Premisses* e *Instigations* das *Rules*

As *Premisses* e *Instigations* são identificadas por meio da análise dos *Attributes* e *Methods* dos *FBEs* do modelo de classes e as *Rules* são identificadas por meio dos requisitos e casos de uso. A Tabela 11 exibe as *Premisses* e *Instigations* definidas para cada *Rule* identificada.

Tabela 11 – *Rules*, *Premisses* e *Instigations* identificados

Rules	***Premisses***	***Instigations***
Rule 1 rlScenarioMoving	atGameStatus == PLAYING && atLifePoints (Airplane) > 0	Scenario -> mtMovement
Rule 2 rlProgressUpdating	atGameStatus == PLAYING || atGameStatus == PAUSED	Stage -> mtProgressUpdating
Rule 3 rlAirplaneStayInPosition	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atAirplaneRightButton == false && atAirplaneLeftButton == false	Airplane -> mtStayInPosition
Rule 4 rlAirplaneMovingLeft	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atPosX1 (Airplane) > 0 && atAirplaneLeftButton == true	Airplane -> mtMoveLeft
Rule 5 rlAirplaneMovingRight	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atPosX2 (Airplane) < 800 && atAirplaneRightButton == true	Airplane -> mtMoveRight
Rule 6 rlAirplaneShoots	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 atAirplaneFireButton == true atBullet > 0	Airplane -> mtAirplaneAddNewBullet
Rule 7 rlHelicopterShoots	atGameStatus == PLAYING && atLifePoints (Helicopter) > 0 && atTimeToShoot > 2 (segundos)	Helicopter -> mtHelicopterAddNewBullet

***Rules*, *Premisses* e *Instigations* identificados (continuação)**

Rule 8 rlAllegroDrawHelicopter	(atGameStatus == PLAYING || atGameStatus == PAUSED) && atLifePoints (Helicopter) > 0	Character -> mtDraw
Rule 9 rlAirplaneDecreaseLifePoints	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atCollision (Airplane) == true	Character -> mtDecreaseLifePoints
Rule 10 rlHelicopterDecreaseLifePoints	atGameStatus == PLAYING && atLifePoints (Helicopter) > 0 && atCollision (Helicopter) == true	Character -> mtDecreaseLifePoints
Rule 11 rlHelicopterDies	atGameStatus == PLAYING && atLifePoints (Helicopter) <= 0	Character -> mtDeath
Rule 12 rlPlayerGameOver	atGameStatus== PLAYING && atLifePoints (Airplane)<=0	Character -> mtDeath
Rule 13 rlGamePause	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atPauseButton == true	Stage -> mtGamePause
Rule 14 rlGameUnpause	atGameStatus == PAUSE && atUnpauseButton == true	Stage -> mtGameUnpause
Rule 15 rlGameStop	(atGameStatus == PLAYING || atStopButton == PAUSED) && atStopButton ==true	Stage -> mtGameStop
Rule 16 rlAllegroClear	atGameStatus == PLAYING || atGameStatus == PAUSED	Stage -> mtAllegroClear
Rule 17 rlAllegroBlit	atGameStatus == PLAYING || atGameStatus == PAUSED	Stage -> mtAllegroBlit
Rule 18 rlAllegroKeyboard	atGameStatus == PLAYING || atGameStatus == PAUSED	Stage-> mtKeyboardInterruption
Rule 19 rlAllegroDrawAirplane	(atGameStatus == PLAYING || atGameStatus == PAUSED) && atLifePoints (Airplane) > 0	Airplane -> mtDraw
Rule 20 rlAllegroDrawTheScenario	atGameStatus == PAUSED || atGameStatus == PLAYING	Scenario -> mtDraw
Rule 21 rlAllegroDrawBullet	atGameStatus == PLAYING || atGameStatus == PAUSED	Bullet -> mtDraw
Rule 22 rlAllegroHelicopterCollision	atGameStatus == PLAYING && atLifePoints (Helicopter) > 0 && atBulletPosX >= atPosX1 && atBulletPosX <= atPosX2 && atBulletPosY >= atPosY1 && atBulletPosY <= atPosY2	Helicopter -> mtCollision

***Rules*, *Premisses* e *Instigations* identificados (continuação)**

Rule 23 rlAllegroAirplaneCollision	atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atBulletPosX >= atPosX1 && atBulletPosX <= atPosX2 && atBulletPosY >= atPosY1 && atBulletPosY <= atPosY2	Airplane->mtCollision
Rule 24 rlAllegroMoveTheBulletUp	atGameStatus == PLAYING atLifePoints (Airplane) > 0	Bullet->mtBulletMovingUp
Rule 25 rlHelicopterMoving	atGameStatus == PLAYING atLifePoints (Helicopter) > 0	Helicopter->mtMoviment
Rule 26 rlAllegroMoveTheBulletDown	atGameStatus == PLAYING atLifePoints (Airplane) > 0	Bullet-> mtBulletMovingDown

Outra atividade realizada neste passo é a definição do operador lógico para a *Condition* dos elementos. A Figura 21 apresenta o operador lógico "CONJUNCTION" na *Condition* da *Rule* rlAirplaneMovingRight. Uma *Rule* pode suportar mais de um operador em *SubConditions* como, por exemplo, o operador "DISJUNCTION" da *Rule* rlAllegroDrawAirplane, que é apresentado na Figura 22.

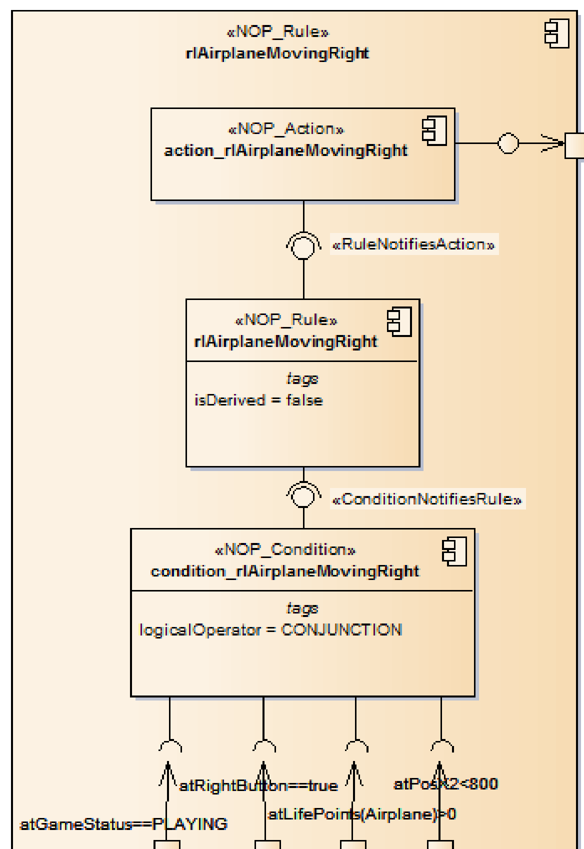


Figura 21 – Diagrama de componentes (passo 2): *Rule* e suas interfaces

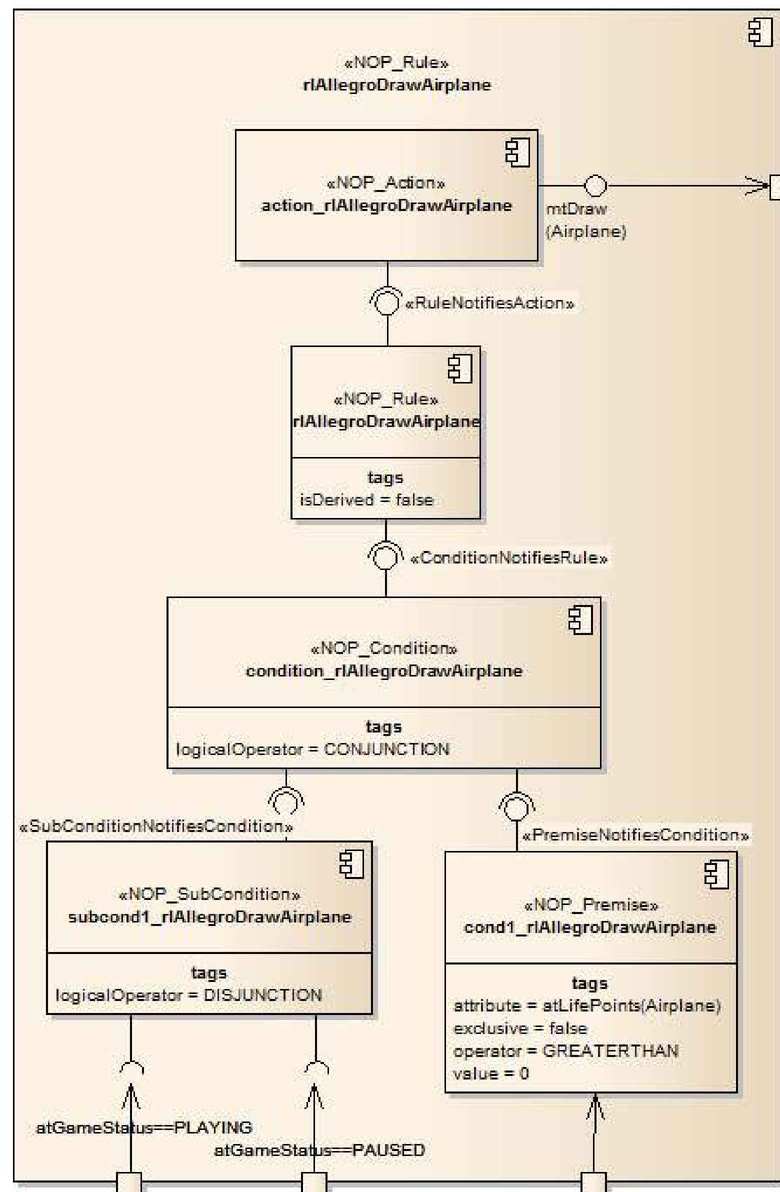


Figura 22 – Diagrama de componentes (passo 2): Operadores lógicos

Passo 3) Unir as *Rules* aos *FBEs*

Por fim, no último passo para a criação dos modelos de componente são apresentadas as conexões entre os *FBEs* e as *Rules*. A Figura 23 mostra um exemplo com a *Rule* rlAirplaneMovingRight, que inclui seus relacionamentos com os *FBEs* Allegro Keyboard (i.e. Teclado), Stage (i.e. Fase) e Airplane (i.e. Avião).

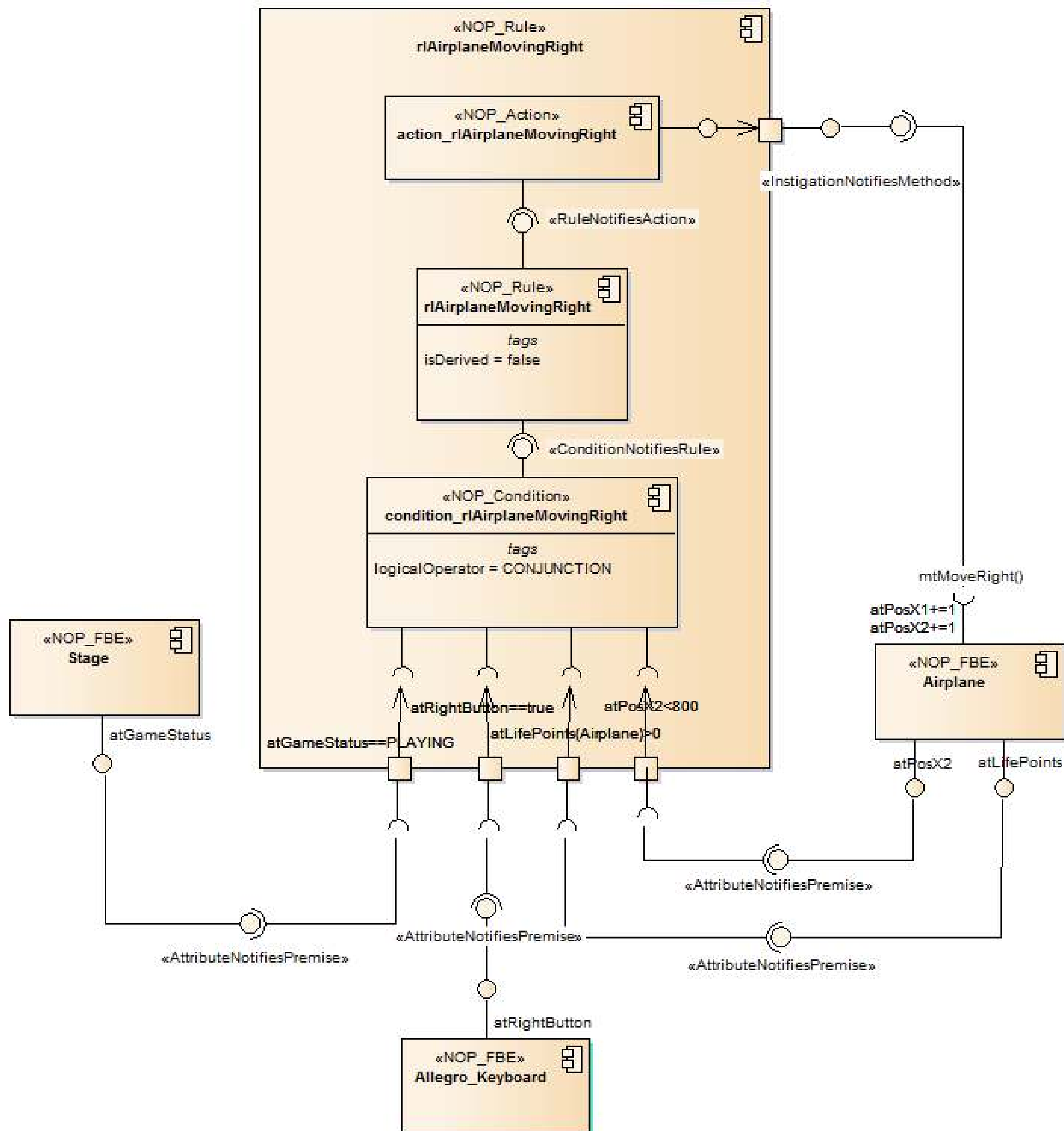


Figura 23 – Diagrama de componentes (passo 3): *Rule*, interfaces e componentes externos

Como pode ser visto na Figura 23, os atributos do *FBE* (e.g. atGameStatus, atLifePoints, atRightButton e atPosX2) são representados por interfaces fornecidas, enquanto os métodos diretos ou modificados são representados por atributos de interfaces utilizadas (e.g. mtMoveRight do Airplane), pelo fato deles serem fornecidos para outro componente.

Todos os outros diagramas de componentes estão disponíveis no APÊNDICE C.

3.2.6 Modelo de Sequência

O modelo de sequência apresenta uma visão da dinâmica de execução do sistema em PON. Para o desenvolvimento desta etapa, optou-se por utilizar o modelo de sequência resumido proposto por Wiecheteck (2011). Neste modelo, as entidades *Attribute*, *Method*, *Premise* e *Instigation* não aparecerem por questões de simplificação.

Para possibilitar a visualização de todas as entidades que compõem o diagrama de sequência do caso de uso Controlar Avião, foi necessário dividir o diagrama em três partes (ou visões) focando nas entidades que realizam as *Rules* que compõem o caso de uso: rlAirplaneMovingLeft (Figura 24), rlAirplaneMovingRight (Figura 25) e rlAirplaneShoots (Figura 26).

Por exemplo, na Figura 24 é possível observar a existência do objeto Allegro_Keyboard responsável por realizar a "leitura do teclado" para reconhecer e dar o tratamento adequado a tecla (ou botão) pressionada. Nesse diagrama é possível perceber que quando o jogador pressiona o botão esquerdo, o *Attribute* atLeftButton recebe valor verdadeiro. Esse *Attribute* faz parte de uma *Premise* que notifica a *Condition* cond_rlAirplaneMovingLeft. Quando essa *Condition* possui seu estado lógico verdadeiro notifica a *Rule* rlAirplaneMovingLeft, que por sua vez, notifica sua Acition que invoca, por meio de uma *Instigation*, o *Method* mtMoveLeft do *FBE* Airplane. Do mesmo modo, a Figura 25 apresenta as entidades envolvidas na execução da *Rule* rlAirplaneMovingRight.

Por sua vez, a Figura 26 apresenta as entidades envolvidas na elaboração e aprovação da *Rule* rlAirplaneShoots. Para isso, requer que o *Attribute* atFireButton seja verdadeiro para que a *Premise* que o avalia seja verdadeira e possa notificar a *Condition* cond_rlAirplaneShoots. Quando a *Condition* estiver satisfeita e for aprovada notifica a *Rule* rlAirplaneShoots para executar. Quando a *Rule* for aprovada, invoca o *Method* mtAirplaneAddNewBullet para executar. Esse *Method* realiza várias operações como a definição de valores para o *FBE* Allegro_Bullet, e para as *Rules* rlAllegroDrawBullet, rlAllegroMoveBulletUp e rlAllegroHelicopterCollision. Respectivamente, essas *Rules* são responsáveis por desenhar o projétil na tela, movimentar o projétil disparado e "conhecer" qual é o *FBE* que pode ser atingido, no caso, Helicopter.

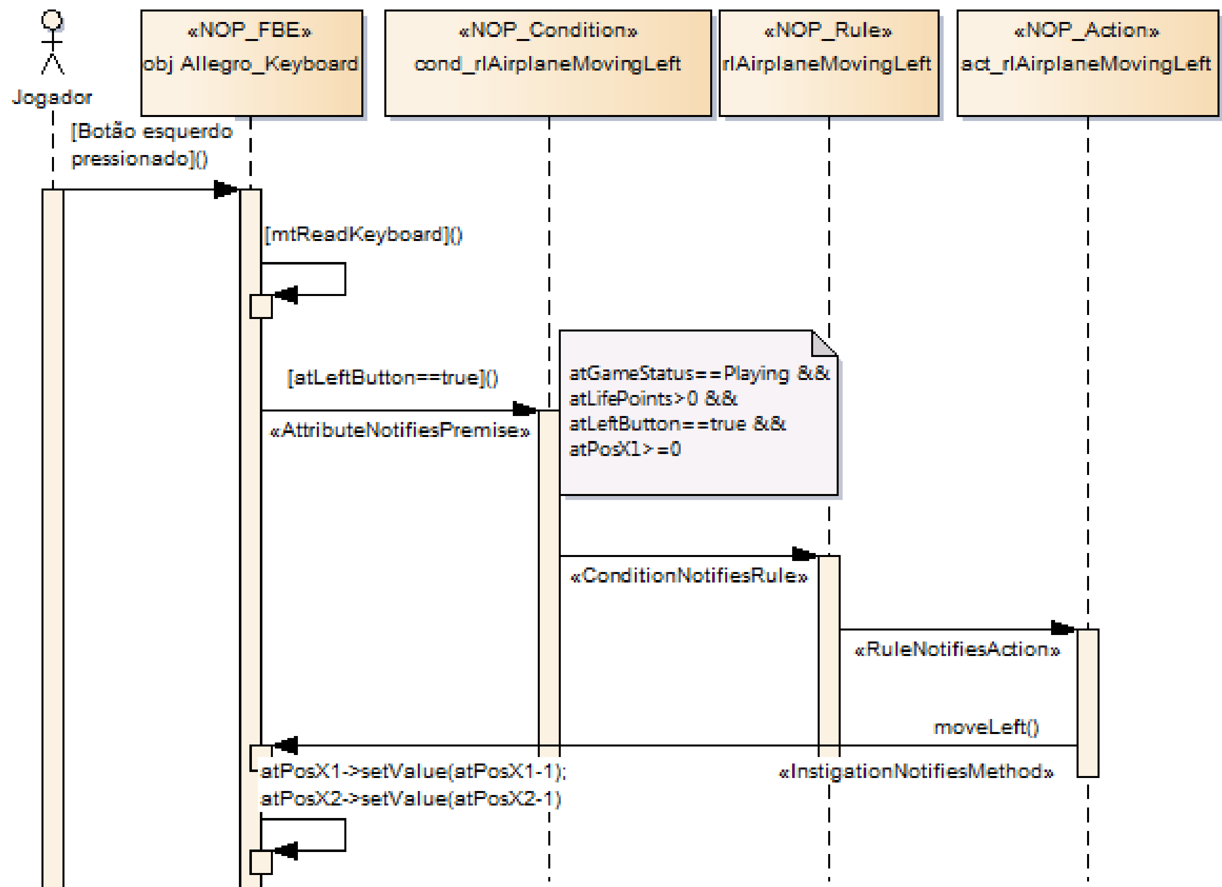


Figura 24 – Diagrama de sequência do caso de uso Controlar Avião (rlAirplaneMovingLeft)

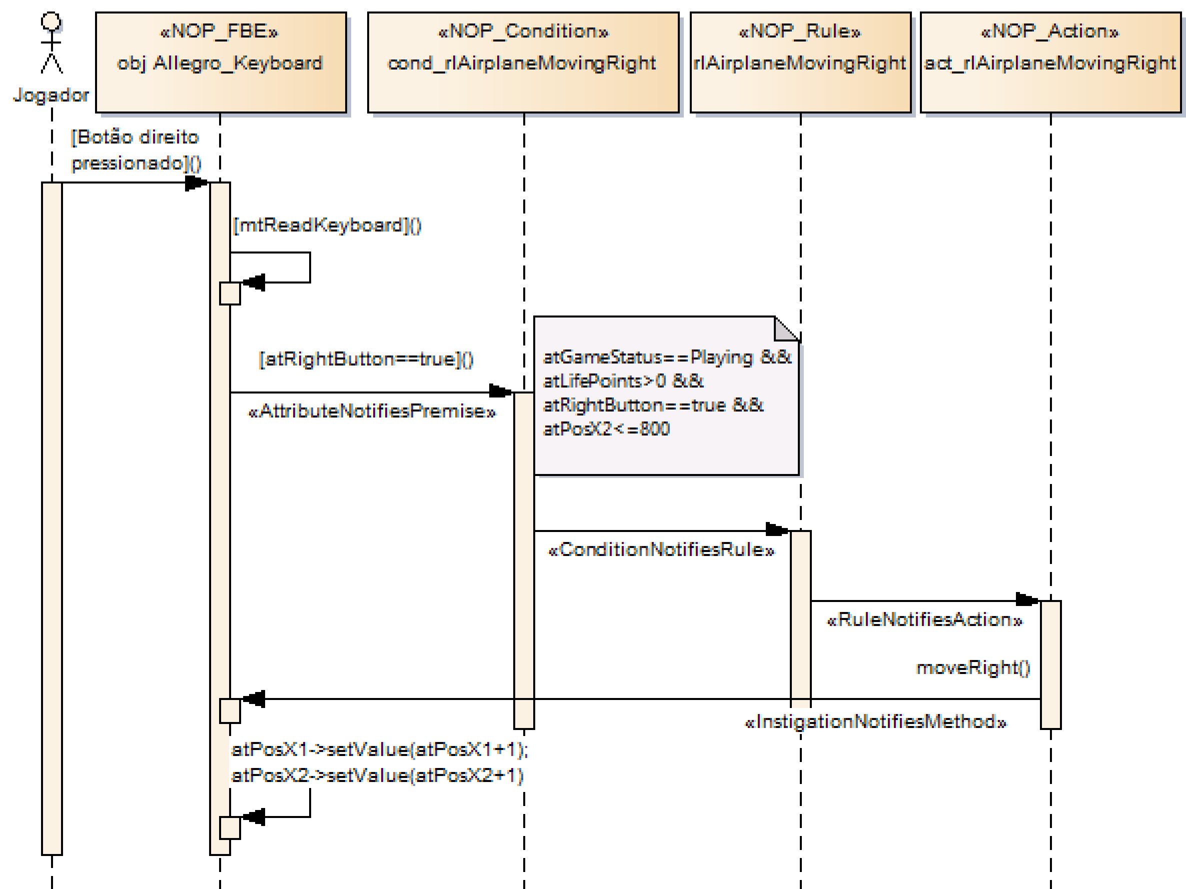


Figura 25 – Diagrama de sequência do caso de uso Controlar Avião (rlAirplaneMovingRight)

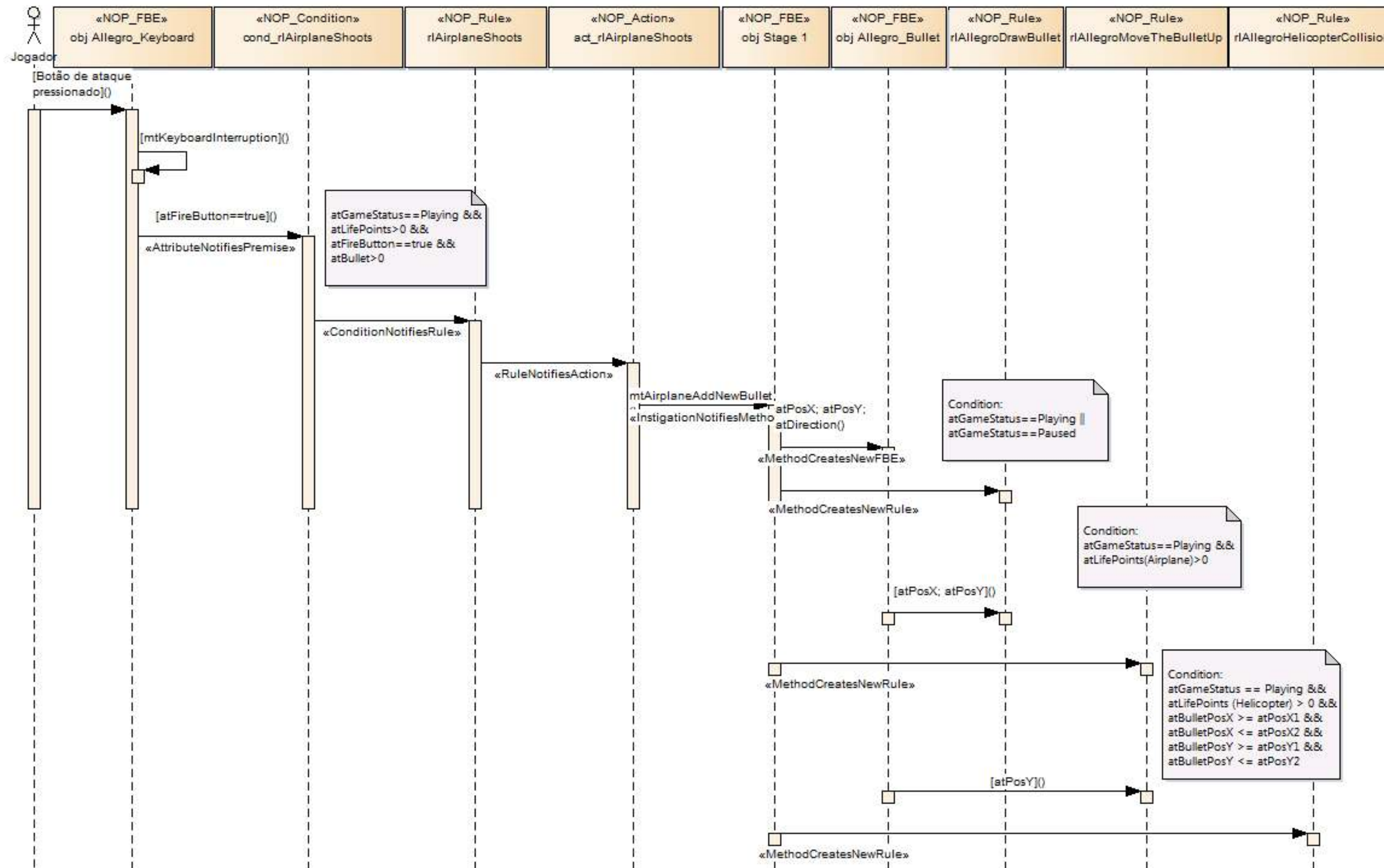


Figura 26 – Diagrama de sequência do caso de uso Controlar Avião (rlAirplaneShoots)

Todos os outros diagramas de sequência desse software estão disponíveis no APÊNDICE D.

3.2.7 Modelo de Objetos

Um modelo de objetos foi proposto e apresentado nessa dissertação. Tal diagrama consiste em um avanço de modelagem em relação ao então atual *status* do DON. Ainda que esta dissertação se oriente para testes, sentiu-se a necessidade desse diagrama para o PON.

Esse modelo apresenta uma visão dos elementos PON instanciados. Pode ser utilizado para representar as entidades que compõem o sistema e também, caso seja necessário, para isolar elementos que realizam determinado caso de uso, com o objetivo de facilitar análises específicas de fluxo de execução. Com isso, o desenvolvedor pode ter uma visão gráfica sobre o ciclo de notificações dos elementos envolvidos.

A Figura 27 apresenta a notação proposta para representar os objetos PON. Um *FBE* está representado como um objeto retangular. O *Attribute* é representado como um triângulo. A *Premise* é representada como um losango simbolizando uma decisão. Uma *Rule* é representada com um objeto que desencadeia um fluxo de execução em um sentido. Considerou-se *Condition* como parte integrante da respectiva *Rule*. O *Method* é representado como uma engrenagem, símbolo alusivo a uma ação no PON (v.g. alteração do valor de um *Attribute* e/ou criação de outros objetos PON). O sentido das notificações é representado como uma seta.

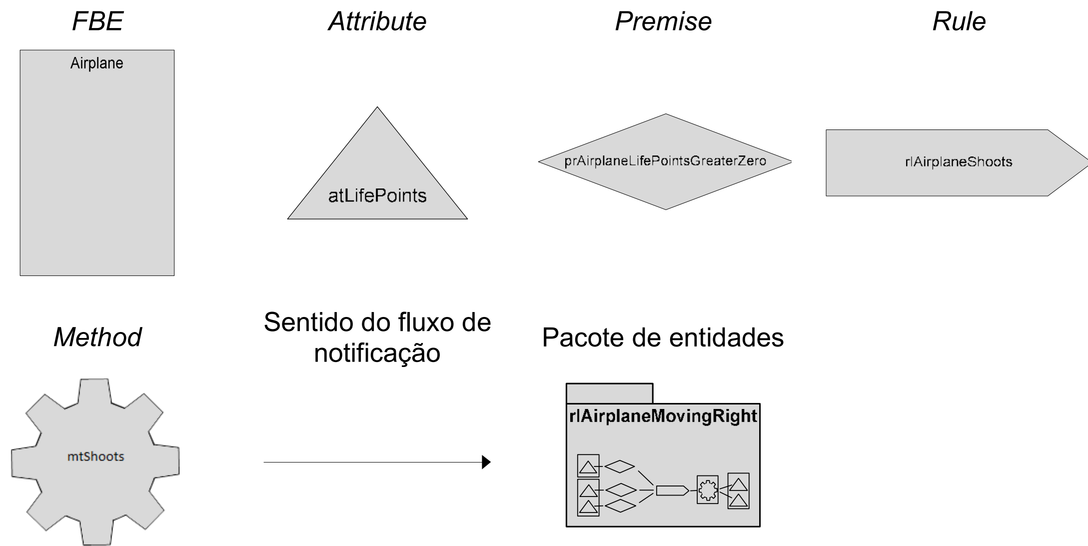


Figura 27 – Elementos do novo diagrama de objetos PON

O “Pacote de entidades” é um agrupamento de elementos que representa várias entidades que participam da composição de determinada entidade. Por exemplo, o pacote de entidades da *Rule* rlAirplaneMovingRight abriga todas as entidades que estão relacionadas com esta *Rule* como *Attributes*, *Premisses*, *Conditions* e inclusive *Methods*, conforme apresenta a Figura 28. Essa notação é utilizada para simplificar a representação das entidades no diagrama de objetos PON.

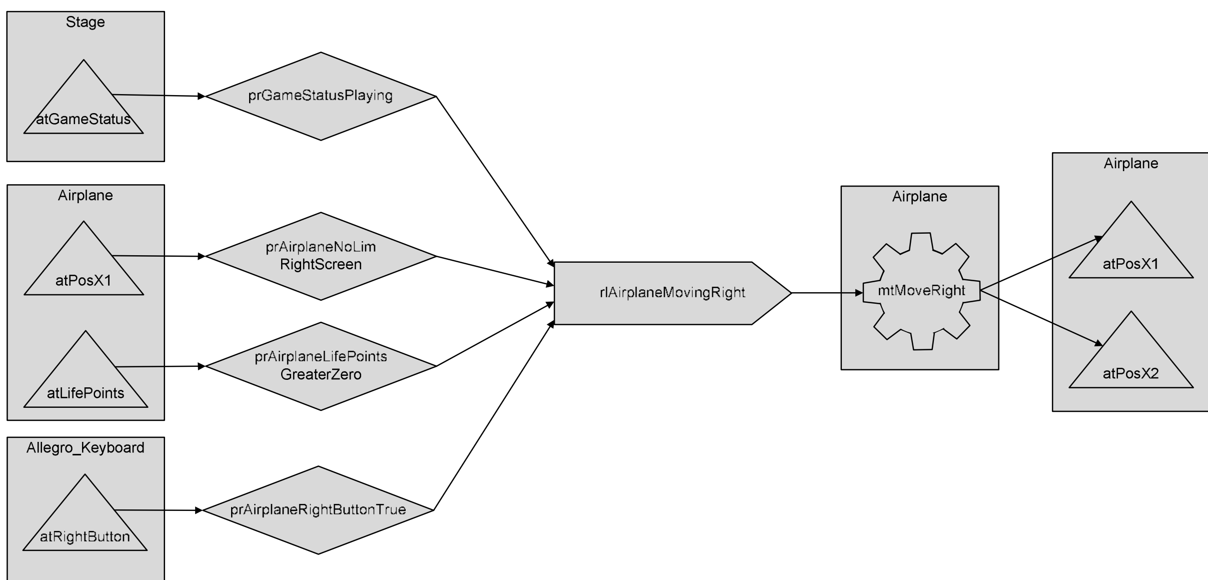


Figura 28 – *Rule* rlAirplaneMovingRight, suas entidades e relacionamentos representados com o diagrama de objetos PON

A Figura 28 mostra a *Rule* rlAirplaneMovingRight, seus elementos colaboradores e relacionamentos por meio do diagrama de objetos. É possível observar que as *FBEs* envolvidas para realizar a *Rule* são Stage, Airplane e Allegro Keyboard. As *Premisses* são prGamePlaying, prAirplaneLifePointsGreaterZero, prAirplaneNoLimRightScreen e prAirplaneRightButtonTrue. Quando a *Rule* for aprovada e pronta para executar, o único *Method* que poderá ser executado é mtMoveRight. Por sua vez, quando esse *Method* executar, modificará valores dos *Attributes* atPosX1 e atPosX2 do *FBE* Airplane. Todos os outros diagramas de objetos são apresentados no Capítulo 5.

3.3 CONSIDERAÇÕES SOBRE O DESENVOLVIMENTO EM PON

Este capítulo apresentou a modelagem de um jogo de combate aéreo em 2D desenvolvido utilizando o *framework* PON Otimizado e biblioteca gráfica Allegro. Esse software possibilitou compreender melhor como ocorre o desenvolvimento de software em PON e foi utilizado para apresentar a aplicabilidade do método de teste de software proposto para o PON que será apresentado no Capítulo 4.

Foram seguidas algumas etapas de desenvolvimento propostas pelo DON. Entretanto, foi necessário fazer algumas adaptações. A etapa de levantamento de *Rules* não utilizou diagrama de estados de alto nível proposto por Wiecheteck (2011). Para isso, foi realizada uma análise sobre os requisitos e casos de uso levantados, abordagem que se apresentou eficaz para o levantamento de *Rules* para esse software.

Também, não foram desenvolvidos os diagramas de comunicação e redes de Petri. Segundo Wiecheteck (2011) o diagrama de comunicação é baseado no diagrama de comunicação da UML e apresenta, basicamente, as mesmas informações do diagrama de sequência, mas com enfoque diferente, uma vez que não se preocupa com a cronologia do processo, porém o seu desenvolvimento não traria contribuições adicionais para esta dissertação. Por sua vez, o diagrama de redes de Petri ainda não possui um método completamente apropriado para que possa representar o PON adequadamente em qualquer tipo de software. Ademais,

foi proposto um novo diagrama de objetos que facilita a representação dos elementos do PON e o ciclo de notificações que ocorre entre eles.

4 PROPOSTA DE UM MÉTODO DE TESTE DE SOFTWARE PARA O PON

Este capítulo propõe um método de teste de software para o PON envolvendo as fases de teste unitário e teste de integração. Para isso, foi considerada a experiência do autor e da equipe de pesquisa no desenvolvimento de software em PON, os conceitos clássicos de teste de software e as peculiaridades do paradigma PON. Assim, a Seção 4.1, apresenta brevemente uma descrição sobre o método de teste de software proposto. A Seção 4.2 apresenta a fase de teste unitário. A Seção 4.3 apresenta a fase de teste de integração. Concluindo esse capítulo, a Seção 4.4 apresenta as considerações sobre o método de teste proposto.

4.1 APRESENTAÇÃO DO MÉTODO PROPOSTO

O método de teste de software proposto para o PON aplica-se nas fases de teste unitário e teste de integração. Basicamente, o teste unitário está focado no teste individual de cada entidade testável de uma aplicação PON. O teste de integração, por sua vez, considera todas as entidades do PON em conjunto e visa avaliar os fluxos de execução do software. A Figura 29 apresenta uma visão geral das fases de teste de software propostas para o PON. Para isso foi utilizada a notação SADT (do acrônimo em inglês *Structured Analysis and Design Technique)* (ROSS, 1978).

A fase de teste unitário envolve um “Planejamento e geração de testes unitários” para cada entidade que será testada (identificadas como Definição das entidades PON) e a “Execução dos testes unitários e avaliação dos resultados” que requer o “Plano de testes unitários e casos de teste” e o “Código fonte da aplicação PON”. Ao longo da execução dos testes unitários das entidades do software PON, deve-se registrar o “Resultado da execução dos testes unitários e avaliar os resultados”.

A fase de teste de integração é realizada após o teste unitário e envolve o “Planejamento e geração de testes de integração” a partir da “Definição das entidades PON” e “Modelos do DON”. A “Execução de teste de integração” requer o

"Plano de testes de integração e casos de teste" e o "Código fonte da aplicação PON" contendo todas as entidades que serão integradas.

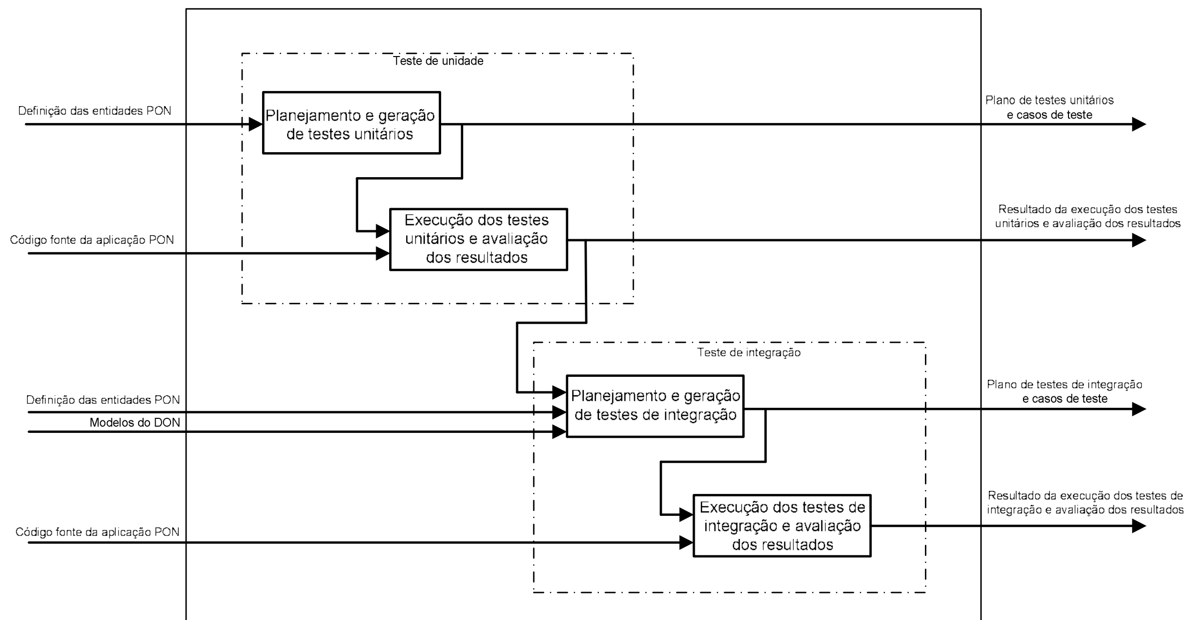


Figura 29 – Fases e atividades do método de teste de software em PON

Para a proposição dos testes unitários e de integração, foi estudado o paradigma PON e os trabalhos clássicos e mais conhecidos na área de teste de software. Foram adaptados alguns critérios clássicos de teste funcional (caixa-preta) e estrutural (caixa-branca). Com base nesses critérios, foram estudadas maneiras de produzir os planos de teste e casos de teste visando garantir a qualidade de cada unidade da aplicação PON individualmente (teste unitário) e do conjunto das unidades (teste de integração).

Nesse capítulo, os critérios de teste foram adaptados e exemplificados com algumas entidades PON e trechos do software apresentado no Capítulo 3. Por sua vez, o Capítulo 5 (estudo de caso) apresenta a aplicação desse método de teste sobre toda a aplicação desenvolvida.

Isso considerado, as subseções a seguir, apresentam o conjunto de critérios de teste de software propostos para as fases de teste unitário e teste de integração.

4.2 TESTE UNITÁRIO EM PON

Esta seção apresenta o teste unitário proposto para o PON. A Seção 4.2.1 apresenta uma visão geral da fase de teste unitário, a Seção 4.2.2 apresenta o

planejamento e geração de testes unitários em PON, a Seção 4.2.3 apresenta a análise dos resultados e a Seção 4.2.4 apresenta as considerações sobre o método de teste unitário apresentado.

4.2.1 Visão Geral do Teste Unitário

O teste unitário considera a menor unidade ou trecho de código que possa ser testado ou exercitado em um programa. Para o teste unitário de aplicações PON, foi considerado que cada entidade é uma unidade. Considerando que cada tipo de entidade PON possui um propósito e estrutura distintos, elaborou-se estratégias de teste particulares para *Premisses*, *Conditions, SubConditions*, *Rules* e *Methods* de *FBEs.*

No que diz respeito aos *Attributes* de um *FBE* e as *Actions* e *Instigations* das *Rules*, eles não precisam sofrer testes unitários. Os *Attributes* representam apenas declarações nos programas PON (com nome e tipo). Por sua vez, *Actions* e *Instigations* apresentam sempre o mesmo comportamento garantido por definição, i.e. cada *Action* tem por função chamar *Instigations*, cada *Instigation* tem por função chamar um *Method* (ou mais *Methods*) podendo parametrizá-los (teoricamente), assim, não oferecem a possibilidade de serem exercitados isoladamente em casos de teste.

A execução dos testes unitários das demais entidades PON é baseada no cumprimento do plano de teste. Essa visão da execução adota a abordagem típica usada em teste de software convencional e não foi alvo de proposição de novas abordagens no caso do PON.

A Figura 30 apresenta a visão expandida da fase de teste unitário que inclui as atividades de planejamento e geração de casos de teste, execução dos casos de teste e análise dos resultados. Essa figura detalha a fase de teste unitário apresentado na Figura 29, decompondo as atividades de Planejamento e de Execução da fase de teste de unitário.

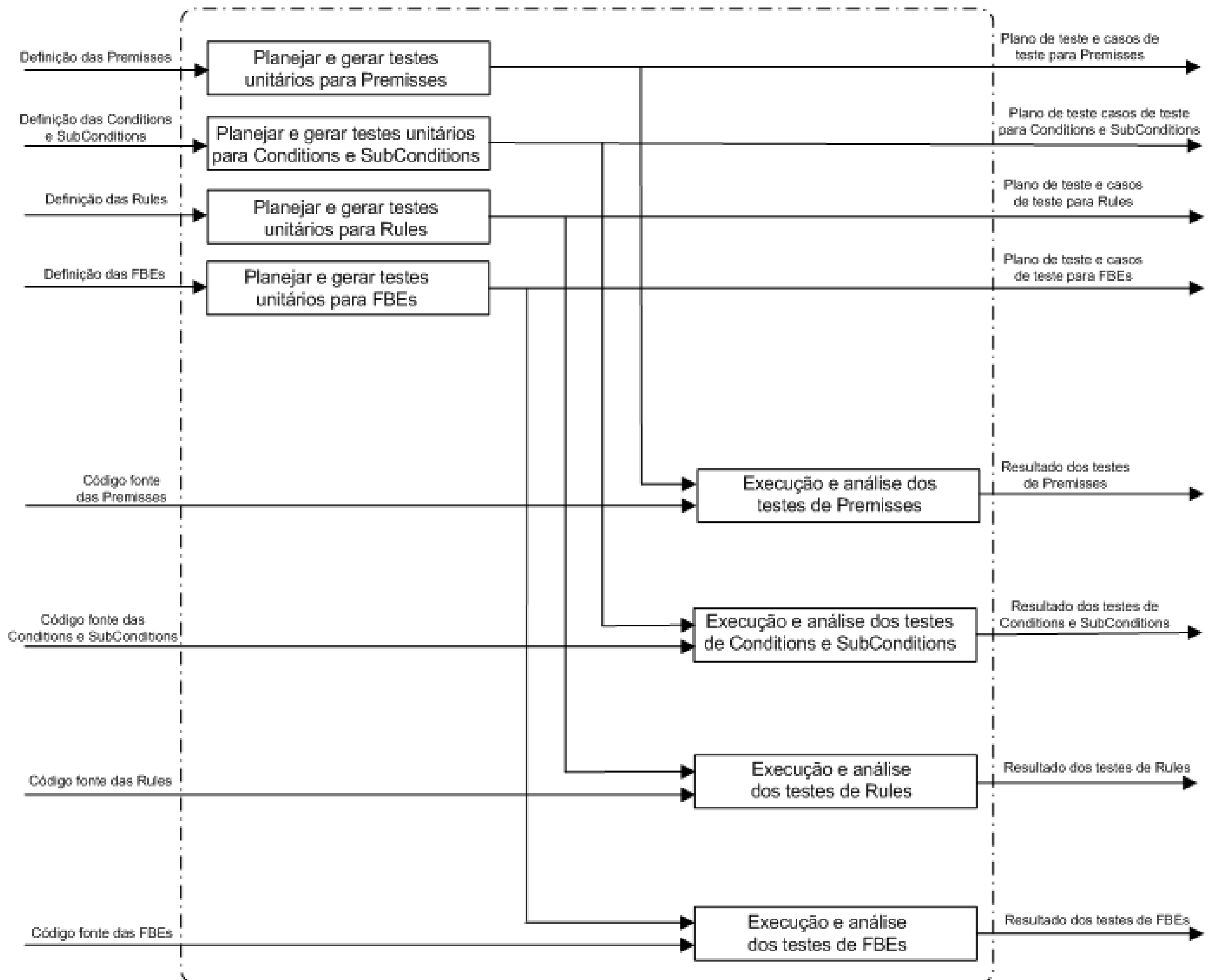


Figura 30 – Visão expandida da fase de teste unitário

As quatro atividades de planejamento e geração de casos de testes unitários são: (1) Planejar e gerar testes unitários para *Premisses*, (2) Planejar e gerar testes unitários para *Conditions* e *SubConditions*, (3) Planejar e gerar testes unitários para *Rules* e (4) Planejar e gerar testes unitários para *FBEs*). Essas quatro atividades têm como dados de entrada definições (que se dão por meio de documentação). Tais definições são no tocante as unidades cujos testes serão planejados e, a seguir, executados. Cada uma dessas atividades produz como saída o Plano de teste unitário e casos de teste do tipo de entidade PON respectiva.

A execução dos casos de teste e análise dos resultados requer o Código fonte de cada entidade a ser testada e seu respectivo Plano de teste e Casos de teste. Em geral, a execução dos casos de teste permite a avaliação dos resultados obtidos em relação aos resultados esperados.

Como indica a Figura 30, não há uma ordem obrigatória para elaboração dos planos e execução dos testes unitários dos quatro tipos de entidades PON. Mas a

disposição que aparece no diagrama sugere uma ordem que é pertinente, pois realizando-se o planejamento e execução dos testes unitários nesta ordem segue-se a ordem cronológica da cadeia de notificação do PON.

4.2.2 Abordagens de Testes Unitários

Esta seção apresenta o documento de plano de teste e detalha as abordagens de testes testes unitários para o PON. São apresentadas particularidades do teste de *Premise*, *Condition e SubCondition*, *Rule* e *Method* de *FBE*.

4.2.2.1 Documento de plano de teste unitário

Um plano de teste é um documento que descreve como conduzir os testes sobre todo ou parte de um software. Essencialmente, um plano de teste contém uma parte de definição dos testes a serem conduzidos e uma parte que descreve a lista dos casos de teste contemplados.

A parte de definição de um plano de teste contém informações que identificam o plano e os itens de software a serem testados e inclui, ainda, um conjunto de informações relacionadas às atividades envolvidas, critérios de aceite, abordagens e ferramentas a serem utilizadas (IEEE, 1998). A Tabela 12 ilustra um exemplo reduzido da parte de definição de um plano de teste. Definições mais completas podem ser desenvolvidas conforme a norma IEEE 829 (1998).

Tabela 12 – Plano de teste para uma *Premise*

Identificador do Plano de Teste test_prAirplaneLimRightScreen01
Descrição Este documento do plano de testes para *Premise* lista os Requisitos que serão testados, bem como recomenda e descreve as estratégias a serem empregadas nesses testes.
Referências Documento de Levantamento de Requisitos. Documento de Levantamento de Regras (o qual inclui a definição de *Premisses*).
Itens testáveis Deverá ser testado o comportamento da *Premise* com base na alteração de valor do (s) *Attribute* (s) envolvido (s). Determinar classes de equivalência e valores limite que exercitem os estados da *Premise*. Tipo: Teste funcional (caixa-preta). Particionamento em classes de equivalência. Análise de valor limite. Versão da *Premise*: 1_0. *FBE*: Airplane.
Abordagem Visual Studio 2010, *framework* PON Otimizado, código fonte da aplicação Teste de unidade Necessário código fonte da aplicação

A parte de descrição dos casos de teste contém uma lista com os casos de teste que compõem o plano e seu detalhamento. Cada caso de teste representa uma situação ou condição a ser testada para o item de software indicado no plano e suas pré-condições, passos e pós-condições. A Tabela 13 ilustra um exemplo de descrição de um caso de teste resumido baseado no modelo IEEE 829 (1998).

Tabela 13 – Exemplo de descrição de caso de teste no plano de teste

ID	test_metodo01	
Propósito	Verificar o resultado da execução do método	
Pré-requisitos	Visual Studio 2010 configurado com *framework* PON Otimizado e código fonte da aplicação	
Passos	**Ação / necessidade**	**Resultado esperado**
1	Executar a aplicação PON	Aplicação PON em execução
2	Inserir *breakpoint* na linha 45 do arquivo modulo.cpp	*Breakpoint* inserido e software interrompido no *breakpoint*
3	Verificar se o valor da variável mod é igual a 200	Valor de mod igual a 200
4	Pressionar F5 para que a aplicação continue normalmente.	Aplicação PON em execução

Nas próximas seções, serão apresentadas abordagens e/ou estratégias de planejamento de teste. Será ilustrada apenas a parte de geração de casos de teste visando ser sucinto na extensão do texto desta dissertação.

4.2.2.2 Estratégia de geração de casos de teste para Premisses

Uma *Premise* é uma entidade do PON que realiza uma expressão lógica do tipo *se-então*. Essa expressão pode envolver dois *Attributes* ou um *Attribute* e uma constante, que são avaliados por operadores lógicos. Atualmente, os tipos de operadores lógicos empregados no PON são: *Equal, Different, Greater or Equal, Smaller or Equal, Greater Than* e *Smaller Than*.

É necessário conhecer a estrutura de uma *Premise* para que possa ser realizado o planejamento e geração de casos de teste. Para a *Premise* que avalia um *Attribute* e uma constante, deve-se considerar o tipo de dado do *Attribute*, o operador lógico que está sendo utilizado e o valor da constante que está sendo avaliado. Por exemplo, a Tabela 14 apresenta a estrutura da *Premise* prAirplaneLimRightScreen. Nota-se que o tipo do *Attribute* atPosX2 é inteiro, o operador é *Smaller or Equal* e o valor da constante é 800.

Tabela 14 – Estrutura da *Premise* prAirplaneLimRightScreen

Premise – prAirplaneLimRightScreen		
FBE	Airplane	
Attribute	Integer	atPosX2
Operador	*Smaller or Equal*	
Constante	800	

Com estas informações, é possível planejar casos de teste por meio da determinação de classes de equivalência de valores válidos para o *Attribute* avaliado, que são um valor ou uma faixa de valores que possibilitam a aprovação de uma *Premise*. Por outro lado, também é possível determinar as classes de equivalência de valores inválidos que são um valor ou uma faixa de valores que não possibilitam a aprovação de uma *Premise*. Em conjunto com a determinação de classes de equivalência, é possível também determinar os valores limite que estão nas bordas da condição para que a *Premise* possa ou não ser aprovada.

Com base na definição de teste da *Premise* prAirplaneLimRightScreen, a determinação de classes de equivalência define uma classe de valores válidos (valores inteiros menores ou iguais à 800) e uma classe de valores inválidos (valores inteiros superiores à 800). Por sua vez, a análise de valor limite estabelece o maior

valor de borda válido que é 800 e o menor valor de borda inválido que é 801 (menor valor inteiro positivo mais próximo de 800, não suportado na condição lógica do operador), conforme apresentado na Figura 31.

					Borda esquerda	Borda direita				
...	796	797	798	799	**800**	801	802	803	804	...
Classe de equivalência Valores válidos						Classe de equivalência Valores inválidos				

Figura 31 – Classes de equivalência e análise de valores limite para prAirplaneLimRightScreen

A Tabela 15 apresenta a determinação de classes de equivalência e análise de valor limite para a *Premise* prAirplaneLimRightScreen.

Tabela 15 – Classes de equivalência e análise de valor limite para a *Premise* prAirplaneLimRightScreen

***Premise* –** prAirplaneLimRightScreen		
FBE	Airplane	
Attribute	**Integer**	atPosX2
Operador	*Smaller or Equal*	
Constante	800	

Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
atPosX2<=800	atPosX2>800	800	801

A partir da definição do plano de teste, determinação de classes de equivalência e análise de valor limite é possível gerar casos de teste para esta *Premise,* conforme apresentada a Tabela 16.

Tabela 16 – Exemplo de descrição de caso de teste para prAirplaneLimRightScreen

Pré-condições	Visual Studio 2010, *framework* PON Otimizado, código fonte da aplicação	
Passo	**Ação / necessidade**	**Resultado Esperado**
1	Executar a aplicação PON	Aplicação PON em execução
2	Inserir *breakpoint* na primeira linha do método moveRight do *FBE* Airplane.	*Breakpoint* inserido
3	Inserir *breakpoint* na linha em que está localizada a *Premise* prAirplaneLimRightScreen	*Breakpoint* inserido
4	Pressionar o botão de Seta para a Direita do teclado	Software interrompido no *breakpoint* do *FBE* Airplane (inserido no Passo 2)
5	Verificar se é criado um objeto de Airplane que é requerido pela *Premise*	Objeto Airplane criado
6	Definir valores para o *Attribute* atPosX2 (classes de equivalência, análise de valor limite)	Valor de AtPosX2 alterado para 800
7	Pressionar F5 para que a aplicação continue normalmente.	Software interrompido no *breakpoint* da *Premise* (inserido no Passo 3) indicando que a premissa foi aprovada.
Pós-condições	Verificar o resultado da execução do caso de teste	

A Tabela 17 apresenta 4 casos de teste gerados para a *Premise* prAirplaneLimRightScreen. Os casos de teste 1 e 2 representam os casos sobre os valores limite. Os casos de teste 3 e 4 representam casos para os valores escolhidos abaixo e acima dos valores limite. Como observado, os valores limites são casos de teste que exercitam as duas classes de equivalência identificadas. Assim sendo, optou-se por escolher mais dois valores para exercitar estas classes para fins de exemplificação. Naturalmente, podem ser escolhidos quaisquer outros valores possíveis nas classes de equivalência, porém, apresentarão o mesmo comportamento de outros valores equivalentes.

Tabela 17 – Casos de teste para a *Premise* prAirplaneLimRightScreen

Casos de teste	**Valor de atPosX2**	**Saída esperada ou comportamento esperado**
1	800	Aprova a *Premise*
2	801	Não aprova a *Premise*
3	799	Aprova a *Premise*
4	802	Não aprova a *Premise*

Para uma *Premise* que possui operador lógico de *Greater, Smaller, Equal* ou *Different*, também podem ser gerados, no mínimo, dois casos de teste para exercitar a *Premise,* seguindo esse mesmo critério.

Por sua vez, em uma *Premise* que avalia dois *Attributes*, ocorre uma modificação na determinação de classes de equivalência e valores limite. São considerados em conjunto os dois *Attributes* e o operador lógico. Por exemplo, a Tabela 18 apresenta a determinação de classes de equivalência e análise de valor limite para a *Premise* prBulletXGreaterX1Helicopter. Nesta *Premise* são avaliados dois *Attributes* de *FBEs* diferentes (Bullet e Airplane) e o operador lógico é “Greater or Equal*”*.

Tabela 18 – Classes de equivalência e análise de valor limite para a *Premise* prBulletXGreaterX1Helicopter

Premise	prBulletXGreaterX1Helicopter		
FBE	Bullet		
Attribute	**Integer**	atPosX	
FBE	Airplane		
Attribute	**Integer**	atPosX1	
Operador	*Greater or Equal*		
Valor	atPosX>=atPosX1		
Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
atPosX>=atPosX1	atPosX<atPosX1	Refere-se ao valor de atPosX, número igual a atPosX1	Refere-se ao valor de atPosX, primeiro número inteiro menor que atPosX1

Para exercitar uma *Premise* que avalia dois *Attributes* deve ser gerado no mínimo três casos de teste com: os dois *Attributes* com valores iguais, o valor do primeiro *Attribute* maior que o segundo e o valor do segundo *Attribute* maior que o primeiro.

A Tabela 19 apresenta a geração de 3 casos de teste unitário para a *Premise* prBulletXGreaterX1Helicopter. Como os valores que serão utilizados pela aplicação do teste unitário não são conhecidos para exercitar esta *Premise* (i.e., o domínio dos valores dos *Attributes*), foram determinados valores inteiros quaisquer para os dois *Attributes*.

Tabela 19 – Casos de teste para a *Premise* prBulletXGreaterX1Helicopter

Casos de teste	Valor atPosX	Valor atPosX1	Saída esperada ou comportamento esperado
1	1	1	Aprova a *Premise*
2	1	2	Não aprova a *Premise*
3	2	1	Aprova a *Premise*

É possível agrupar as *Premisses* por características semelhantes como tipo de operador lógico e quantidade de *Attributes* avaliados. A partir disto, considera-se que o planejamento e geração de casos de teste para cada entidade pode ser repetido para as demais do mesmo grupo, atentando-se apenas a particularidades como o valor considerado na avaliação do *Attribute*. Por exemplo, se existem duas *Premisses* que avaliam um *Attribute* (não necessariamente o mesmo) do tipo inteiro e o operador lógico é "Greater or Equal", ambas podem ser testadas da mesma maneira, apenas atentando-se ao valor da constante considerada.

Podem existir *Premisses* que avaliam *Attributes* impertinentes e *Premisses* configuradas como exclusivas. De acordo com Ronszcka (2012), um *Attribute* pode ser definido como impertinente quando apresenta alterações constantes em seu estado, que no cenário em questão raramente impactariam na aprovação de sua *Condition*. De acordo com Banazewski (2009), uma *Premise* pode ser definida como exclusiva quando não precisar notificar todas as *Conditions* de sua lista, bastando que as notificações ocorram até que uma das *Conditions* seja satisfeita.

Premisses que avaliam *Attributes* configurados como impertinentes são testadas da mesma maneira que se testa *Premisses* que avaliam *Attributes* sem esta *flag*. O mesmo ocorre com *Premisses* configuradas como exclusivas.

4.2.2.3 Estratégia de geração de casos de teste para Conditions e SubConditions

Uma *Condition* é uma entidade do PON que avalia uma ou mais *SubConditions* ou uma ou mais *Premisses* por meio de uma expressão condicional de conjunção *(i.e., "*and*")* ou disjunção *(i.e., "*or*")*. Uma *SubCondition* pode avaliar *Premisses, também,* por meio de uma expressão condicional do tipo conjunção ou disjunção. Para que uma *SubCondition* que utiliza operador de conjunção seja aprovada, requer-se que todas suas *Premisses* sejam aprovadas. Para que uma *SubCondition* que utilize operador de disjunção seja aprovada, basta que uma de suas *Premisses* seja aprovada.

Para que uma *Condition* que utiliza operador de conjunção seja aprovada, requer-se que todas suas *Premisses* ou *SubConditions* sejam aprovadas. Por sua vez, para que uma *Condition* que utilize um operador de disjunção seja aprovada, requer-se que pelo menos uma *Premise* ou *SubCondition* seja aprovada.

Para exemplificar a geração de casos de teste, a Tabela 20 apresenta a estrutura de uma *Condition* que avalia uma conjunção de quatro *Premisses*. Foram definidos identificadores para cada *Premise* (pr1, pr2, pr3 e pr4) a fim de simplificar sua apresentação na tabela de casos de teste.

Tabela 20 – Estrutura de uma *Condition* que avalia quatro *Premisses*

***Condition* da *Rule* rlAirplaneMovingRight**	
Operador	***Conjunção***
Premisses	Pr1 **–** prGameStatusPlaying
	Pr2 **–** prAirplaneLifePointsGreaterZero
	Pr3 **–** prAirplaneRightButtonTrue
	Pr4 **–** prAirplaneLimRightScreen

Para esta *Condition*, a Tabela 21 apresenta um exemplo de Caso de Teste.

Tabela 21 – Caso de teste para a *Condition* da *Rule* rlAirplaneMovingRight

Pré-condições	Visual Studio 2010, *framework* PON Otimizado, código fonte da aplicação	
Passo	**Ação / necessidade**	**Resultado Esperado**
1	Executar a aplicação PON	Aplicação PON em execução
2	Inserir *breakpoint* na primeira linha da definição da *Rule* rlAirplaneMovingRight do *FBE* Airplane.	*Breakpoint* inserido
3	Pressionar o botão de Seta para a Direita do teclado	Software interrompido no *breakpoint* do *FBE* Airplane (inserido no Passo 2)
4	Verificar se a *Condition* é aprovada	*Condition* aprovada
5	Pressionar F5 para que a aplicação continue normalmente.	Software interrompido no *breakpoint* da *Premise* (inserido no Passo 2) indicando que a *Condition* foi aprovada.
Pós-condições	Verificar o resultado da execução do caso de teste	

A Tabela 22 apresenta os casos de teste para esta *Condition*. Como a *Condition* avalia *Premisses* que podem ter apenas dois estados diferentes (aprovada ou não aprovada) é possível aplicar a fórmula da tabela verdade 2^n (CAPUANO; IDOETA, 2000) para descobrir o número máximo de casos de teste possíveis. Com isso, $n = 4$ (número de *Premisses*) resulta em 16 casos de teste possíveis para esta *Condition*.

Tabela 22 – Casos de teste para a *Condition* da *Rule* rlAirplaneMovingRight

Casos de teste	Estado Pr1	Estado Pr2	Estado Pr3	Estado Pr4	Saída esperada ou comportamento esperado
1	Não aprovada	Não aprovada	Não aprovada	Não aprovada	Não aprova *Condition*
2	Não aprovada	Não aprovada	Não aprovada	Aprovada	Não aprova *Condition*
3	Não aprovada	Não aprovada	Aprovada	Não aprovada	Não aprova *Condition*
4	Não aprovada	Não aprovada	Aprovada	Aprovada	Não aprova *Condition*
5	Não aprovada	Aprovada	Não aprovada	Não aprovada	Não aprova *Condition*
6	Não aprovada	Aprovada	Não aprovada	Aprovada	Não aprova *Condition*
7	Não aprovada	Aprovada	Aprovada	Não aprovada	Não aprova *Condition*
8	Não aprovada	Aprovada	Aprovada	Aprovada	Não aprova *Condition*
9	Aprovada	Não aprovada	Não aprovada	Não aprovada	Não aprova *Condition*
10	Aprovada	Não aprovada	Não aprovada	Aprovada	Não aprova *Condition*
11	Aprovada	Não aprovada	Aprovada	Não aprovada	Não aprova *Condition*
12	Aprovada	Não aprovada	Aprovada	Aprovada	Não aprova *Condition*
13	Aprovada	Aprovada	Não aprovada	Não aprovada	Não aprova *Condition*
14	Aprovada	Aprovada	Não aprovada	Aprovada	Não aprova *Condition*
15	Aprovada	Aprovada	Aprovada	Não aprovada	Não aprova *Condition*
16	Aprovada	Aprovada	Aprovada	Aprovada	Aprova *Condition*

Para exemplificar uma *Condition* que tem um operador de disjunção, considere a estrutura da *Condition* apresentada na Tabela 23.

Tabela 23 – *Condition* que avalia uma disjunção de duas *Premisses*

Condition da Rule rlAllegroBlit	
Operador	***Disjunção***
Premisses	Pr1 – prGameStatusPlaying
	Pr2 – prGameStatusPaused

A Tabela 24 apresenta os casos de teste para esta *Condition*. Nota-se que o operador de disjunção requer que pelo menos uma *Premise* esteja aprovada para que sua *Condition* seja aprovada.

Tabela 24 – Casos de teste para a *Condition* da *Rule* rlAllegroBlit que avalia uma disjunção de duas *Premisses*

Casos de teste	Estado prGameStatusPlaying	Estado prGameStatusPaused	Saída esperada ou comportamento esperado
1	Aprovada	Aprovada	Aprova a *Condition*
2	Aprovada	Não aprovada	Aprova a *Condition*
3	Não aprovada	Aprovada	Aprova a *Condition*
4	Não aprovada	Não aprovada	Não aprova a *Condition*

Por sua vez, para exemplificar uma *Condition* que avalia uma conjunção de duas *SubConditions*, considere a estrutura da *Condition* apresentada na Tabela 25.

Tabela 25 – *Condition* que avalia conjunção de duas *SubConditions*

<table>
<tr><td colspan="3">Condition da Rule rlAllegroDrawAirplane</td></tr>
<tr><td>Operador</td><td colspan="2">Conjunção</td></tr>
<tr><td rowspan="3">SubConditions</td><td rowspan="2">SubCondition 1 – Operador disjunção</td><td>prGameStatusPlaying</td></tr>
<tr><td>prGameStatusPaused</td></tr>
<tr><td>SubCondition 2 – Operador conjunção</td><td>prAirplaneLifePointsGreaterZero</td></tr>
</table>

A Tabela 26 apresenta os casos de teste para a *SubCondition* 1. A Tabela 27 apresenta a execução de casos de teste para a *SubCondition* 2.

Tabela 26 – Casos de teste para a *SubCondition 1*

Casos de teste	Estado prGameStatusPlaying	Estado prGameStatusPaused	Saída esperada ou comportamento esperado
1	Aprovada	Aprovada	Aprova a *SubCondition* 1
2	Aprovada	Não aprovada	Aprova a *SubCondition* 1
3	Não aprovada	Aprovada	Aprova a *SubCondition* 1
4	Não aprovada	Não aprovada	Não aprova a *SubCondition* 1

Tabela 27 – Casos de teste para a *SubCondition 2*

Casos de teste	Estado prAirplaneLifePointsGreaterZero	Saída esperada ou comportamento esperado
1	Aprovada	Aprova a *SubCondition* 2
2	Não aprovada	Não aprova a *SubCondition* 2

A Tabela 28 apresenta os casos de teste para a *Condition* que avalia as *SubConditions* 1 e 2 por meio do operador de conjunção.

Tabela 28 – Casos de teste para uma a *Condition* que avalia duas *SubConditions*

Casos de teste	Estado *SubCondition* 1	Estado *SubCondition* 2	Saída esperada ou comportamento esperado
1	Aprovada	Aprovada	Aprova a *Condition*
2	Aprovada	Não aprovada	Não aprova a *Condition*
3	Não aprovada	Aprovada	Não aprova a *Condition*
4	Não aprovada	Não aprovada	Não aprova a *Condition*

4.2.2.3.1 Geração de casos de teste unitário para Condition que avalia Premise impertinente e Premise exclusiva

Conditions e *SubConditions* também podem avaliar *Premisses Exclusivas* e *Premisses* que avaliam *Attributes* definidos como *Impertinentes*. O planejamento de

casos de teste para *Conditions* e *SubConditions* que avaliam *Premisses* impertinentes sofre modificação em relação aos casos em que elas apenas avaliam *Premisses* sem esta característica. A principal diferença é que nem sempre será avaliada a *Premise* com *Attribute* impertinente, i.e., será avaliada apenas quando as outras *Premisses* da (mesma) *Condition* tiverem sido aprovadas.

Por exemplo, a Tabela 29 apresenta a estrutura de uma *Condition* que avalia apenas duas *Premisses*, sendo que uma delas é *Premise* normal (prGameStatusPlaying) e outra é uma *Premise* configurada como impertinente (prImpertinent – nome fictício para exemplificar, não foi implementada *Premise* com *Attribute* impertinente na aplicação).

Tabela 29 – *Condition* que avalia uma *Premise* com *Attribute* impertinente

Condition que avalia Premise impertinente	
Operador	Conjunção
Premisses	Pr1 **–** prGameStatusPlaying
	Pr2 **–** prImpertinent

A Tabela 30 apresenta os casos de teste para esta *Condition*. Para os três casos de teste, nota-se que a avaliação da *Premise* prImpertinent apenas ocorrerá quando a outra *Premise* estiver aprovada. Assim, se prGameStatusPlaying for aprovada, prImpertinent poderá ser avaliada e, caso seja aprovada, a *Condition* será então aprovada.

Tabela 30 – Execução de casos de teste para uma *Condition* que avalia uma *Premise* com *Attribute* impertinente

Casos de teste	**Estado prGameStatusPlaying**	**Estado prImpertinent**	**Saída esperada ou comportamento esperado**
1	Aprovada	Aprovada	Aprova a *Condition*
2	Aprovada	Não aprovada	Não aprova a *Condition*
3	Não aprovada	(Não avaliada)	Não aprova a *Condition*

A *Premise* configurada como exclusiva não sofre modificações em relação ao planejamento de casos de teste unitário em *Conditions* e *SubConditions*.

4.2.2.4 Estratégia de geração de casos de teste para Rules

Rule avalia apenas o estado de uma *Condition*. Para que uma *Rule* seja aprovada requer-se, obrigatoriamente, que sua *Condition* seja aprovada. Assim, para testar uma *Rule*, deve-se avaliar as situações que permitem ou não aprovar sua *Condition*. A Tabela 31 apresenta a *Rule* rlAirplaneMovingRight e sua única *Condition* que avalia a conjunção de quatro *Premisses*.

Tabela 31 – *Rule* que avalia apenas uma *Condition*

***Rule* –** rlAirplaneMovingRight	
Condition	*Operador de Conjunção*
Premisses	Pr1 **–** prGameStatusPlaying
	Pr2 **–** prAirplaneLifePointsGreaterZero
	Pr3 **–** prAirplaneRightButtonTrue
	Pr4 **–** prAirplaneLimRightScreen

A Tabela 32 apresenta um exemplo de caso de teste para esta *Rule*.

Tabela 32 – Caso de teste para a *Rule* rlAirplaneMovingRight

Pré-condições	Visual Studio 2010, *framework* PON Otimizado, código fonte da aplicação	
Passo	**Ação / necessidade**	**Resultado Esperado**
1	Executar a aplicação PON	Aplicação PON em execução
2	Inserir *breakpoint* na primeira linha da definição da *Rule* rlAirplaneMovingRight do *FBE* Airplane.	*Breakpoint* inserido
3	Pressionar o botão de Seta para a Direita do teclado	Software interrompido no *breakpoint* do *FBE* Airplane (inserido no Passo 2)
4	Verificar se a *Rule* é aprovada	*Rule* aprovada
5	Pressionar F5 para que a aplicação continue normalmente.	Aplicação PON em execução
Pós-condições	Verificar o resultado da execução do caso de teste	

A Tabela 33 apresenta os casos de teste para essa *Rule*. Nota-se que a *Rule* está apenas interessada no estado de sua *Condition*, ou seja, se está aprovada ou não. Para isto, precisaram ser instigadas as *Premisses* que compõem a *Condition*. O planejamento e execução de casos de teste para *Premisses*, *Conditions* e *SubConditions* foram apresentados nas seções 4.2.2.2 e 4.2.2.3, respectivamente.

Tabela 33 – Execução de casos de teste para a *Rule* rlAirplaneMovingRight

Casos de teste	Estado *Condition*	Saída esperada ou comportamento esperado
1	Aprovada	Aprova a *Rule*
2	Não aprovada	Não aprova a *Rule*

4.2.2.5 Estratégia de geração de casos de teste para FBEs

FBEs são implementados em PON na forma de classes similares em termos de abstração estrutural às do paradigma OO e podem conter atributos, métodos e possuir uma ou mais instâncias na forma de objetos. Deve-se considerar que os *FBEs* definidos para uma aplicação PON podem fazer parte de uma estrutura que inclui relacionamentos de especialização, agregação e associação entre as classes que os definem.

A Figura 19 (Seção 3.2.3, pg. 77) mostrou um diagrama de classes para a aplicação PON de exemplo apresentada nesta dissertação. Neste diagrama, observa-se a existência de relacionamentos de herança, agregação e associação entre *FBEs* visando organizar suas definições, de acordo a lógica da orientação a objetos. A estratégia adotada para teste de *FBEs* envolve selecionar, não todos os *FBEs*, mas somente os seguintes:

- *FBEs* desenvolvidos especificamente para a aplicação PON. Neste caso, excluem-se os *FBEs* importados ou produzidos por terceiros, pois considera-se que todo código adquirido ou reaproveitado, já foi submetido aos testes necessários. Por exemplo, o *FBE* Allegro Interface, que representa uma classe produzida por terceiros (Allegro), não será incluído nos testes daquela aplicação;
- Todos os *FBEs* derivados, quando houver relacionamentos de herança, serão testados. *FBEs* descritos por superclasses (i.e. classes base) não serão testados, pois seu comportamento será testado nas classes (*FBEs*) derivadas;
- Todos os *FBEs* representados por classes agregadas (ou seja, que incorporam objetos de outras classes por meio de agregações) serão testados;

- Todos os *FBEs* isolados ou apenas associados a outros *FBEs* serão testados.

FBEs podem conter, essencialmente, *Attributes* e *Methods,* e podem estar relacionados a outras classes ou fazerem uso de outros recursos da OO. Como dito na Seção 4.2.1, os *Attributes* dos *FBEs* não serão submetidos a testes unitários.

No que diz respeito aos *Methods* de um *FBE*, considera-se que eles se equiparam a um método de uma classe OO e apenas acrescentam a possibilidade de criação ou alteração de entidades PON (como *Attributes*, *Premisses*, *Rules*). Assim, é possível utilizar abordagens de planejamento e geração de testes análogos às existentes para teste unitário de métodos OO.

O Plano de Teste do *Method*, além da parte de definição, deverá informar a lista dos casos de teste unitários para o *Method*. No procedimento geral de teste de aplicações PON proposto nesta dissertação, os testes de *Methods* serão conduzidos na forma de testes funcionais (i.e., testes em caixa-preta).

Para a geração dos casos de teste unitários de um *Method* é necessário que se conheça quais são as variáveis que influenciam o comportamento deste *Method*. Estas variáveis podem ser de dois tipos:

- Parâmetros de entrada do *Method*

  Um *Method* pode ou não conter parâmetros de entrada (ou seja, de chamada). Os parâmetros de entrada são argumentos (valores) que são passados ao método na sua chamada pelas *Instigations*. O comportamento esperado de um *Method* é influenciado pelo valor dos parâmetros e guiarão a elaboração dos casos de teste.

- Estados de *Attributes*

  Attributes do próprio *FBE* que contém o *Method* a ser testado, ou de outros *FBEs*, são equivalentes para o *Method* a variáveis globais em programação convencional. Assim, um *Method* pode fazer referência e ter comportamentos variáveis em função do estado de *Attributes*. Como o comportamento esperado de um *Method* pode ser influenciado pelo valor de *Attributes*, eles também guiarão a elaboração dos casos de teste.

A partir dos parâmetros e dos *Attributes* referenciados, são gerados os casos de teste do *Method* por meio da escolha de valores pertencentes às classes de equivalência e análise de valor limite para valores válidos e inválidos. O uso de valores limite para execução dos casos de teste garante a cobertura dos principais pontos de vulnerabilidade, que são os valores de borda do *Method*.

Considera-se que um *Method* pode ter uma ou mais instruções que realizam operações lógicas que precisam ser exercitadas (v.g. “if-then”) e que podem consultar um ou mais *Attributes* resultando em comportamentos diferentes do método. Neste caso, o *Attribute* seria como um parâmetro de entrada e deveria ter classes de equivalência como os parâmetros.

Conforme visto na Seção 4.2.2.2, casos de teste que exercitem *Premisses* também podem utilizar classes de equivalência e valores limite. No entanto, *Methods* permitem o desenvolvimento de atividades mais complexas que *Premisses*. Por isto, a seguir, são apresentadas regras para determinação de classes de equivalência e análise de valores limite (que também podem ser úteis para gerar casos de teste para *Premisses* e para gerar casos de teste de integração, conforme será visto no próximo capítulo).

As regras para o levantamento de classes de equivalência apresentadas por Myers et al. (2004) podem ser utilizadas para exercitar parâmetros (como *Attributes)* em *Method* PON:

1) Se uma condição de entrada especifica um intervalo, pode-se definir uma classe de equivalência válida de duas classes de equivalência inválidas.
   Por exemplo: se o valor de posição do avião deve estar entre 1 e 800, identifica-se uma classe de equivalência válida (1 >= posição <= 800) e duas classes de equivalência inválidas (posição < 1 ou posição > 800).
2) Se uma condição de entrada especifica um conjunto de valores, pode-se definir uma classe de equivalência válida e duas classes de equivalência inválidas.
   Por exemplo: se existem três condições possíveis para o estado do jogo: 1 - pausado, 2 - jogando e 3 - parado (os três representam uma classe de equivalência de valores válidos), as duas classes de equivalência inválidas seriam valores iguais ou menores que zero, ou valores iguais ou superiores a 4, representando estados de jogo impossíveis.

3) Se uma condição de entrada especifica um membro de um conjunto, pode-se definir uma classe de equivalência válida e uma classe de equivalência inválida.
   Por exemplo: se um personagem deve ser do tipo avião ou helicóptero (classes válidas), então o tipo de personagem carro seria inválido.
4) Se uma condição de entrada especifica que determinada situação “deve ser de determinada maneira” pode-se definir uma classe de equivalência válida e uma inválida.
   Por exemplo: para decrementar os pontos de vida de um personagem é preciso que haja colisão com projétil. Então, essa condição representaria uma classe de equivalência válida. Enquanto a ausência dessa condição representaria uma classe de equivalência inválida.

Depois que as classes de equivalência tiverem sido identificadas, o próximo passo é projetar casos de teste com os seguintes passos:

1) Cada classe de equivalência deve ter um único identificador. Um valor inteiro é suficiente.
2) Devem ser realizados testes com todas as classes de equivalência válidas até que todas sejam cobertas (incluídas nos testes). Como informado, um dado caso de teste pode cobrir mais de uma classe de equivalência.
3) Devem ser realizados testes com todas as classes de equivalência inválidas até que todas sejam cobertas individualmente. Isto se faz para garantir que um caso de teste com classe de equivalência inválida não mascare o efeito de outros testes com outras classes de equivalência.

Enquanto que o particionamento em classes de equivalência direciona o testador a selecionar casos de teste em qualquer elemento da classe de equivalência, a análise de valor limite requer que o testador selecione elementos perto das fronteiras, em que valores maiores ou menores que os de fronteira sejam cobertos nos casos de teste. O procedimento descrito a seguir é útil para identificar os valores limite:

- Se uma condição de entrada para o sistema sob teste é especificada como um intervalo de valores, devem ser realizados testes com valores válidos para as extremidades do intervalo, e testes com valores inválidos para possibilidades ligeiramente acima e abaixo das extremidades do intervalo.
- Por exemplo, se uma especificação indica que um valor de entrada (*Attribute*) para um *Method* deve situar-se no intervalo entre 1 e 800, testes válidos que incluem valores para as extremidades do intervalo, assim como testes inválidos com valores imediatamente acima e abaixo das extremidades deve ser incluídos. Isto resultaria nos seguintes valores em casos de teste: 0, 1 e 800, 801.
- Se uma condição de entrada para um *Method* é especificada como um número de valores, devem ser realizados testes para os números máximos e mínimos, assim como testes com números inválidos que incluem um valor ligeiramente menor e um valor ligeiramente maior que os valores máximo e o mínimo.
- Por exemplo, se uma especificação de entrada para um módulo deve ser exatamente o número 2, testes que incluem o número 2 assim como os números 1 e o 3 precisam ser realizados.
- Se a entrada ou saída de um *Method* sob teste é um conjunto ordenado, como uma tabela ou uma lista linear, devem ser realizados testes que incidem sobre o primeiro e último elementos do conjunto.

Para gerar casos de teste que explorem a combinação de classes de equivalência e valores limite, uma tabela de decisão também pode ser utilizada. Conforme visto na Seção 2.4.3.1, por meio de uma tabela de decisão é possível expressar, em forma de tabela, qual o conjunto de condições necessárias para que um determinado conjunto de ações possam ser executadas (MYERS et al., 2004). O ponto principal de uma tabela de decisão é a regra de decisão, que define o conjunto de ações a serem tomadas, a partir de um conjunto de condições.

Para análise dos resultados da execução dos casos de teste, considera-se que os *Methods* podem retornar valor a quem os chama, embora isso não se faça regularmente para as *Instigations* (que invocam a execução do *Method*). Então, quando um método retorna valor, deve ser observado se este valor, como resultado

dos casos de teste do *Method*, corresponde aos valores esperados, ou seja, se o caso de teste foi aprovado ou não. Os *Methods* que não retornam valores podem, por outro lado, alterar *Attributes*. Neste caso, os casos de teste irão considerar o valor dos *Attributes* para fins de checagem do teste. É necessário, assim, identificar quais são os *Attributes* que são alterados pelo *Method*.

4.2.2.5.1 Planejamento e geração de casos de teste para Method

Para exemplificar como classes de equivalência e valores limite podem ser derivadas a partir da especificação, considere a descrição do plano de teste para o *Method* mtAirplaneAddNewBullet apresentado na Tabela 34. Esta é a especificação de um *Method* que deve produzir um projétil na tela. São descritas condições sobre valores de entrada e saídas esperadas para execução deste *Method*.

Tabela 34 – Plano de teste para o *Method* mtAirplaneAddNewBullet

Identificador do Plano de Teste test_mtAirplaneAddNewBullet01
Descrição Este documento do plano de testes do *Method* mtAirplaneAddNewBullet descreve o que deverá ser testado, recomenda e descreve as estratégias a serem empregadas nesses testes.
Referências Documento de Levantamento de Requisitos.
Itens testáveis O *Method* precisa gerar um projétil na tela quando a tecla de espaço for pressionada. Para isto, devem ser verdadeiras as seguintes condições: 1) O *Attribute* atBullet do *FBE* Airplane precisa ser maior que zero 2) O *Attribute* atLifePoints do *FBE* Airplane precisa ser maior que zero 3) O *Attribute* atGameStatus do *FBE* Stage 1 precisa ser igual a 2 (Playing) Estes três *Attributes* são consultados uma vez para que o *Method* possa realizar a ação. Determinar classes de equivalência e valores limite que exercitem os estados da *Premise*. Versão do *Method*: 1_0. *FBE*: Airplane.
Abordagem Teste de unidade Necessidades: Visual Studio 2010, *framework* PON Otimizado, código fonte da aplicação

As condições de entrada são os *Attributes* atBullet pertencente ao *FBE* Airplane, atLifePoints pertencente ao *FBE* Airplane e atGameStatus pertencente ao *FBE* Stage 1. (1) o *Attribute* atBullet precisa ser um número inteiro com valor maior que zero, (2) o *Attribute* atLifePoints precisa ser um número inteiro com valor maior

que zero e (3) o *Attribute* atGameStatus precisa ser um número inteiro com valor igual a 2 (Playing). Se as três condições não forem satisfeitas, não deve ser gerado nenhum projétil, i.e. o *Method* não deve realizar nenhuma ação.

Inicialmente, são identificadas as classes de equivalência e pode-se atribuir um identificador para cada uma. Posteriormente devem ser identificados os valores limite. Tabelas podem ser utilizadas para organizar e anotar os valores encontrados. Podem se utilizados identificadores para agrupar classes de equivalência e valores limite. Neste exemplo, serão utilizados os identificadores CExxx, onde CE significa “classe de equivalência” e xxx é um identificador inteiro único. Cada classe será categorizada como válida ou inválida para o domínio de entrada. Os *Attributes* são definidos como variáveis de entrada ou variáveis de saída.

Para gerar as primeiras classes de equivalência, considere a condição 1, que avalia o *Attribute* atBullet. Esta é uma condição de considera a faixa de valores maiores que zero, então, devem ser geradas duas classes de equivalência:

- CE1. A variável de entrada atBullet é maior que zero, válido.
- CE2. A variável de entrada atBullet é menor ou igual a zero, inválido.

É possível perceber que a condição 2, que avalia o *Attribute* atLifePoints, também requer valores maiores que zero, portanto, desenvolvem-se classes de equivalência da mesma maneira:

- CE3. A variável de entrada atLifePoints é maior que zero, válido.
- CE4. A variável de entrada atLifePoints é menor ou igual a zero, inválido.

Por sua vez, a condição 3, que avalia o *Attribute* atGameStatus, especifica que a o valor do *Attribute* “deve ser” igual a 2. Portanto, classes de equivalência válidas e inválidas são determinadas da seguinte forma:

- CE5. A variável de entrada atGameStatus é igual a 2, válido.
- CE6. A variável de entrada atGameStatus é diferente de 2, inválido.

Como visto, as classes de equivalência são identificadas separadamente. Por sua vez, a análise de valor limite é utilizada para refinar os resultados do particionamento em classes de equivalência. Um conjunto de identificadores pode ser utilizado para representar os tipos de valores limite identificáveis. Por exemplo:

BLI – Um valor imediatamente abaixo do valor do limite inferior.

LI – O valor do limite inferior.

CLI – Um valor imediatamente acima do valor do limite inferior.

BLS – Um valor imediatamente abaixo do valor do limite superior.

LS – O valor do limite superior.

CLS – Um valor imediatamente acima do valor do limite superior.

Para o *Method* de exemplo, os grupos de valores limite são:

- atLifePoints

  BLI: 0, LI: 1, CLI: 2

- atBullet

  BLI: 0, LI: 1, CLI: 2

- atGameStatus

  BLI: 1, LI: 2, CLI: 3

Como nenhuma das três condições é relacionada a um intervalo de valores, não foi necessário determinar os valores limite superiores BLS, LS e CLS. A Tabela 35 apresenta a determinação de classes de equivalência e valores limite para cada *Attribute* deste *Method*.

Tabela 35 – Classes de equivalência e análise de valor limite para exercitar mtAirplaneAddNewBullet

<table>
<tr><td colspan="4">Method – mtAirplaneAddNewBullet</td></tr>
<tr><td rowspan="3">Parâmetros de entrada</td><td>Integer</td><td colspan="2">atBullet</td></tr>
<tr><td>Integer</td><td colspan="2">atLifePoints</td></tr>
<tr><td>Integer</td><td colspan="2">atGameStatus</td></tr>
<tr><td>Ação</td><td colspan="3">Criação de um projétil do avião durante o jogo</td></tr>
<tr><td>Classes de equivalência válidas</td><td>Classes de equivalência inválidas</td><td>Valor limite válido</td><td>Valor limite inválido</td></tr>
<tr><td>atBullet>0</td><td>atBullet<=0</td><td>1; 2</td><td>0</td></tr>
<tr><td>atLifePoints>0</td><td>atLifePoints<=0</td><td>1; 2</td><td>0</td></tr>
<tr><td>atGameStatus=2</td><td>atGameStatus<>2</td><td>2</td><td>1; 3</td></tr>
</table>

O próximo passo no projeto dos casos de teste é selecionar um conjunto dos valores que cubra todas as classes de equivalência e fronteiras. Novamente, uma

tabela pode ser utilizada para apresentar os resultados de maneira organizada. A Tabela 36 apresenta um procedimento de caso de teste para este método.

Tabela 36 – Caso de teste para mtAirplaneAddNewBullet

ID	test_mtAirplaneAddNewBullet	
Propósito	Verificar se o método produz um projétil com os parâmetros esperados	
Pré-requisitos	Visual Studio 2010 configurado com *framework* PON Otimizado e código fonte da aplicação	
Passos	**Ação / necessidade**	**Resultado Esperado**
1	Executar a aplicação PON	Aplicação PON em execução
2	Inserir *breakpoint* na primeira linha do método mtAirplaneAddNewBullet do *FBE* Airplane.	*Breakpoint* inserido
3	Pressionar o botão de ataque	Software interrompido no *breakpoint* do *FBE* Airplane (inserido no passo 2)
4	Verificar se o método recebe por parâmetro os *Attributes*: atBullet (Airplane): precisa ter valor maior que 0 atLifePoints (Airplane): precisa ser maior que 0 atGameStatus (Stage): precisa ser igual a 2 (Jogando)	Os *Attributes* estavam configurados conforme o esperado
5	Pressionar F5 para que a aplicação continue normalmente e verificar se foi criado um projétil durante a execução do jogo.	A aplicação continuou normalmente e foi criado um projétil na tela durante o jogo.

A Tabela 37 apresenta uma combinação das possibilidades de valores limite válidos (i.e. V) e valores limite inválidos (i.e. I) para cada caso de teste. Por exemplo, para o caso de teste 1, os três *Attributes* avaliados precisam ter valores limite válidos para que a ação (cria um projétil) possa ocorrer.

Tabela 37 – Casos de teste que exercitam valores limite do *Method*

Caso de teste	**Condições**			
#	**atBullet**	**atLifePoints**	**atGameStatus**	**Ação ou saída esperada**
1	V	V	V	Cria um projétil
2	V	V	I	<sem ação>
3	V	I	V	<sem ação>
4	V	I	I	<sem ação>
5	I	V	V	<sem ação>
6	I	V	I	<sem ação>
7	I	I	V	<sem ação>
8	I	I	I	<sem ação>

A partir desta tabela de decisão, podem ser gerados casos de teste com valores reais, como apresenta a Tabela 38. Por exemplo, o caso de teste 1 testa

valores limite válidos identificados para provocar a criação de um projétil na tela de jogo.

Tabela 38 – Casos de teste para o *Method* mtAirplaneAddNewBullet

Casos de Teste	atBullet	atLifePoints	atGameStatus	Ação ou saída esperada
1	1	1	2	Cria um projétil
2	2	1	1	<sem ação>
3	1	0	2	<sem ação>
4	2	0	3	<sem ação>
5	0	1	2	<sem ação>
6	0	2	1	<sem ação>
7	0	0	2	<sem ação>
8	0	0	3	<sem ação>

A Tabela 39 apresenta uma maneira para descobrir o número de casos de testes que podem ser gerados para exercitar este *Method* com base nas combinações de valores limite suportados pelos *Attributes*. Nota-se que podem ser gerados no mínimo 12 casos de testes diferentes.

Tabela 39 – Determinação de número de casos de teste

atBullet	atLifePoints	atGameStatus
2 valores possíveis (válido: maior que zero, inválido: menor ou igual que zero)	2 estados possíveis (válido: maior que zero, inválido: menor ou igual que zero)	3 estados possíveis (1 pausado, 2 jogando e 3 parado)
Número mínimo de casos de teste: 2*2*3 = 12 casos de teste diferentes (exercitando valores limite).		

4.2.3 Avaliação dos Resultados dos Casos de Teste Unitários

O teste unitário ocorre em escopo bastante restrito às entidades testáveis do PON. As entidades têm um propósito simples e bem definido que permite, basicamente, apenas um critério para gerar casos de teste. Portanto, nessa fase, são conhecidos os casos de teste que exercitam cada entidade de acordo com o plano de teste individual.

O teste unitário em PON deve ser confrontado com os casos de teste de integração, em que as entidades são consideradas em conjunto. No teste de integração é possível confirmar se cada entidade foi projetada adequadamente e se notifica as entidades esperadas.

4.2.4 Considerações Sobre o Teste Unitário proposto

As abordagens de teste unitário para aplicações PON apresentada, propôs uma abordagem para planejar e executar os testes sobre as unidades testáveis das aplicações PON. Cada uma possui características e funcionamentos distintos que requerem um planejamento e geração de casos de teste específicos.

Para geração de casos de teste para *Premisses* é requerida a determinação de classes de equivalência e análise de valores limite que exercitem o operador lógico e o (s) *Attribute* (s) avaliado. Para geração de casos de teste para *Conditions* e *SubConditions* são requeridos casos de teste que exercitem os estados das *Premisses* ou *SubConditions* avaliadas pelo operador condicional. Para geração de casos de teste para as *Rules* são considerados apenas os estados da *Condition* avaliada (aprovada ou não aprovada), portanto, apenas dois casos de teste são suficientes para exercitá-las. Para geração de casos de teste para *Methods* dos *FBEs* é utilizada a técnica de teste funcional para exercitar parâmetros de entrada do *Method* (quando tem) e a configuração de outros *Attributes* ou estados de objetos que serão utilizados. Quando um *Method* não tem parâmetros de entrada deve-se verificar no plano de teste e descrição do caso de teste se é necessário configurar entidades externas como valores de *Attributes* do próprio *FBE* ou de outras classes e também estados de objetos.

Esses critérios permitem conhecer os principais casos de teste que exercitam cada entidade PON e podem ser aplicados em qualquer sistema desenvolvido em PON. Na fase de teste de integração, apresentada na próxima seção, as entidades são avaliadas em conjunto e confere-se se cada unidade está notificando as entidades que deveria notificar, conforme o planejado no documento de Levantamento de Regras do Sistema.

O teste unitário em PON possibilita identificar casos de teste e seus respectivos resultados esperados. O teste de integração, por sua vez, permite identificar casos de teste que exercitem as unidades em conjunto, e para isto, possibilita a identificação de variados tipos de erros.

4.3 TESTE DE INTEGRAÇÃO EM PON

Esta seção apresenta o teste de integração proposto para o PON. A Seção 4.3.1 apresenta uma visão geral da fase de teste de integração, a Seção 4.3.2 apresenta uma estratégia de teste com casos de uso, a Seção 4.3.3 apresenta o teste de integração em PON utilizando casos de uso, a Seção 4.3.4 apresenta a avaliação dos resultados e a Seção 4.3.5 apresenta as considerações sobre o teste de integração apresentado.

4.3.1 Visão Geral do Teste de Integração

Apenas a verificação individual ou isolada de unidades (ou também denominados componentes) não garante o funcionamento adequado de todo o sistema. Falhas na integração entre os componentes podem causar interações que não deveriam ser produzidas (BINDER, 1999). Por isto, o teste de integração sucede o teste de unidade e visa testar o comportamento e o funcionamento dos componentes operando em conjunto.

Os componentes participam da realização de uma ou mais atividades e podem interagir com vários outros componentes para realização dos casos de uso de um software OO ou PON. Casos de uso representam uma abstração de funcionalidade de um sistema sob o ponto de vista do usuário. Também, são um complemento à especificação de requisitos no desenvolvimento orientado a objetos. Uma coleção de casos de uso apresenta todas as funcionalidades do sistema (BOOCH; RUMBAUGH; JACOBSON, 2005). Além disto, casos de uso podem ser utilizados como base para orientar o planejamento dos testes de integração (BINDER, 1999).

Em um software PON, testes de integração podem ser realizados por meio de duas abordagens. A primeira busca gerar casos de teste que exercitem as descrições, funcionalidades e comportamentos esperados de cada caso de uso. A segunda busca gerar casos de teste que exercitem diretamente as entidades que implementam cada caso de uso (i.e. *Attributes*, *Premisses*, *Conditions*, *Rules* e *Methods*) provocando condições que possibilitem avaliar os fluxos de notificação da aplicação PON.

A Figura 32 apresenta a visão expandida da fase de teste de integração que inclui as atividades de planejamento, geração de casos de teste, execução e avaliação dos resultados. Essa figura detalha a fase de teste de integração apresentada na Figura 29 (Seção 4.1, pg. 96), decompondo as atividades de Planejamento e de Execução da fase de teste de integração.

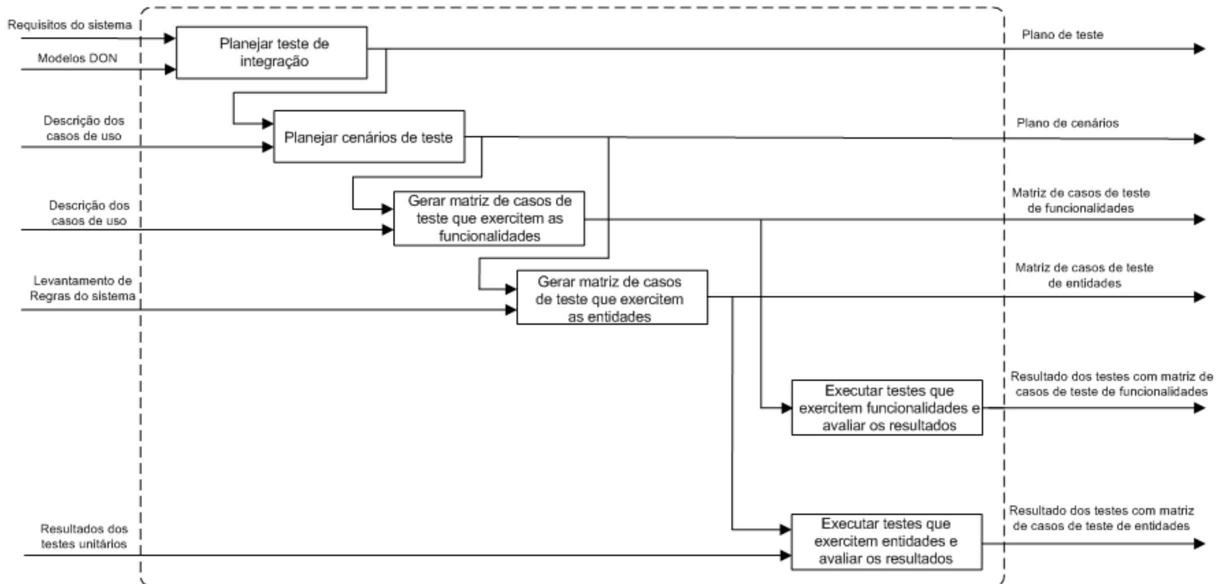


Figura 32 – Visão expandida da fase de teste de integração

A atividade de “Planejar teste de integração” envolve a elaboração do “Plano de teste”, principal documento norteador do teste de integração, e tem como dados de entrada os “Requisitos do sistema” e “Modelos DON”. A atividade de “Planejar cenários de teste” requer a identificação dos principais “caminhos” que devem ser exercitados durante a execução do software, ou seja, interações que causam diferentes comportamentos e que podem ser visíveis ao usuário. Estes caminhos têm como dados de entrada a “Descrição dos casos de uso” e o “Plano de teste”.

A atividade de “Gerar matriz de casos de teste que exercitem as funcionalidades” compreende a identificação das funcionalidades previstas para a realização dos casos de uso e necessita as informações da “Descrição dos casos de uso” e “Plano de cenários”, que guiam a elaboração dos casos de teste.

A atividade de “Gerar matriz de casos de teste que exercitem as entidades” requer o “Plano de cenários” e o “Levantamento de Regras do sistema”. Esta atividade busca exercitar diretamente o fluxo de notificações das entidades PON (como *Attributes*, *Premisses*, *Conditions*) que implementam os casos de uso. Verifica

se cada entidade foi desenvolvida conforme o esperado e se notifica as entidades que deveria notificar.

A atividade de “Executar testes que exercitem as funcionalidades e avaliar os resultados” requer a “Matriz de casos de teste de funcionalidades”. Durante a execução destes testes, os resultados são registrados na forma de "Resultado dos testes com matriz de casos de teste de funcionalidades”.

A atividade de “Executar testes que exercitem as entidades e avaliar os resultados” requer a “Matriz de casos de teste de entidades”, e os “Resultados dos testes unitários” de cada entidade que implementa o caso de uso.

4.3.2 Estratégia de Teste com Caso de Uso

O Fluxo de Eventos dos casos de uso é o aspecto mais importante considerado para o planejamento e geração de casos de teste com casos de uso. Basicamente, existem dois tipos de fluxos de eventos: o Fluxo Básico e o(s) Fluxo(s) Alternativo(s). O Fluxo Básico de eventos deve cobrir o que normalmente deve acontecer quando o caso de uso é realizado. Os Fluxos Alternativos abrangem os comportamentos opcionais (ou variações de comportamento) relativos ao comportamento normal ou básico. Fluxos Alternativos também são chamados de "desvios" do Fluxo Básico (HEUMANN, 2001). Podem existir ainda os fluxos de exceção, que neste caso, podem ser equiparados a fluxos alternativos por também serem desvios do fluxo básico ou alternativo. A Figura 33 ilustra os fluxos básico e alternativos que podem existir para realização de um caso de uso.

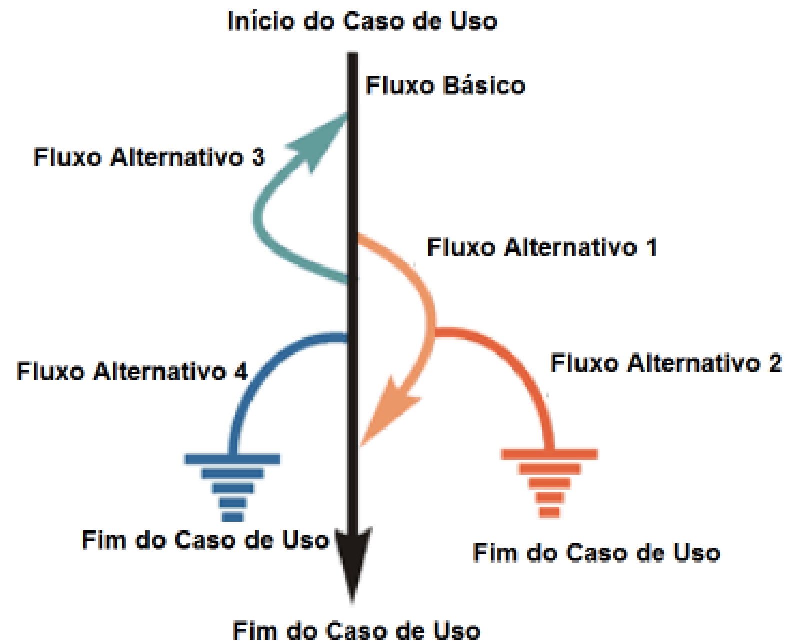


Figura 33 – Fluxo básico e fluxos alternativos de eventos em um caso de uso
Fonte: Adaptado de Heumann (2001)

Segundo Heumann (2001), um caso de uso é formado por um conjunto de cenários (ou instâncias). Um cenário é um "caminho" completo através do caso de uso, que pode envolver um ou mais fluxos de execução. Por exemplo, na Figura 33, seguir o fluxo básico seria um cenário. Seguir o fluxo básico mais o Fluxo Alternativo 1 seria outro cenário. Seguir o Fluxo Básico mais o Fluxo Alternativo 2 seria o terceiro cenário, e assim por diante. Cada cenário descreve comportamentos diferentes do sistema, suas falhas ou casos excepcionais. Dependendo da natureza dos casos de uso considerado, um grande número de cenários pode ser enumerado.

Embora sejam utilizados para descrever requisitos no escopo de sistema, os casos de uso são desenvolvidos em escopo de classes, *clusters* (grupo de classes interdependentes) ou subsistemas. Com isso, é possível planejar e desenvolver casos de teste por meio da análise de instâncias específicas de um caso de uso e os comportamentos correspondentes esperados.

4.3.3 Teste de Integração em PON Utilizando Casos de Uso

Foram propostas duas abordagens para o teste de integração utilizando casos de uso: (1) Testes que exercitem as informações da documentação e comportamento esperado dos casos de uso e (2) Testes que exercitem diretamente

as entidades que implementam o caso de uso (i.e. *Attributes*, *Premisses*, *SubConditions*, *Conditions* e *Rules*). Para ambas as abordagens, foi utilizada a Matriz de Casos de Teste para casos de uso apresentada por Heumann (2001), que considera a geração de casos de teste sobre cenários e consiste em três passos: (1) Planejar Cenários de Teste, (2) Identificar Casos de Teste e (3) Identificar Valores de Casos de Teste. Após o passo de identificar valores de casos de teste, devem ser executados os casos de teste e avaliados seus resultados.

4.3.3.1 Testes que exercitem as informações da documentação do caso de uso (funcionalidades)

A primeira abordagem do teste de integração exercita o comportamento da aplicação e utiliza as informações do Plano de Teste e Descrição do Caso de Uso. Requer a identificação dos cenários que serão testados, a elaboração da matriz de casos de teste, que utiliza informações dos cenários identificados, a geração, execução e avaliação dos casos de teste. As subseções a seguir detalham os passos envolvidos nos testes de integração relacionados às funcionalidades dos casos de uso.

4.3.3.1.1 Planejar os cenários de teste

Para exemplificar o planejamento de testes de um sistema PON utilizando casos de uso, considere o Plano de Teste para o caso de uso Controlar Avião apresentado na Tabela 40. Esse caso de uso faz parte do Diagrama de Casos de Uso apresentado na Figura 18 (Seção 3.2.2, pg. 76).

Tabela 40 – Plano de Teste para o caso de uso Controlar Avião

Identificador do Plano de Teste testplan_ucControlarAviao1
Referências Especificação dos requisitos.
Introdução Este documento de Plano de Testes informa ao testador as condições necessárias para execução e análise de resultados do caso de uso Controlar Avião pertencente ao diagrama de Casos de Uso do sistema. Objetiva apresentar os requisitos que serão testados, recomenda e descreve as estratégias a serem empregadas durante os testes e identifica os recursos necessários.
Itens testáveis Deverão ser testados os cenários de execução do caso de uso. Tipo: Teste funcional (caixa-preta) e Teste estrutural (caixa-branca). Versão do diagrama de casos de uso: 1_0. Caso de uso: Controlar Avião.
Questões de risco de software Requisitos: Requisitos mal especificados.
Recursos a serem testados Geração de projéteis na tela de jogo quando o jogador pressiona o botão de ataque. Movimentação horizontal do avião para a direita. Movimentação horizontal do avião para a esquerda. Limite lateral para à esquerda. Limite lateral para à direita.
Recursos a não serem testados As atividades a seguir não deverão ser testadas: • Velocidade de movimentação do avião; • Velocidade de movimentação dos projéteis; • Dimensões do projétil gerado.
Abordagem O teste de integração será feito pelo testador e será aprovado pelo líder da equipe de teste. Seguir a folha de descrição do caso de teste do caso de uso (Matriz de Casos de Teste) que deve ser elaborada pelo testador com base na análise deste Plano de Teste.
Critérios de aprovação • Critérios de aprovação Todos os casos de teste precisam ser executados. Casos de teste devem cobrir todos os cenários possíveis.
Critérios de suspensão e reinício • Critérios de suspensão Problemas de execução durante os casos de teste. Problemas de execução da ferramenta que auxilia a realização dos testes. • Critérios de reinício Realizar ações corretivas para corrigir eventuais problemas identificados.

Plano de Teste para o caso de uso Controlar Avião (continuação)

Entregas de Teste Aceitação do plano de testes. Tabela com resultados dos casos de teste gerados e executados. Registros de ocorrências de problemas durante a execução dos casos de teste e ações corretivas tomadas.
Tarefas remanescentes de teste Não tem.
Necessidades de pessoal e de treinamento Não tem.
Responsabilidades • Testador Executar casos de teste. Confrontar os resultados produzidos com os resultados esperados. Registrar todas as ocorrências de falhas ou atividades que necessitaram ser realizadas.
Agenda Não tem.
Planejamento de riscos e contingências Não tem.
Aprovações • Testador. • Líder da equipe de teste.

A Tabela 41 apresenta a Descrição do Caso de Uso Controlar Avião.

Tabela 41 – Descrição do caso de uso Controlar Avião

ID Caso de Uso	UC1		
Nome do Caso de Uso	Controlar Avião		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Jogador		
Descrição	Este caso de uso controla as funções de ataque e movimentação do avião.		
Disparador	O caso de uso inicia quando o jogador iniciar o jogo.		
Pré-condições	O jogo precisa estar em estado jogando.		
Pós-condições	Pós-condição 1) Deverá ser disparado um projétil a partir do bico do avião. Pós-condição 2) O avião movimenta-se para a direita. Pós-condição 3) O avião movimenta-se para a esquerda.		
Fluxo Básico	1. O jogo fica com estado jogando. 2. O sistema aguarda o jogador pressionar algum botão.		
[Fluxo Alternativo 1]	O jogador pressiona o botão de ataque. 1. No passo 2 do fluxo básico o jogador pressiona o botão de ataque. 2. O sistema verifica se o estado de jogo é “jogando”. 3. O sistema verifica se o jogador possui mais que zero de pontos de vida. 4. O avião precisa ter mais que zero de quantidade de munição. 5. O sistema gera um novo projétil na tela. 6. O sistema permite que o projétil seja desenhado na tela. 7. O sistema permite que o projétil siga uma trajetória em linha reta para cima na tela.		
[Fluxo Alternativo 2]	O jogador pressiona o botão esquerdo. 1. No passo 2 do fluxo básico o jogador pressiona o botão esquerdo. 2. O sistema verifica se o estado de jogo é “jogando”. 3. O sistema verifica se o jogador possui mais que zero de pontos de vida. 4. O sistema verifica se o jogador não está no limite esquerdo da tela. 5. O sistema move o avião uma posição para esquerda.		
[Fluxo Alternativo 3]	O jogador pressiona o botão direito. 1. No passo 2 do fluxo básico o jogador pressiona o botão direito. 2. O sistema verifica se o estado de jogo é “jogando”. 3. O sistema verifica se o jogador possui mais que zero de pontos de vida. 4. O sistema verifica se o jogador não está no limite direito da tela. 5. O sistema move o avião uma posição para direita.		

Descrição do caso de uso Controlar Avião (continuação)

Exceções	Exceção 1) O jogador não pode disparar projétil 1. No passo 4 do fluxo alternativo 1, se o avião não tiver mais munição, não poderá ser gerado projétil. 2. O avião fica parado na posição em que está. 3. O caso de uso continua no passo 2 do fluxo básico. Exceção 2) O jogador não pode se movimentar para a esquerda 1. No passo 4 do fluxo alternativo 2, se o avião estiver no limite esquerdo da tela, o avião não pode se mover para esquerda. 2. O avião fica parado na posição em que está. 3. O caso de uso continua no passo 2 do fluxo básico. Exceção 3) O jogador não pode se movimentar para a direita 1. No passo 4 do fluxo alternativo 3, se o avião estiver no limite direito da tela, o avião não pode se mover para direita. 2. O avião fica parado na posição em que está. 3. O caso de uso continua no passo 2 do fluxo básico.
Inclusões	O caso de uso Controlar Avião não inclui outros Casos de Uso.
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando.
Requisitos	Os requisitos funcionais implementados total ou parcialmente são: RF002, RF008, RF012, RF015, RF016. Os requisitos não realizados funcionais implementados total ou parcialmente são: RNF001, RNF004, RNF005.
Considerações	Considera-se que: • O botão de ataque é a tecla de espaço do telado padrão. • O botão esquerdo é a seta esquerda do teclado padrão • O botão direito é a seta direita do teclado padrão. • O avião não tem projétil quando o número de projéteis é igual ou menor que zero. • O avião está no limite esquerdo quando a posição do avião é zero. • O avião está no limite direito quando a posição do avião é 800. • Os estados de jogo possuem valores: (1) Estado de jogo “pausado”, (2) Estado de jogo “jogando” e (3) Estado de jogo “parado”.

Para identificar os cenários de teste é preciso analisar o Plano de Teste e Descrição do Caso de Uso observando cada combinação possível de fluxo principal, fluxos alternativos e fluxos de exceção. A Tabela 42 apresenta a matriz de identificação de cenários para o caso de uso Controlar Avião. A elaboração dessa matriz é uma atividade criativa e está relacionada à experiência do testador e a leitura e compreensão dos documentos de requisitos e descrição do caso de uso.

Tabela 42 – Matriz de identificação de cenários do caso de uso Controlar Avião

Nome do cenário	Fluxo Inicial	Fluxo Alternativo	Fluxo de Exceção
Cenário 1 – Avião não realiza ação	Fluxo Básico		
Cenário 2 – Avião dispara projétil	Fluxo Básico	Fluxo Alternativo 1	
Cenário 3 – Avião não tem munição	Fluxo Básico	Fluxo Alternativo 1	Fluxo de Exceção 1
Cenário 4 – Avião se movimenta para esquerda	Fluxo Básico	Fluxo Alternativo 2	
Cenário 5 – Avião não pode se movimentar para esquerda	Fluxo Básico	Fluxo Alternativo 2	Fluxo de Exceção 2
Cenário 6 – Avião se movimenta para a direita	Fluxo Básico	Fluxo Alternativo 3	
Cenário 7 – Avião não pode se movimentar para direita	Fluxo Básico	Fluxo Alternativo 3	Fluxo de Exceção 3

4.3.3.1.2 Identificar casos de teste

A partir da identificação dos cenários, deve-se novamente analisar as descrições do Plano de Teste e da Descrição do Caso de Uso Controlar Avião para identificar as variáveis operacionais que serão necessárias para projetar a matriz de casos de teste. As variáveis operacionais são todos os fatores que variam de um cenário para outro e determinam diferentes respostas do sistema que incluem entradas e saídas explícitas, condições ambientais que resultam em comportamentos diferentes do sistema, atributos valoráveis, abstrações de estados do sistema sob teste (e.g., o estado do jogo precisa ser “jogando”, o jogador precisa ter mais que zero de pontos de vida, e assim por diante), conforme apresentado na Seção 2.4.3.1.1.

Com base na análise desses documentos, principalmente a descrição do caso de uso Controlar Avião, foram identificadas 8 variáveis operacionais:

- Estado de jogo *“*jogando*”*: Fluxos alternativos 1, 2 e 3;
- Ter pontos de vida: Fluxos alternativos 1, 2 e 3;
- Ter munição: Fluxo alternativo 1;
- Pressionar botão de ataque: Fluxo alternativo 1;
- Pressionar botão esquerdo: Fluxo alternativo 2;
- Pressionar botão direito: Fluxo alternativo 3;
- Limite esquerdo da tela: Fluxo alternativo 2;
- Limite direito da tela: Fluxo alternativo 3.

Na primeira versão da matriz de casos de teste (Tabela 43), cada variável operacional pode assumir um dos seguintes estados: Válido (V), Inválido (I) e Não Avaliado (N/A)[5] para cada caso de teste. Estes estados são identificados pela análise dos fluxos e variáveis operacionais que implementam os cenários. Por exemplo, a descrição do fluxo alternativo 1 no passo 2 (descrição do caso de uso apresentada na Tabela 41) diz: "O sistema verifica se o estado de jogo é 'jogando'". Então, o estado válido (i.e. V) corresponde a "jogando" e o estado inválido (i.e. I) é um conjunto com quaisquer outros valores que esta variável poderia assumir e que não seria válido para o caso de teste.

Tabela 43 – Matriz de Casos de Teste do caso de uso Controlar Avião

ID caso de teste	Cenário/ Condição	Pontos de vida	Estado de jogo	Munição	Botão ataque	Botão esquer-do	Botão direito	Limite esquerdo	Limite direito	Resultado esperado
CT01	Cenário 1 Avião não realiza ação	V	V	N/A	N/A	N/A	N/A	N/A	N/A	O avião não dispara e não se movimenta
CT02	Cenário 2 Avião dispara projétil	V	V	V	V	N/A	N/A	N/A	N/A	Avião disparar um projétil
CT03	Cenário 3 Avião não tem munição	V	V	I	V	N/A	N/A	N/A	N/A	Avião não pode disparar
CT04	Cenário 4 Avião se moviment a para esquerda	V	V	N/A	N/A	V	N/A	V	N/A	Avião se movimenta à esquerda
CT05	Cenário 5 Avião não pode se moviment ar para esquerda	V	V	N/A	N/A	V	N/A	I	N/A	Avião não pode se movimentar à esquerda
CT06	Cenário 6 Avião se moviment a para a direita	V	V	N/A	N/A	N/A	V	N/A	V	Avião se movimenta à direita
CT07	Cenário 7 Avião não pode se moviment ar para direita	V	V	N/A	N/A	N/A	V	N/A	I	Avião não pode se movimentar à direita

[5] Não Avaliada (N/A) significa que a variável operacional não deve receber valor durante a execução do caso de teste (i.e. não deve ser considerada no caso de teste).

4.3.3.1.3 *Identificar valores de casos de teste*

Quando os primeiros conjuntos de casos de teste com valores válidos e inválidos forem identificados, devem ser revistos e validados para garantir a precisão e identificar testes redundantes ou em falta. Então, a etapa final é substituir com valores reais de dados para V (i.e. Válido) e I (i.e. Inválido). Sem valores reais para teste, eles são apenas descrições de condições e os casos de teste não podem ser implementados ou executados (HEUMANN, 2001). Nesta etapa, também podem surgir novos casos de teste, considerando os valores suportáveis pelas variáveis.

Para o PON, sugere-se a determinação de classes de equivalência e análise de valores limite das variáveis operacionais para a geração de casos de teste. Por exemplo, para identificar as classes de equivalência da variável operacional "Ter munição" bastou notar a descrição no bloco de considerações do caso de uso: "O avião não tem projétil quando o número de projéteis é igual ou menor que zero". A partir disto, deve-se determinar também os valores limites válido e inválido, respectivamente, 1 e 0. Desta maneira, a Tabela 44 apresenta a determinação de classes de equivalência e análise de valores limite para cada variável operacional identificada.

Tabela 44 – Classes de equivalência e análise de valores limite para as variáveis operacionais

	Valores limite		Classes de equivalência	
Condições ou Variáveis Operacionais	Válido	Inválido	Válidas	Inválidas
Ter pontos de vida	1	0	Pontos de vida>0	Pontos de vida<=0
Estado de jogo *"jogando"*	2	1;3	2	Estados diferentes de 2
Ter munição	1	0	Munição>0	Munição<=0
Pressionar botão de ataque	Verdadeiro	Falso	Verdadeiro	Falso
Pressionar botão esquerdo	Verdadeiro	Falso	Verdadeiro	Falso
Limite esquerdo da tela	Verdadeiro	Falso	Verdadeiro	Falso
Pressionar botão direito	Verdadeiro	Falso	Verdadeiro	Falso
Limite direito da tela	Verdadeiro	Falso	Verdadeiro	Falso

Com base nos valores limite para as variáveis operacionais, a Tabela 45 apresenta alguns casos de teste com valores reais. Em PON, o ideal seria gerar casos de teste com todas as combinações de valores limite possíveis para garantir que todos os principais pontos de vulnerabilidade do sistema sejam devidamente

testados, porém nem sempre isto é viável. É possível conhecer o número de casos de teste que podem ser gerados para esta tabela, apenas considerando os valores limite. Considerando que foram utilizadas 8 variáveis operacionais e que cada variável operacional pode ter 3 estados, o número de casos de teste possíveis pode ser deduzido com a equação: $nro. de\ testes\ possíveis = 3^8$ que resulta em 6561 casos de teste diferentes.

Por isto, para simplificação neste exemplo, são apresentados apenas alguns casos de teste possíveis. Se a atividade de execução de casos de teste for realizada manualmente, como no caso do exemplo da Tabela 45, para tentar garantir que o mínimo de casos de teste sejam executados, deve-se executar cada caso de teste duas vezes, sendo que uma vez estritamente com valores limite válidos (que aprove o caso de teste) e outra com pelo menos um valor limite inválido (que não aprove o caso de teste).

Tabela 45 – Matriz de Casos de Teste com valores (caso de uso Controlar Avião)

ID caso de teste	Cenário/ Condição	Pontos de vida	Estado de jogo	Munição	Botão ataque	Botão esquer-do	Botão direito	Limite esquerdo	Limite direito	Resultado esperado
CT01	Cenário 1 Avião não realiza ação	1	2	N/A	N/A	N/A	N/A	N/A	N/A	O avião não dispara e não se movimenta
CT02	Cenário 2 Avião dispara projétil	1	2	1	V	N/A	N/A	N/A	N/A	Avião disparar um projétil
CT03	Cenário 3 Avião não tem munição	1	2	0	V	N/A	N/A	N/A	N/A	Avião não pode disparar
CT04	Cenário 4 Avião se moviment a para esquerda	1	2	N/A	N/A	V	N/A	V	N/A	Avião se movimenta à esquerda
CT05	Cenário 5 Avião não pode se moviment ar para esquerda	1	2	N/A	N/A	V	N/A	I	N/A	Avião não pode se movimentar à esquerda
CT06	Cenário 6 Avião se moviment a para a direita	1	2	N/A	N/A	N/A	V	N/A	V	Avião se movimenta à direita
CT07	Cenário 7 Avião não pode se moviment ar para direita	1	2	N/A	N/A	N/A	V	N/A	I	Avião não pode se movimentar à direita

Por exemplo, o caso de teste CT02 (Cenário 2) considera o valor 1 para a variável "pontos de vida", 2 para "estado de jogo" (sendo que 1 é "pausado", 2 é "jogando" e 3 é "parado"), 1 para "quantidade de munição" e verdadeiro para "botão de ataque pressionado", mas não considera valores para "botão direito pressionado", "botão esquerdo pressionado", "limite esquerdo" e "limite direito". Ou seja, estas quatro últimas variáveis não devem ser exercitadas neste caso de teste. Com a execução deste caso de teste, deve ser criado um objeto projétil, uma regra para desenhar o projétil e uma regra para movimentar o projétil. Porém, para este mesmo caso de teste, se for modificado o valor da variável pontos de vida para 0, o caso de teste não deve ser aprovado.

4.3.3.2 Testes diretamente sobre as entidades que implementam o caso de uso

Uma típica implementação de um caso de uso é uma colaboração, i.e. um fluxo de mensagens trocadas entre objetos e entidades. Conforme visto na Seção 2.3, as fases de análise de um software PON são Levantamento de Requisitos e Elaboração de Casos de Uso. As fases de projeto de um software PON envolvem a elaboração de vários diagramas do DON como Diagrama de Componentes e Diagrama de Sequência.

Esta segunda abordagem do teste de integração exercita diretamente as entidades PON que realizam os casos de uso. O diagrama de objetos desenvolvido nas fases de projeto PON (Seção 3.2.7, pg. 91) auxilia a apresentação de todas as entidades que participam da realização de cada caso de uso. Este diagrama possui uma notação para cada tipo de entidade PON e exibe o sentido do fluxo de notificações entre elas.

Para geração e execução de casos de teste também é utilizada a Matriz de Casos de Teste. Aqui, deve-se considerar que os *Attributes* são equivalentes às variáveis operacionais e os resultados da execução dos casos de teste são os estados esperados das *Premisses*, *SubConditions*, *Conditions* e *Rules*.

4.3.3.2.1 Planejar cenários de teste

O Plano de Teste e o Plano de Cenários para o teste de integração que exercitam diretamente as entidades PON que implementam cada caso de uso são, basicamente, os mesmos apresentados para o teste sobre a descrição ou funcionalidades do caso de uso, conforme apresentado na Seção 4.3.3.1.1.

Para gerar casos de teste que exercitem os caminhos do fluxo de execução, é necessário ter o diagrama de objetos (ou outro diagrama equivalente) que apresenta todas as entidades que realizam o caso de uso sob teste (desenvolvido durante a modelagem com o uso do DON). No diagrama de objetos, pode-se considerar que as entidades centrais são as *Rules*. Conhecendo as entidades que fazem parte da composição das *Rules* (*Conditions*, *SubConditions*, *Premisses*), e as *Instigations* e *Methods* associadas a elas, é possível conhecer todas as entidades que realizam um caso de uso.

Por exemplo, a Figura 34 apresenta as *Rules* e todas as entidades que compõem o caso de uso Controlar Avião representado na forma de um diagrama de objetos. Para realizar este caso de uso, foram identificadas as seguintes *Rules*: rlAirplaneMovingLeft, rlAirplaneMovingRight e rlAirplaneShoots. A partir da identificação destas *Rules* é possível determinar todas as outras entidades PON envolvidas na implementação deste caso de uso.

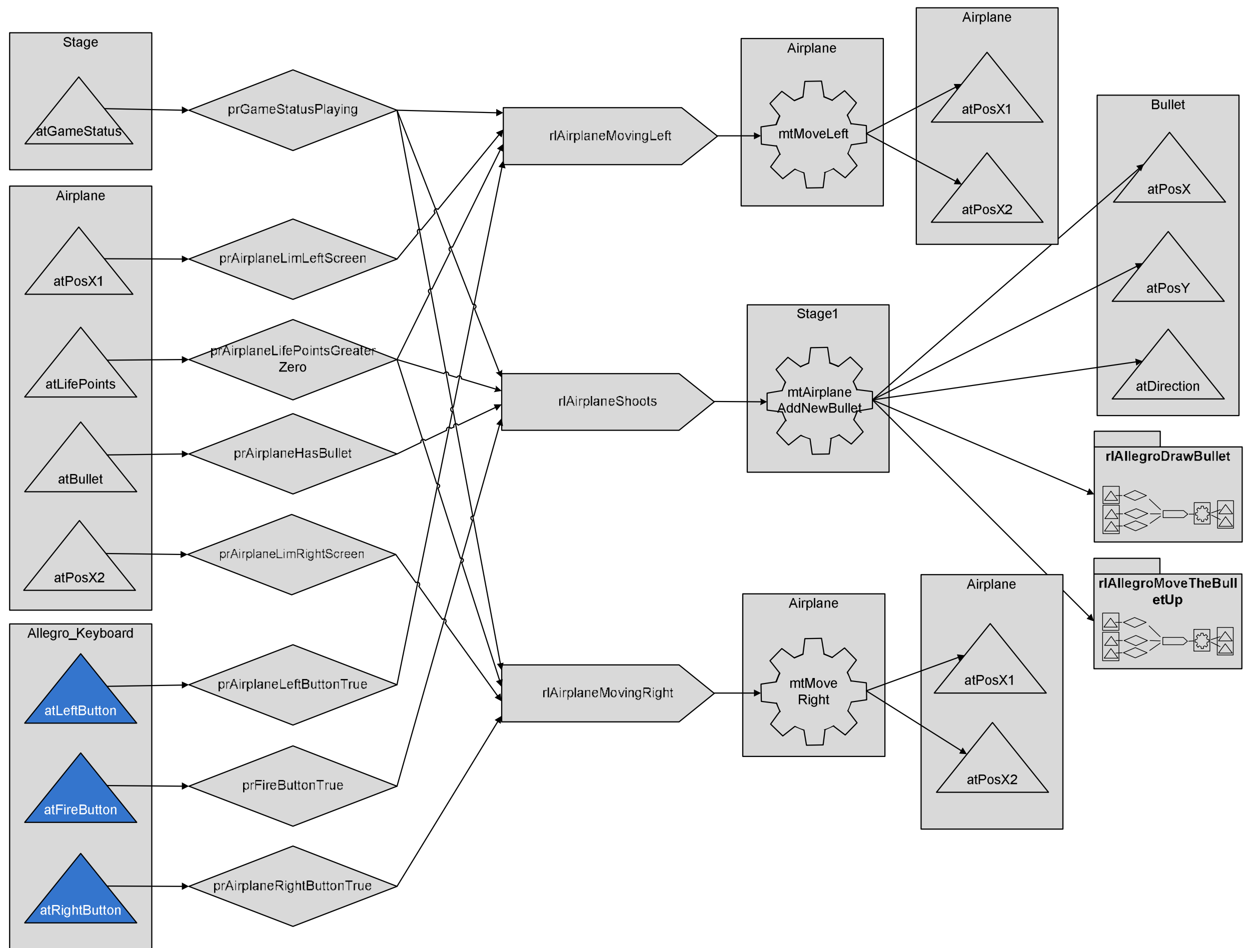


Figura 34 – Diagrama de objetos do caso de uso Controlar Avião

Os pacotes rlAllegroDrawBullet e rlAllegroMoveTheBulletUp representam todas as entidades que participam da realização (compõem) estas *Rules* (incluindo *Attributes*, *Premisses*, *Conditions* e *SubConditions*) que foram agrupadas para simplificar a visualização neste diagrama. A Figura 35 apresenta a visão interna do pacote rlAllegroDrawBullet. Nota-se que a *Rule* rlAllegroDrawBullet não alterna valor de nenhum *Attribute*, i.e. apenas apresenta uma imagem do projétil na tela de jogo (por meio de uma função da biblioteca Allegro). A Figura 36 apresenta a visão interna do pacote rlAllegroMoveTheBulletUp.

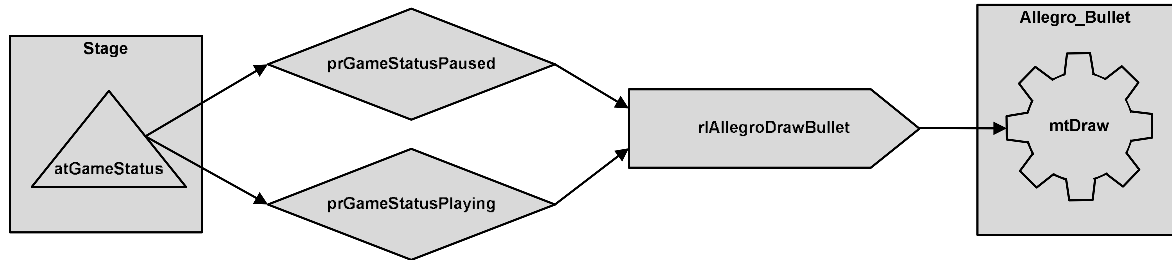


Figura 35 – Visão interna do pacote rlAllegroDrawBullet

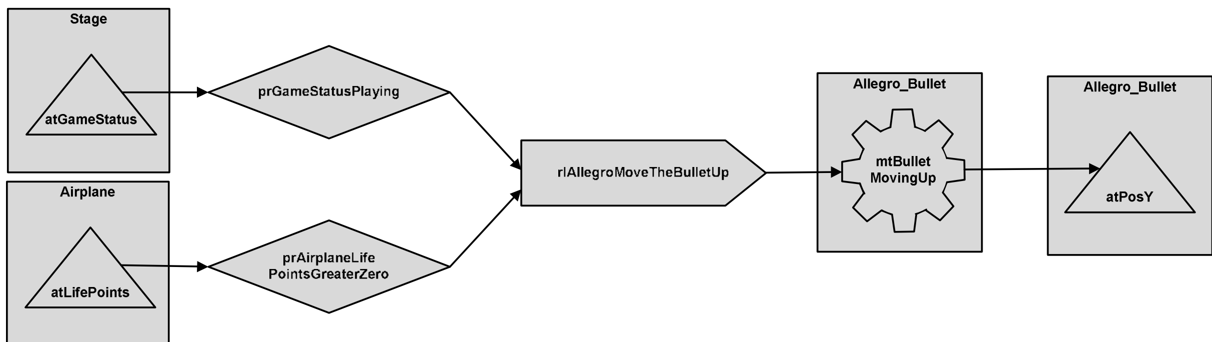


Figura 36 – Visão interna do pacote rlAllegroMoveTheBulletUp

Como pode ser observado, a alteração de valores dos *Attributes* resulta em fluxos de notificação que notificam desde as entidades *Premise* até *Rule* (que consequentemente executa os *Methods*). Nestes diagramas, as entidades *Condition* e *SubCondition* podem ser suprimidas quando o operador lógico é de Conjunção. Por exemplo, a *Condition* da *Rule* rlAirplaneShoots avalia uma conjunção de quatro *Premisses* (prGameStatusPlaying, prAirplaneLifePointsGreaterZero, prAirplaneHasBullet e prAirplaneLimRightScreen) e, por isto, foi suprimida. O ator jogador controla diretamente o estado dos *Attributes* atFireButton, atLeftButton e atRightButton do *FBE* Allegro Keyboard, que aparecem destacados.

Posteriormente, é necessário identificar quais são os *Attributes* correspondentes às variáveis operacionais para que seja possível exercitá-las nos casos de teste. Para isto, a Tabela 46 sugere uma associação entre cada variável operacional identificada na Seção 4.3.3.1.2 e seu equivalente *Attribute*.

Tabela 46 – Variáveis operacionais e *Attributes* correspondentes

Variáveis operacionais	***Attributes***
Pontos de vida	atLifePoints
Estado de jogo	atGameStatus
Munição	atBullet
Limite esquerdo	atPosX1
Limite direito	atPosX2
Botão ataque	**atFireButton**
Botão esquerdo	**atLeftButton**
Botão direito	**atRightButton**

4.3.3.2.2 Identificar Casos de Teste

Quando forem identificadas todas as entidades PON que implementam o caso de uso e os *Attributes* (equivalentes às variáveis operacionais), é possível elaborar uma matriz de casos de teste. Esta matriz é semelhante à apresentada no teste sobre as funcionalidades do caso de uso e, também, é composta por diferentes tipos de campos: (1) ID do caso de teste, (2) cenário de teste, (3) variáveis operacionais e (4) resultados esperados.

- No campo 1 deve ser apresentado o identificador (ID) do caso de teste;
- No campo 2 deve ser apresentado o nome do cenário de teste;
- No campo 3 devem ser apresentados todos os *Attributes* que serão diretamente exercitados pelos casos de teste;
- No campo 4 são apresentadas todas as entidades *Premisses*, *Conditions*, *SubConditions* e *Rules* que são influenciados pela mudança de valor dos *Attributes*.

Para exemplificar, a Tabela 47 apresenta a matriz de casos de teste com os cenários identificados anteriormente. Foram identificados apenas valores Válidos (i.e. V), Inválidos (i.e. I) e N/A (“Não Avaliado” para *Attributes* e “Não Aprovado” para outras entidades) para cada caso de teste.

Tabela 47 – Matriz de Casos de Teste para o Caso de Uso Controlar Avião

		Variáveis operacionais								**Resultados esperados**								
		Attributes								***Premisses***								***Rules***
ID caso de teste	**Cenário/ Condição**	**atLifePoints**	**atGameStatus**	**atBullet**	**AtFireButton**	**atLeftButton**	**atRighButton**	**atPosX1**	**atPosX2**	**prAirplaneLifePointsGreaterZero**	**prGamStatusPlaying**	**prAirplaneHasBullet**	**prFireButtonTrue**	**prLeftButtonTrue**	**prRightButtonTrue**	**prAirplaneLimLeftScreen**	**prAirplaneLimRightScreen**	
CT01	Cenário 1 Avião não realiza ação	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Nenhuma *Rule* aprovada
CT02	Cenário 2 Avião dispara projétil	V	V	V	V	N/A	N/A	N/A	N/A	V	V	V	V	N/A	N/A	N/A	N/A	**AprovarlAirplaneShoots**
CT03	Cenário 3 Avião não tem munição	V	V	I	V	N/A	N/A	N/A	N/A	V	V	I	V	N/A	N/A	N/A	N/A	Nenhuma *Rule* aprovada
CT04	Cenário 4 Avião se movimenta para esquerda	V	V	N/A	N/A	V	N/A	V	N/A	V	V	N/A	N/A	V	N/A	V	N/A	AprovarlAirplaneMovingLeft
CT05	Cenário 5 Avião não pode se movimentar para esquerda	V	V	N/A	N/A	V	N/A	I	N/A	V	V	N/A	N/A	V	N/A	I	N/A	Nenhuma *Rule* aprovada
CT06	Cenário 6 Avião se movimenta para a direita	V	V	N/A	N/A	N/A	V	N/A	V	V	V	N/A	N/A	N/A	V	N/A	V	AprovarlAirplaneMovingRight
	Cenário 7 Avião não pode se movimentar para direita	V	V	N/A	N/A	N/A	V	N/A	I	V	V	N/A	N/A	N/A	V	N/A	I	Nenhuma *Rule* aprovada

4.3.3.2.3 Identificar valores de casos de teste

Com base na análise de valores limite, a Tabela 48 apresenta alguns casos de teste que exercitam a *Rule* rlAirplaneShoots e as entidades que a compõem. Por exemplo, para exercitar o Caso de Teste 2 (Cenário 2: Avião dispara projétil), foram definidos valores limite válidos para os *Attributes* atLifePoints, atGameStatus, atFireButton e atBullet. Estes valores de *Attributes* permitem a aprovação das *Premisses* prAirplaneLifePointsGreaterZero, prGameStatusPlaying, prFireButtonTrue e prAirplaneHasBullet. Consequentemente, a *Condition* que avalia a conjunção destas *Premisses* também é aprovada, que resulta na aprovação da *Rule* rlAirplaneShoots. Esta *Rule* executa apenas um *Method* que cria uma entidade Bullet e duas outras *Rules* não apresentadas na tabela (rAllegroDrawBullet, rlAllegroMoveTheBulletUp).

Tabela 48 – Matriz de Casos de Teste para o Caso de Uso Controlar Avião

		Variáveis operacionais								**Resultados esperados**								
		Attributes								***Premisses***								***Rules***
ID caso de teste	**Cenário/ Condição**	**atLifePoints**	**atGameStatus**	**atBullet**	**AtFireButton**	**atLeftButton**	**atRighButton**	**atPosX1**	**atPosX2**	**prAirplaneLifePointsGreaterZero**	**prGamStatusPlaying**	**prAirplaneHasBullet**	**prFireButtonTrue**	**prLeftButtonTrue**	**prRightButtonTrue**	**prAirplaneLimLeftScreen**	**prAirplaneLimRightScreen**	
CT01	Cenário 1 Avião não realiza ação	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Nenhuma *Rule* aprovada
CT02	Cenário 2 Avião dispara projétil	1	2	V	V	N/A	N/A	N/A	N/A	V	V	V	V	N/A	N/A	N/A	N/A	**AprovarlAirplaneShoots**
CT03	Cenário 3 Avião não tem munição	1	2	I	V	N/A	N/A	N/A	N/A	V	V	I	V	N/A	N/A	N/A	N/A	Nenhuma *Rule* aprovada
CT04	Cenário 4 Avião se movimenta para esquerda	1	2	N/A	N/A	V	N/A	1	N/A	V	V	N/A	N/A	V	N/A	V	N/A	AprovarlAirplaneMovingLeft
CT05	Cenário 5 Avião não pode se movimentar para esquerda	1	2	N/A	N/A	V	N/A	-1	N/A	V	V	N/A	N/A	V	N/A	I	N/A	Nenhuma *Rule* aprovada
CT06	Cenário 6 Avião se movimenta para a direita	1	2	N/A	N/A	N/A	V	N/A	799	V	V	N/A	N/A	N/A	V	N/A	V	AprovarlAirplaneMovingRight
CT07	Cenário 7 Avião não pode se movimentar para direita	1	2	N/A	N/A	N/A	V	N/A	800	V	V	N/A	N/A	N/A	V	N/A	I	Nenhuma *Rule* aprovada

A Tabela 49 apresenta um procedimento para conhecer o número de casos de teste possíveis para exercitar os valores limite suportados pelas *Premisses* que avaliam cada *Attribute*.

Tabela 49 – Determinação do número de casos de teste possíveis com análise de valores limite

atLifePoints	atGameStatus	atFireButton	atBullet
3 valores possíveis (N/A, 0 ou 1)	4 estados possíveis (N/A, 1 (“pausado”), 2 (“jogando”) ou 3 (“parado”))	3 valores possíveis (N/A, verdadeiro ou falso)	3 valores possíveis (N/A, zero ou um)
Total de casos de teste: 3*4*3*3 = 108 casos de teste diferentes (possíveis) com valores limite			

4.3.3.2.4 Avaliação do fluxo de notificações

Para complementar a matriz de casos de teste, o critério de teste caixa branca de “análise de cobertura” é uma atividade que pode ser realizada para avaliar o fluxo de notificações do PON. Esse critério consiste em exercitar todas as notificações possíveis e, consequentemente, identificar “todos os caminhos” existentes no software PON sob teste. Um processo geral do teste caixa-branca apresentado por Copeland (2004) continua sendo válido para o PON:

- A implementação do sistema sob teste é analisada;
- Caminhos através do sistema são identificados;
- Entradas são escolhidas para que o sistema sob teste execute os caminhos selecionados. Resultados esperados para estas entradas são determinados;
- Os testes são executados;
- São comparadas as saídas produzidas com as esperadas.

Usando métodos de teste caixa-branca, é possível derivar casos de teste que (COPELAND, 2004):

- Garantam que todos os caminhos independentes de um módulo sejam executados pelo menos uma vez.
- Exercitem todas as decisões lógicas de seu lado verdadeiro e falso.

- Executem todos os ciclos (laços de repetição) nos seus limites e dentro de seus intervalos operacionais.
- Exercitem as estruturas de dados internas.

Normalmente, em sistemas procedimentais ou orientados a objetos, para realização da análise de cobertura o sistema é representado por meio de um grafo de fluxo de controle, que é uma notação gráfica composta de nós e arcos, utilizada para abstrair o fluxo de controle lógico do programa. Um nó representa uma ou mais instruções que sempre são executadas em sequência, ou seja, uma vez executada a primeira instrução de um nó todas as demais instruções daquele nó também são executadas. Um arco, também chamado de ramo ou aresta, representa o fluxo de controle entre os nós.

Para o PON, foi proposta uma adaptação do grafo de fluxo de controle tradicional para comportar as particularidades do paradigma e, com isto, recebeu a denominação de Diagrama de Fluxo de Notificações (DFN). Isto foi feito porque em PON podem ser executados fluxos de execução paralelos e existem entidades que apenas notificam outras entidades e entidades que realizam uma operação lógica antes de notificar outras entidades. As entidades que apenas notificam outras entidades são: *Attribute*, *Action* e *Instigation* (estas duas últimas não precisam ser representadas no DFN). Entidades que realizam operação lógica antes de notificar outras entidades são *Premise*, *Condition* e *SubCondition*, *Rule* e *Method* (em alguns casos).

A Figura 37 apresenta o diagrama de fluxo de notificações de entidades que implementam o caso de uso Controlar Avião. Este DFN é obtido após análise do diagrama de objetos do caso de uso apresentado na Figura 34. É possível omitir a *Condition* quando uma *Rule* possui apenas uma *Condition* e esta *Condition* realiza apenas uma operação lógica de conjunção de *Premisses*, conforme pode ser visto na *Rule* nomeada como 17(E). Esta *Rule*, por exemplo, realiza apenas uma operação de conjunção (i. e. “E”) sobre as *Premisses* 2, 4, 6 e 10 avaliadas. Alguns nós têm uma cor diferenciada porque representam nós que são notificados durante a execução do fluxo de notificações e foram destacados para facilitar a visualização no diagrama.

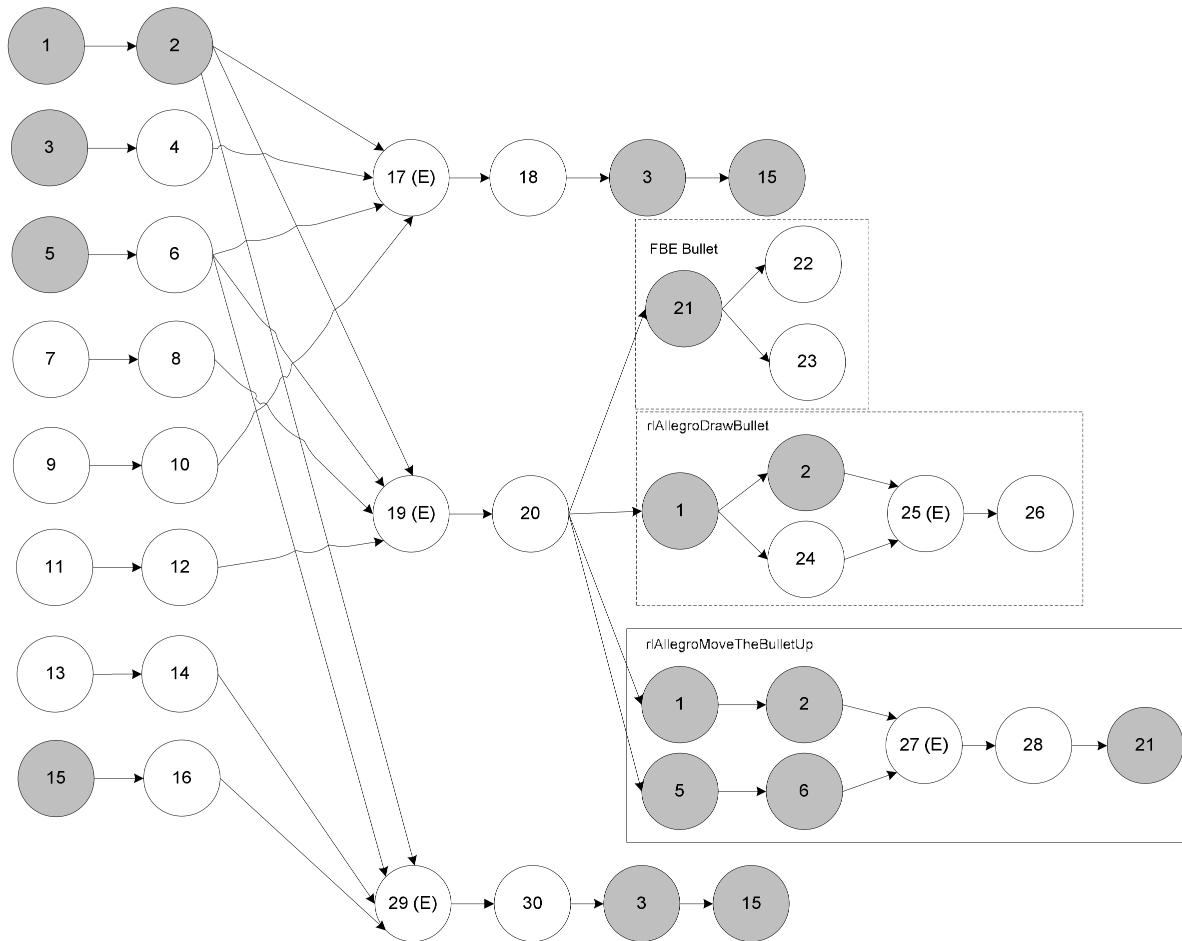


Figura 37 – Diagrama de fluxo de notificações do caso de uso Controlar Avião

Como visto, este diagrama pode conter diversas entradas e diversas saídas de acordo com as possibilidades de implementação do paradigma PON. A Tabela 50 apresenta uma proposta notação para o DFN.

Um programa PON pode ser considerado como um conjunto de várias entidades, sendo que cada entidade pode ser representada como um bloco de código (nó). Com exceção do primeiro e do último bloco, cada bloco pode ter um ou mais predecessores e um ou mais sucessores.

Tabela 50 – Notação do grafo de fluxo de notificações

Notação	Significado
1-2	O fluxo de notificações partiu do nó 1 que notifica nó 2
1-2-3	O fluxo de notificações partiu do nó 1 e notifica o nó 2 que, por sua vez, notifica o nó 3
[1-2] [3-4]	Fluxos de notificações paralelos Fluxo 1: nó 1 notifica nó 2 Fluxo 2: nó 3 notifica nó 4
1(E)	Nó 1 representa uma *Rule* que possui uma *Condition* que realiza uma operação de conjunção

Casos de teste que exercitem todos os caminhos deste DFN precisam considerar os valores que aprovam as entidades *Premise*, *Condition*, *SubCondition* e *Rule*. A Tabela 51 apresenta casos de teste para exercitar o fluxo de notificações do caso de uso Controlar Avião.

Tabela 51 – Alguns casos de teste para exercitar o fluxo de notificações do caso de uso Controlar Avião

Casos de teste	***Attribute-Premise***	***SubCondition-Condition-Rule-Method***	***Premise* (notificada pela mudança de valor de *Attribute* no *Method*)**
1	1-2		
2	3-4		
3	5-6		
4	7-8		
5	9-10		
6	[1-2] [3-4] [5-6] [9-10]	17-18-3-15	[4] [16]
7	[1-2] [5-6] [7-8] [11-12]	[21-22] [21-23] [25-26] [27-28-21]	[22]
8	[1-2] [5-6] [13-14] [15-16]	29-30-3-15	[4] [16]

Por exemplo, o caso de teste número 1 considera que o nó 1 notifica o nó 2, sendo que este nó (*Premise*) não foi aprovado e, consequentemente, não notificou nenhum outro nó (*Condition)*. O caso de teste número 3 considera a ocorrência de quatro fluxos de notificações paralelos: (1) nó 1 notifica nó 2 (*Premise)*, (2) nó 3 notifica nó 4, (3) nó 5 notifica nó 6 e (4) nó 9 notifica nó 10. O caso de teste número 6 resultou na aprovação do nó 17, que por sua vez, notificou a execução do nó 18 (*Method*), que mudou o valor dos nós 3 e 15, que resultaram nas notificações (em paralelo) dos nós 4 e 16, que não fizeram mais notificações.

Os critérios baseados em fluxo de controle, sobre o fluxo de notificações PON, foram agrupados de acordo com tipos de caminhos no DFN. Os critérios de fluxo de controle em PON são:

- **Critério Todos-Nós:** Cada nó precisa ser exercitado pelo menos uma vez, ou seja, não devem existir nós que não possam ser exercitados por pelo menos um caso de teste;
- **Critério Todos-Arcos:** Cada arco precisa ser exercitado menos uma vez, ou seja, cada desvio condicional (ou decisão) provocado por operadores lógicos precisa ser exercitado.
- **Critério Todos-Caminhos:** Todos os caminhos possíveis do diagrama de fluxo de notificações devem ser executados.

Com estes critérios de teste baseados em fluxo de controle é possível gerar centenas de casos de teste para exercitar as entidades do diagrama de fluxo de notificações apresentado na Figura 37. Para simplificar a apresentação desses critérios, nesta dissertação, seguem-se as seguintes regras para elaboração dos casos de teste para cada critério:

Todos-Nós

- Devem ser gerados casos de teste que possam executar pelo menos uma vez cada entidade do PON.

Todos-Arcos (entidades *Premise*, *SubCondition* e *Condition*)

- Cada nó correspondente a uma *Premise* deve notificar pelo menos uma vez cada uma *SubCondition* ou *Condition* que o avalia.
- Cada nó correspondente a uma *SubCondition* deve notificar pelo menos uma vez cada *Condition* que o avalia.
- Cada nó correspondente a uma *Condition* deve notificar pelo menos uma vez cada *Rule* que o avalia.

Todos-Caminhos

- Devem ser gerados casos de teste para que cada nó correspondente a um *Attribute* possa notificar todas as *Premisses* que o avaliam.
- Devem ser gerados casos de teste para que cada nó que correspondente a um *Premise* possa notificar todas as *SubConditions* e/ou *Conditions* que o avaliam.
- Devem ser gerados casos de teste para cada nó que correspondente a uma *SubCondition* possa notificar todas as *Conditions* que o avaliam.
- Devem ser gerados casos de teste para cada nó que correspondente a um *Condition* possa notificar todas as *Rules* que o avaliam.
- Devem ser gerados casos de teste para cada nó que correspondente a um *Attribute* possa executar todas as entidades que o pertencem.

Para complementar estes critérios, foi proposto ainda o critério Toda-Rede-de-Alcançabilidade que busca gerar casos de teste para exercitar todas as entidades relacionadas à alteração de valor de um *Attribute*. Este critério foi inspirado no critério Potenciais-Usos apresentado por Maldonado (1991) (apresentado na Seção 2.4.3.2.2, pg. 62), porém, ainda é empírico e experimental (ainda não validado). Passos para realização do critério Toda-Rede-de-Alcançabilidade são apresentados no APÊNDICE G, Seção 1.

Com a introdução do conceito Toda-Rede-de-Alcançabilidade, procura-se explorar todos os possíveis efeitos a partir de uma mudança de estado do programa em teste, decorrente de definição de *Attributes* em um determinado nó i. Da mesma forma, como os demais critérios baseados na análise de fluxo de controle, o critério Toda-Rede-de-Alcançabilidade pode utilizar o DFN como base para o estabelecimento dos casos de teste.

4.3.4 Avaliação dos Resultados dos Casos de Teste de Integração

Os testes de integração em PON são muito mais abrangentes que os testes unitários. Conforme visto, é possível exercitar as interfaces entre as unidades e o comportamento do software como um todo e identificar erros de definição de entidades PON. São utilizados os resultados dos testes unitários para confrontar se o fluxo de notificações está ocorrendo de acordo com o esperado tanto no teste unitário quanto no teste de integração.

Durante a execução dos casos de teste apresentados na Tabela 48, foram revelados alguns erros, que foram posteriormente corrigidos, conforme apresentado a seguir:

- *Premise*

Supondo que o programador tivesse cometido o seguinte engano ao definir a *Premise* prAirplaneLimLeftScreen, conforme apresenta a Tabela 52. Ao invés de definir a constante como 0, definiu-se com o valor de 1:

Tabela 52 – Estrutura da *Premise* prAirplaneLimRightScreen

***Premise* –** prAirplaneLimLeftScreen		
FBE	Airplane	
Attribute	Integer	atPosX1
Operador	*Greater Than*	
Constante	~~0~~ **1**	

O caso de teste CT04 revelaria esse erro, pois não aprovaria essa *Premise* e que, consequentemente, não aprovaria a *Condition* da *Rule* rlAirplaneMovingLeft. Do mesmo modo, caso o programador tivesse cometido engano ao definir outro operador lógico ou até mesmo outro *FBE* a qual pertence o *Attribute*, o caso de teste possibilitaria a revelação desses erros.

- *Condition* e *SubCondition*

Supondo que o programador tivesse cometido o seguinte engano ao definir a *Condition* pertencente à *Rule* rlAirplaneMovingRight, conforme apresenta a Tabela 53. Ao invés de declarar quatro *Premisses* para serem avaliadas pelo operador de conjunção da *Premise*, o programador declarou apenas três *Premisses*:

Tabela 53 – Estrutura de uma *Condition* que avalia quatro *Premisses*

Condition da Rule rlAirplaneMovingRight	
Operador	Conjunção
Premisses	~~Pr1 – prGameStatusPlaying~~
	Pr2 – prAirplaneLifePointsGreaterZero
	Pr3 – prAirplaneRightButtonTrue
	Pr4 – prAirplaneLimRightScreen

Caso o programador tivesse cometido o engano de não definir uma das quatro *Premisses*, o caso de teste CT06 aprovaria esta *Condition*. No entanto, a *Premise* prGameStatusPlaying não seria aprovada e caracterizaria a detecção de um erro, ou seja, a *Condition* seria aprovada em desconformidade com o caso de teste. Caso o programador tivesse cometido o engano de definir o operador lógico como disjunção, o caso de teste CT06 não revelaria este erro, porém um caso de teste que testasse especificamente essa *Condition* (teste unitário) revelaria o erro.

- *Rule*

Supondo que o programador tivesse cometido o engano de definir uma *Premise* errada para ser avaliada por uma *Condition* conforme apresenta a Tabela 54, também o caso de teste CT06 revelaria este erro. Considerando que cada *Condition* possui características exclusivas (como avaliação de *Premisses* e operador lógico), então o caso de teste CT06 não deveria aprovar a *Condition* da *Rule* rlAirplaneMovingRight, que por sua vez, revelaria esse erro.

Tabela 54 – *Rule* rlAirplaneMovingRight que avalia apenas uma *Condition*

<table>
<tr><td colspan="2">Rule – rlAirplaneMovingRight</td></tr>
<tr><td rowspan="4">Condition –
Operador Conjunção</td><td>Pr1 – prGameStatusPlaying</td></tr>
<tr><td>~~Pr2 – prAirplaneLifePointsGreaterZero~~
Pr2 – prHelicopterLifePointsGreaterZero</td></tr>
<tr><td>Pr3 – prAirplaneRightButtonTrue</td></tr>
<tr><td>Pr4 – prAirplaneLimRightScreen</td></tr>
</table>

- *Method*

Supondo que o programador tivesse cometido o engano de definir qualquer *Method* diferente de mtAirplaneAddNewBullet na *Action* da *Rule* rlAirplaneShoots. Com isso, a matriz de casos de teste revelaria esse erro, pois a *Rule* rlAirplaneShoots deveria invocar o *Method* mtAirplaneAddNewBullet, conforme visto nos testes unitários dessa *Rule*.

O diagrama de fluxo de notificações apresenta as entidades que estão contidas na definição dos *Methods*. Com isso, casos de teste devem ser criados também para exercitar todos os fluxos de notificação existentes dentro e fora de um *Method*. Ou seja, não devem existir entidades que não possam ser exercitadas nos casos de teste.

Cada entidade PON precisa ser adequadamente planejada, ter características únicas (i.e. sem duplicação) e é preciso ter uma tabela de relacionamentos que apresente o relacionamento entre cada entidade PON, para que o programador possa fazer consultas e eventualmente simular estados e comportamentos do software com relação a cada entidade acrescentada.

4.3.5 Considerações Sobre o Teste de Integração Proposto

A fase de teste de integração para aplicações PON apresentada permite gerar casos de teste de maneira planejada, organizada e com base em conceitos sólidos existentes na literatura pertinente. Com o uso deste método é possível exercitar a integração das entidades que realizam cada caso de uso por meio de critérios de teste funcional e estrutural. Desta maneira, os testes passam a ser realizados por meio de critérios técnicos e não apenas de maneira aleatória ou apenas intuitiva e restrita a alguns casos de teste cogitados no momento do desenvolvimento, como ocorre quando não se tem embasamento metodológico para sua elaboração.

A priori, não é possível tratar a resolução de conflitos e acesso concorrente em PON por meio desses critérios de teste de software apresentados. Para isto, Wiecheteck (2011) sugere a representação da explicação por meio de redes de Petri. Algumas das características de redes de Petri as tornam propícias para capturar especificações comportamentais, orientadas a objetos e concorrentes. Redes de Petri permitem a modelagem de concorrência, sincronização e compartilhamento de recursos em um sistema; além de que existem muitos resultados teóricos associados a redes de Petri para a análise comportamental, tais como detecção de bloqueio e análise de desempenho. Porém, a representação de um sistema PON por meio de redes de Petri ainda não foi plenamente validada para que possa ser utilizada adequadamente e para teste de software PON.

4.4 CONSIDERAÇÕES SOBRE O MÉTODO DE TESTE PROPOSTO

Esse capítulo propôs um método de teste de software PON para as fases de teste unitário e teste de integração. Além da descrição detalhada do método, são apresentados exemplos de aplicação utilizando como protótipo de testes o software apresentado no Capítulo 3.

O teste unitário considera que cada entidade do PON é uma unidade que possui um propósito e estrutura distinta. Com isto, foram apresentados procedimentos para geração de testes para *Premisses*, *Conditions e SubConditions*,

Rules e *FBEs, exceto Attributes, Actions* e *Instigations*, que não necessitam de testes unitários. A execução dos testes unitários é baseada no cumprimento do plano de teste e anotação dos resultados na folha de teste.

O teste de integração em PON considera o funcionamento e o comportamento de todas as entidades em conjunto para realização de casos de uso. O planejamento de teste de integração em PON com casos de uso requer a elaboração do plano de teste e identificação de cenários que precisam ser exercitados nos casos de teste. Foram apresentadas duas abordagens que se complementam para realização de teste de integração em PON: (1) Teste sobre a descrição e funcionalidades do caso de uso e (2) Teste diretamente sobre as entidades que implementam o caso de uso (como *Attributes*, *Premisses*). A execução dos testes é baseada no cumprimento do plano de teste e descrição do caso de uso, com anotação dos resultados na folha de teste.

O método de teste mostrou-se útil, é solidamente embasado em boas práticas dos principais trabalhos e literatura pertinente e, ainda, apresenta procedimentos específicos para sua elaboração. Permite detectar erros que possam ocorrer sobre o desenvolvimento de entidades PON e na interação entre as entidades por meio de notificações. Para o desenvolvimento deste método foi considerada a experiência do autor e da equipe de pesquisa no desenvolvimento de software em PON, os conceitos clássicos de teste de software e as peculiaridades do paradigma PON.

Fazendo uso deste método, os desenvolvedores e testadores terão mais um instrumento que auxilie no processo de desenvolvimento de uma aplicação PON. Além disto, foi possível compreender um pouco mais sobre o funcionamento deste paradigma com uma ótica voltada para testes.

Nota-se que as atividades de gerar de executar casos de teste requerem bastante trabalho para serem realizadas manualmente. Por isto, faz-se necessário desenvolver uma ferramenta para a realização de testes de software em PON.

5 CASO DE ESTUDO: APLICAÇÃO DO MÉTODO DE TESTE DE SOFTWARE EM PON

Este capítulo apresenta um caso de estudo que aplica o método de teste proposto no software descrito no Capítulo 3. As contribuições desse trabalho foram apresentadas no capítulo anterior e elucidadas nas descrições individuais de cada tipo de teste realizado. A Seção 5.1 apresenta o planejamento e geração de casos de teste unitário e de integração. A Seção 5.2 apresenta as considerações sobre o caso de estudo.

5.1 PLANEJAMENTO E GERAÇÃO DE CASOS DE TESTE

O planejamento e geração de casos de teste envolve a escolha do que será testado. A partir disto, é possível realizar testes que exercitem as condições que deveriam provocar ou não o seu funcionamento.

5.1.1 Teste unitário

Essa subseção apresenta a geração de casos de teste unitário para o software PON. Os documentos de requisitos e descrição das entidades PON foram suprimidos para simplificação.

5.1.1.1 Testes unitários em Premisses

A Tabela 55 apresenta todas as *Premisses* agrupadas por características semelhantes.

Tabela 55 – *Premisses* agrupadas por características semelhantes

Grupo 1) *Premise* avalia apenas um *Attribute*. **Operador lógico: EQUAL.** **Valor do *Attribute*: VERDADEIRO.**	**Grupo 2) *Premise* avalia apenas um *Attribute*** **Operador lógico: EQUAL.** **Valor do *Attribute*: FALSO.**	**Grupo 3) *Premise* avalia dois *Attributes*.** **Operador lógico: GREATER OR EQUAL.** **Valor do primeiro *Attribute* precisa ser maior ou igual ao segundo *Attribute*.**
prAirplaneCollisionTrue prAirplaneRightButtonTrue prUnpauseButtonTrue prPauseButtonTrue prAirplaneLeftButtonTrue prStopButtonTrue prAirplaneFireButtonTrue prPauseButtonTrue prHelicopterCollisionTrue prUnpauseButtonTrue prStopButtonTrue	prAirplaneLeftButtonFalse prAirplaneRightButtonFalse	prBulletXGreaterX1Airplane prBulletXGreaterX1Helicopter prBulletYGreaterY1Airplane prBulletYGreaterY1Helicopter
Grupo 4) *Premise* avalia dois *Attributes*. **Operador lógico: SMALLER OR EQUAL.** **Valor do primeiro *Attribute* precisa ser MENOR OU IGUAL ao segundo *Attribute*.**	**Grupo 5) *Premise* avalia um *Attribute*.** **Operador lógico: GREATERTHAN.** **Valor do *Attribute* precisa ser MAIOR que um valor determinado.**	**Grupo 6) *Premise* avalia um *Attribute*.** **Operador lógico: SMALLERTHAN.** **Valor do *Attribute* precisa ser MENOR que um valor determinado.**
prBulletXSmallerX2Airplane prBulletYSmallerY2Helicopter prBulletXSmallerX2Helicopter prBulletYSmallerY2Airplane	prAirplaneLimLeftScreen prHelicopterLifePointsGreaterZero prHelicopterTimeToShoot	prAirplaneLimRightScreen
Grupo 7) *Premise* avalia um *Attribute*. **Operador lógico: SMALLEROREQUAL.** **Valor do *Attribute* precisa ser MENOR OU IGUAL que um valor determinado.**		
prAirplaneLifePointsZero prHelicopterLifePointsZero		

A Tabela 56 apresenta a determinação de classes de equivalência e análise de valor limite para *Premisses* do Grupo 1 e a Tabela 57 apresenta a geração de alguns casos de teste para esse grupo. De modo análogo, é possível gerar casos de teste para o Grupo 2, levando em conta o valor do *Attribute* considerado.

Tabela 56 – Classes de equivalência e análise de valor limite para a *Premise* prPauseButtonTrue

***Premise* –** prPauseButtonTrue			
FBE	Allegro_Keyboard		
Attribute	**Boolean**	atPauseButton	
Operador	*Equal*		
Constante	True		
Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
True	False	True	False

Tabela 57 – Casos de teste para a *Premise* prPauseButtonTrue

Casos de teste	Valor de atPauseButton	Saída esperada ou comportamento esperado
1	True	Aprova a *Premise*
2	False	Não aprova a *Premise*

A Tabela 58 apresenta a determinação de classes de equivalência e análise de valor limite para *Premisses* do Grupo 3, ou seja, para uma *Premise* que avalia dois *Attributes* e o operador lógico é “Greater Or Equal”. A Tabela 59 apresenta um exemplo de geração de casos de teste para esse grupo. De modo análogo, é possível gerar casos de teste para o Grupo 4, ou seja, uma *Premise* que avalie dois *Attributes* e que seu operador lógico é “Smaller Or Equal”. Deve-se, apenas, modificar o operador lógico e gerar casos de teste compatíveis.

Tabela 58 – Classes de equivalência e análise de valor limite para a *Premise* prBulletXGreaterX1Airplane

Premise	prBulletXGreaterX1Airplane		
FBE	Bullet	Airplane	
Attribute	**Integer**	atPosX	
Attribute	**Integer**	atPosX1	
Operador	*Greater Or Equal*		
Valor	atPosX>=atPosX1		
Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
atPosX>=atPosX1	atPosX<atPosX1	Refere-se ao valor de atPosX, número igual a atPosX1	Refere-se ao valor de atPosX, primeiro número menor que atPosX1

Tabela 59 – Casos de teste para a *Premise* prBulletXGreaterX1Airplane

AtPosX	atPosX1	Saída esperada ou comportamento esperado
-1	0	Não aprova a *Premise*
1	1	Aprova a *Premise*
2	1	Aprova a *Premise*
1	2	Não aprova a *Premise*

A Tabela 60 apresenta um exemplo de determinação de classes de equivalência e análise de valor limite para *Premisses* do Grupo 5, ou seja, para uma *Premise* que avalia apenas um *Attribute* e o operador lógico é “Greater Than”. A Tabela 61 apresenta a geração de casos de teste para este grupo. De modo análogo, é possível gerar casos de teste para o Grupo 6, ou seja, uma *Premise* que avalie um *Attribute* e que seu operador lógico é “Smaller Than”. Deve-se, apenas modificar o operador lógico e gerar casos de teste compatíveis.

Tabela 60 – Classes de equivalência e análise de valor limite para a *Premise* prAirplaneLimLeftScreen

Premise – prAirplaneLimLeftScreen			
FBE	Airplane		
Attribute	**Integer**	atPosX1	
Operador	*Greater Than*		
Constante	0		
Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
atPosX1>0	atPosX1<=0	1	0

Tabela 61 – Casos de teste para a *Premise* prAirplaneLimLeftScreen

Casos de teste	Valor de atPosX1	Saída esperada ou comportamento esperado
1	1	Aprova a *Premise*
2	0	Não aprova a *Premise*
3	2	Aprova a *Premise*
4	-1	Não aprova a *Premise*

A Tabela 62 apresenta a determinação de classes de equivalência e análise de valor limite para o Grupo 7, ou seja, para uma *Premise* que avalia apenas um *Attribute* e o operador lógico é “Greater Than”. A Tabela 63 apresenta a geração de casos de teste para este grupo. De modo análogo, é possível gerar casos de teste para o Grupo 6, ou seja, uma *Premise* que avalie um *Attribute* e que seu operador lógico é “Smaller Than”. Deve-se, apenas, modificar o operador lógico e gerar casos de teste compatíveis.

Tabela 62 – Classes de equivalência e análise de valor limite para a *Premise* prHelicopterLifePointsZero

***Premise* –** prHelicopterLifePointsZero			
FBE	Helicopter		
Attribute	**Integer**	atLifePoints	
Operador	*Smaller or Equal*		
Constante	0		
Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
atLifePoints<=0	atLifePoints>0	1	0

Tabela 63 – Casos de teste para a *Premise* prAirplaneLimLeftScreen

Casos de teste	Valor de atPosX1	Saída esperada ou comportamento esperado
1	0	Aprova a *Premise*
2	1	Não aprova a *Premise*
3	-1	Aprova a *Premise*
4	2	Não aprova a *Premise*

Considerando que podem ser gerados no mínimo 4 casos de teste com análise de valor limite e determinação de classes de equivalência para cada Premisse, nota-se que podem ser criados 112 casos de teste para as 28 *Premisses* da aplicação apresentada no Capítulo 3. O código fonte de todas as *Premisses* está disponível no APÊNDICE E.

5.1.1.2 Testes unitários em SubConditions e Conditions

A Tabela 64 apresenta uma relação das *Conditions* e *SubConditions* agrupadas por características semelhantes.

Tabela 64 – *Conditions* e *SubConditions* agrupadas por características semelhantes

Grupo 1) *Condition* que avalia uma conjunção de *Premisses*	**Grupo 2) *Condition* que avalia uma disjunção de *Premisses***	**Grupo 3) *Condition* que avalia uma conjunção de *SubConditions***
rlScenarioMoving rlAirplaneMoving rlAirplaneMovingLeft rlAirplaneMovingRight rlAirplaneShoots rlHelicopterShoots rlAirplaneDecreaseLifePoints rlHelicopterDecreaseLifePoints rlHelicopterDies rlPlayerGameOver rlGamePause rlGameUnpause rlAllegroHelicopterCollision rlAllegroAirplaneCollision rlAllegroMovingBullet rlAllegroMovingBullet	rlProgressUpdating rlAllegroClear rlAllegroBlit rlAllegroKeyboard rlScenarioDrawing	rlAllegroDrawHelicopter rlGameStop rlAllegroDrawAirplane rlAllegroDrawBullet rlHelicopterMoving
Grupo 4) *Condition* que avalia uma disjunção de *SubConditions*	**Grupo 5) *SubCondition* que avalia conjunção de *Premisses***	**Grupo 6) *SubCondition* que avalia disjunção de *Premisses***
Não tem.	rlAllegroDrawHelicopter rlGameStop rlAllegroDrawAirplane	rlAllegroDrawHelicopter rlGameStop rlAllegroDrawAirplane
Grupo 7) *Condition* que avalia uma conjunção contendo *Premise* impertinente	**Grupo 8) *SubCondition* que avalia uma disjunção contendo *Premise* impertinente**	
Não tem.	Não tem.	

A Tabela 65 apresenta a *Condition* pertencente à *Rule* rlAirplaneMovingRight (Grupo 1) que avalia uma conjunção de quatro *Premisses*. A Tabela 66 apresenta a geração de alguns casos de teste para esta *Condition*. De maneira análoga, ocorre a geração de casos de teste para uma *SubCondition* que avalia uma conjunção de *Premisses* (Grupo 5).

Tabela 65 – *Condition* pertencente a *Rule* rlAirplaneMovingRight

Condition* da *Rule rlAirplaneMovingRight	
Operador	Conjunção
Premisses	Pr1 – prGameStatusPlaying
	Pr2 – prAirplaneLifePointsGreaterZero
	Pr3 – prAirplaneRightButtonTrue
	Pr4 – prAirplaneLimRightScreen

Tabela 66 – Alguns casos de teste para *Condition* que pertence a *Rule* rlAiplaneMovingRight

Casos de teste	Estado Pr1	Estado Pr2	Estado Pr3	Estado Pr4	Saída esperada ou comportamento esperado
1	Aprovada	Aprovada	Aprovada	Aprovada	Aprova a *Condition*
2	Aprovada	Aprovada	Aprovada	Não aprovada	Não aprova a *Condition*
3	Aprovada	Aprovada	Não aprovada	Não aprovada	Não aprova a *Condition*
4	Aprovada	Não aprovada	Não aprovada	Não aprovada	Não aprova a *Condition*
5	Não aprovada	Não aprovada	Não aprovada	Não aprovada	Não aprova a *Condition*

A Tabela 67 apresenta uma *Condition* que avalia uma disjunção de duas *Premisses* pertencente ao Grupo 2. A Tabela 68 apresenta a geração de casos de teste para esta *Condition*. De maneira análoga ocorre a geração e execução de casos de teste para uma *SubCondition* que avalia uma disjunção de *Premisses*.

Tabela 67 – *Condition* pertencente a *Rule* rlScenarioDrawing

Condition da *Rule rlScenarioDrawing*	
Operador	Disjunção
Premisses	Pr1 **–** prGameStatusPlaying
	Pr2 **–** prGameStatusPaused

Tabela 68 – Casos de teste para a *Condition* da *Rule* rlScenarioDrawing

Casos de teste	Estado prGameStatusPlaying	Estado prGameStatusPaused	Saída esperada ou comportamento esperado
1	Aprovada	Aprovada	Aprova a *Condition*
2	Aprovada	Não aprovada	Aprova a *Condition*
3	Não aprovada	Aprovada	Aprova a *Condition*
4	Não aprovada	Não aprovada	Não aprova a *Condition*

5.1.1.3 Testes unitários em Rules

A Tabela 69 apresenta a *Rule* rlAirplaneMovingRight e sua única *Condition* que avalia a conjunção de quatro *Premisses*. A Tabela 70 apresenta a geração de casos de teste para esta *Rule*. Como qualquer *Rule,* apenas avalia o estado de sua única *Condition*. Então presume-se que podem ser gerados os mesmos casos de teste para qualquer uma delas.

Tabela 69 – *Rule* que avalia apenas uma *Condition*

***Rule* –** rlAirplaneMovingRight	
Condition	*Operador de Conjunção*
Premisses	Pr1 **–** prGameStatusPlaying
	Pr2 **–** prAirplaneLifePointsGreaterZero
	Pr3 **–** prAirplaneRightButtonTrue
	Pr4 **–** prAirplaneLimRightScreen

Tabela 70 – Casos de teste para a *Rule* rlAirplaneMovingRight

Casos de teste	**Estado *Condition***	**Saída esperada ou comportamento esperado**
1	Aprovada	Aprova a *Rule*
2	Não aprovada	Não aprova a *Rule*

Considerando que cada *Rule* avalia apenas uma *Condition* e que existem 26 *Rules*, no mínimo podem ser gerados 52 casos de teste para as *Conditions* (dois casos de teste para cada *Rule*, sendo um com a *Condition* aprovada e um com a *Condition* não aprovada).

5.1.1.4 Testes unitários em FBEs

Foram identificados 5 *FBEs* testáveis na aplicação desenvolvida: Airplane, Helicopter, Stage 1 e Allegro Bullet e Allegro Keyboard.

A maioria dos *Methods* desta aplicação é bastante simples, alteram apenas um ou dois *Attributes.* A Tabela 71 apresenta as ações que os *Methods* simples do *FBE* Airplane realizam. A Tabela 72 apresenta as ações que o *Methods* simples do *FBE* Helicopter realizam. A Tabela 73 apresenta as ações que o *Methods* simples do *FBE* Stage 1 realizam. A Tabela 74 apresenta as ações que o *Methods* simples do *FBE* Helicopter realizam.

Tabela 71 – *Methods* do *FBE* Airplane

Method	**Ação**
mtMoveLeft	(atPosX1) <- (atPosX1)-1 (atPosX2) <- (atPosX2)-1
mtMoveRight	(atPosX1) <- (atPosX1)+1 (atPosX2) <- (atPosX2)+1
mtStayInPosition	(atPosX1) <- (atPosX1) (atPosX2) <- (atPosX2)
mtAttack	atFireButton <- True
mtDraw	*Uma linha de instrução do Allegro para mostrar uma imagem na tela.*

Tabela 72 – *Methods* do *FBE* Helicopter

Method	**Ação**
mtMovement	Função aleatória para movimentar o Helicóptero
mtAttack	atFireButton <- True
mtDraw	*Uma linha de instrução do Allegro para mostrar uma imagem na tela.*

Tabela 73 – *Methods* do *FBE* Stage 1

Method	**Ação**
mtAllegroBlit	*Uma linha de instrução do Allegro para mostrar um quadro de imagem na tela*
mtAllegroClear	*Uma linha de instrução do Allegro para apagar um quadro de imagem na tela*

Tabela 74 – *Methods* do *FBE* Allegro Bullet

Method	**Ação**
mtDraw	*Uma linha de instrução do Allegro para mostrar um quadro de imagem na tela*
mtMovingDown	(atPosY) <- (atPosY)+1
mtMovingUp	(atPosY) <- (atPosY)-1

Os dois *Methods* mais complexos da aplicação são mtAirplaneAddNewBullet (que já foi previamente apresentado na Seção 4.2.2.5) e mtAirplaneAddNewBullet. A Tabela 75 apresenta um caso de teste para o *Method* mtHelicopterAddNewBullet. A Tabela 76 apresenta a determinação de classes de equivalência e análise de valor limite para o *Method* mtHelicopterAddNewBullet e a Tabela 77 apresenta a geração de alguns casos de teste para este *Method*.

Tabela 75 – Caso de teste do *Method* mtHelicopterAddNewBullet

Identificador do caso de teste testcase_mtHelicopterAddNewBullet1
Objetivo do caso de teste Este caso de teste busca exercitar o *Method* mtHelicopterAddNewBullet. Tem como base o Plano de Teste e objetiva verificar a seguinte condição: 1. Parâmetros de entrada que produzam um projétil na tela.
Itens testáveis *Attributes* dos parâmetros de entrada. Comportamento do *Method* mtHelicopterAddNewBullet.
Especificações de entrada Código do *Method* mtHelicopterAddNewBullet. Configuração de valores para os *Attributes* atTimeToShoot e atGameStatus.
Resultados de teste esperados Criação de um projétil na tela. Criação de entidades PON que são utilizadas para aprovação das *Rules*.
Necessidades de ambiente IDE Microsoft Visual Studio C++ 2010. Depurador do Microsoft Visual Studio C++ 2010. *Framework* PON Otimizado.
Requisitos procedimentais específicos **Os procedimentos para execução dos casos de teste estão disponíveis na** Tabela 76.
Dependências Não tem.

Tabela 76 – Classes de equivalência e análise de valor limite para exercitar mtHelicopterAddNewBullet

***Method* –** mtHelicopterAddNewBullet			
Parâmetros de entrada	*Integer*	*atTimeToShoot*	
	Integer	atGameStatus	
	Integer	atLifePoints	
Ação	Executar casos de testes que exercitem mtAirplaneAddNewBullet		
Classes de equivalência válidas	**Classes de equivalência inválidas**	**Valor limite válido**	**Valor limite inválido**
atTimeToShoot>=2	atTimeToShoot<2	2	1
atGameStatus=2	atGameStatus<>2	2	1; 3
atLifePoints	atLifePoints>0	1	0

Tabela 77 – Casos de teste para o *Method* mtHelicopterAddNewBullet

Casos de Teste	atLifePoints	atTimeToShoot	atGameStatus	Saída
1	1	2	2	Apresenta um projétil na tela
2	1	1	1	Não apresenta projétil na tela

5.1.2 Testes de Integração

Nesta seção são apresentados os documentos de descrição de cada caso de uso, determinação de classes de equivalência e análise de valores limite e matriz de casos de teste. Também, são apresentados os erros identificados no software durante a execução dos casos de teste.

5.1.2.1 Testes de integração no caso de uso Realizar Leitura de Teclado

A Tabela 78 apresenta a descrição do caso de uso Realizar Leitura de Teclado. A Tabela 79 apresenta a determinação de classes de equivalência e análise de valor limite para as variáveis operacionais identificados para este caso de uso. A Tabela 80 apresenta a matriz de casos de teste com valores válidos, inválidos e não avaliados para cada variável operacional identificada.

Tabela 78 – Descrição do caso de uso Realizar Leitura de Teclado

ID Caso de Uso	UC1		
Nome do Caso de Uso	Realizar Leitura de Teclado		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Jogador.		
Descrição	Este caso de uso realiza a leitura dos comandos fornecidos pelo teclado e toma as devidas providências.		
Disparador	O caso de uso inicia quando o jogador iniciar o jogo.		
Pré-condições	O jogo precisa estar em estado “jogando” ou “pausado”.		
Pós-condições	Pós-condição 1) O sistema deverá reconhecer as teclas pressionadas e tomar as ações devidas.		
Fluxo básico	1. O jogo fica com estado jogando ou pausado. 2. O sistema aguarda o jogador pressionar algum botão.		
Inclusões	O caso de uso Realizar Leitura de Teclado não inclui outros Casos de Uso.		
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando ou pausado.		
Requisitos	Os requisitos funcionais implementados total ou parcialmente são: RF002, RF008, RF012, RF015, RF016. Os requisitos não realizados funcionais implementados total ou parcialmente são: RNF001, RNF004, RNF005.		
Considerações	Considera-se que: • As teclas que serão lidas são: Enter, Esc, Espaço, Seta Direita e Seta Esquerda.		

Tabela 79 – Classes de equivalência e análise de valores limite para o caso de uso Realizar Leitura de Teclado

	Valores limite		Classes de equivalência	
Condições ou Variáveis Operacionais	Válido	Inválido	Válidas	Inválidas
Botão esquerdo	Verdadeiro	Falso	Verdadeiro	Falso
Botão direito	Verdadeiro	Falso	Verdadeiro	Falso
Botão de ataque	Verdadeiro	Falso	Verdadeiro	Falso
Botão de pausa	Verdadeiro	Falso	Verdadeiro	Falso
Botão para continuar jogo pausado	Verdadeiro	Falso	Verdadeiro	Falso
Botão de parar o jogo	Verdadeiro	Falso	Verdadeiro	Falso

Tabela 80 – Matriz de Casos de Teste com valores (caso de uso Realizar Leitura de Teclado)

ID caso de teste	Cenário/ Condição	Botão esquerdo	Botão direito	Botão de ataque	Botão de pausa	Botão para continuar	Botão parar	Resultado esperado
CT01	Cenário 1 Jogador pressiona botão esquerdo	V	N/A	N/A	N/A	N/A	N/A	O sistema retonar parar o caso de uso pertinente
CT02	Cenário 2 Jogador pressiona botão Direito	N/A	V	N/A	N/A	N/A	N/A	O sistema retonar parar o caso de uso pertinente
CT03	Cenário 3 Jogador pressiona botão de ataque	N/A	N/A	V	N/A	N/A	N/A	O sistema retonar parar o caso de uso pertinente
CT04	Cenário 4 Jogador pressiona botão de pausa	N/A	N/A	N/A	V	N/A	N/A	O sistema retonar parar o caso de uso pertinente
CT05	Cenário 5 Jogador pressiona botão para continuar jogo pausado	N/A	N/A	N/A	N/A	V	N/A	O sistema retonar parar o caso de uso pertinente
CT06	Cenário 6 Jogador pressiona botão para parar o jogo	N/A	N/A	N/A	N/A	N/A	V	O sistema retonar parar o caso de uso pertinente

A Figura 38 apresenta o diagrama de objetos do caso de uso Realizar Leitura de Teclado. A Tabela 81 apresenta uma relação entre as variáveis operacionais e os *Attributes* deste caso de uso. A Tabela 82 apresenta a matriz de casos de teste com valores que exercitem diretamente as entidades deste caso de uso.

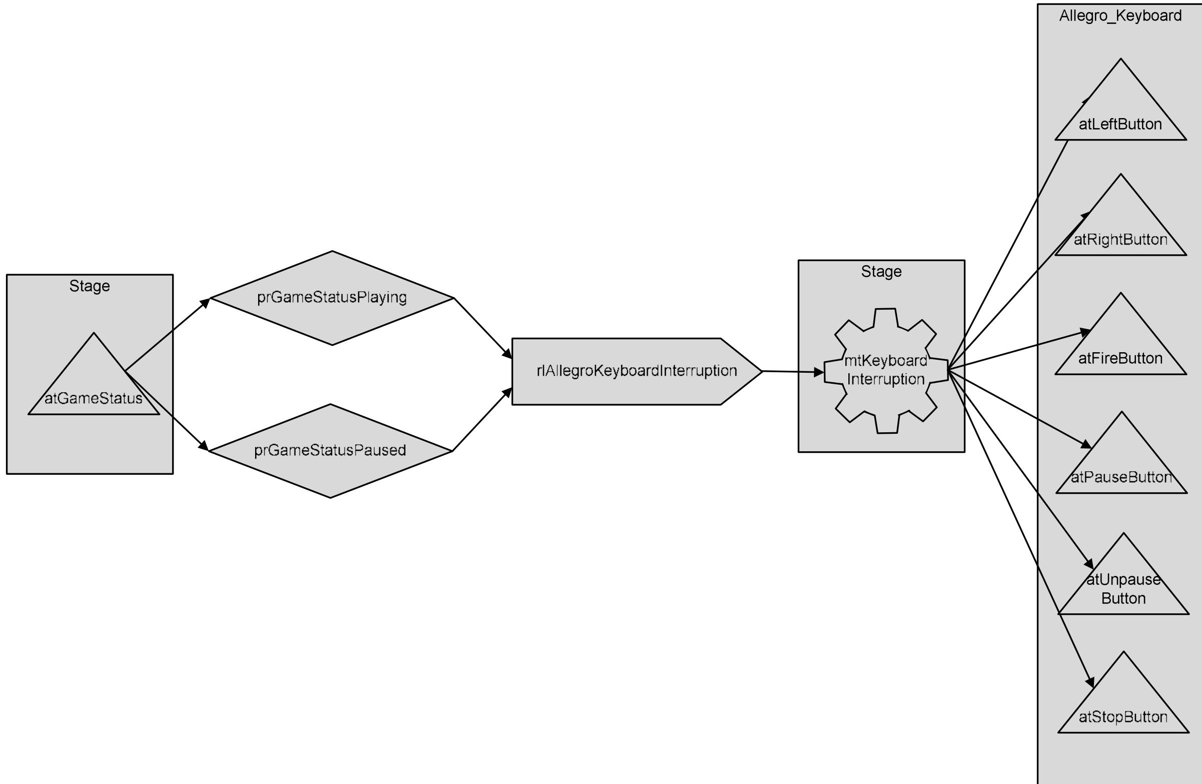


Figura 38 – Diagrama de objetos do caso de uso Realizar Leitura de Teclado

Tabela 81 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Realizar Leitura de Teclado

Variáveis operacionais	*Attributes*
Botão esquerdo	atLeftButton
Botão direito	atRightButton
Botão de ataque	atFireButton
Botão de pausa	atPauseButton
Botão para continuar jogo pausado	atUnpauseButton
Botão de parar o jogo	atStopButton

Tabela 82 – Matriz de Casos de Teste para o Caso de Uso Realizar Leitura de Teclado

		Variáveis operacionais						Resultados esperados
		Attributes						
ID caso de teste	**Cenário/ Condição**	**atLeft Button**	**atRight Button**	**atFireB utton**	**atPause Button**	**atUnpau seButton**	**atStopBu tton**	
CT01	Cenário 1 Jogador pressiona botão esquerdo	V	N/A	N/A	N/A	N/A	N/A	Retorna ao caso de uso Controlar Avião
CT02	Cenário 2 Jogador pressiona botão Direito	N/A	V	N/A	N/A	N/A	N/A	**Retorna ao caso de uso Controlar Avião**
CT03	Cenário 3 Jogador pressiona botão de ataque	N/A	N/A	V	N/A	N/A	N/A	Retorna ao caso de uso Controlar Avião
CT04	Cenário 4 Jogador pressiona botão de pausa	N/A	N/A	N/A	V	N/A	N/A	Retorna ao caso de uso Pausar e Continuar Jogo
CT05	Cenário 5 Jogador pressiona botão para continuar jogo pausado	N/A	N/A	N/A	N/A	V	N/A	Retorna ao caso de uso Pausar e Continuar Jogo
CT06	Cenário 6 Jogador pressiona botão para parar o jogo	N/A	N/A	N/A	N/A	N/A	V	Retorna ao caso de uso Parar Jogo

A execução do caso de teste CT02 revelou um erro. Ao pressionar o botão direito não era retornado para o caso de teste Controlar Avião. Esse erro foi corrigido.

5.1.2.2 Testes de integração no caso de uso Controlar Avião

A descrição e apresentação de testes de integração para o caso de uso Controlar Avião foram apresentados nas seções 4.3.3.1 e 4.3.3.2.

5.1.2.3 Testes de integração no caso de uso Controlar Helicóptero

A Tabela 83 apresenta a descrição do caso de uso Controlar Helicóptero. A Tabela 84 apresenta a determinação de classes de equivalência e análise de valor limite para as variáveis operacionais identificados para este caso de uso. A Tabela 85 apresenta a matriz de casos de teste com valores válidos, inválidos e não avaliados para cada variável operacional identificada. A Figura 39 apresenta o diagrama de objetos. A Tabela 86 apresenta uma relação entre as variáveis operacionais e os *Attributes* deste caso de uso. A Tabela 87 apresenta a matriz de

casos de teste com valores que exercitem diretamente as entidades deste caso de uso.

Tabela 83 – Descrição do caso de uso Controlar Helicóptero

ID Caso de Uso	UC3		
Nome do Caso de Uso	Controlar Helicóptero		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Não tem.		
Descrição	Este caso de uso controla as funções de ataque e movimentação do helicóptero.		
Disparador	O caso de uso inicia quando o jogador iniciar o jogo.		
Pré-condições	O jogo precisa estar em estado jogando.		
Pós-condições	Pós-condição 1) O helicóptero movimenta-se para baixo. Pós-condição 2) O helicóptero ataca.		
Fluxo básico	1. O jogo fica com estado jogando. 2. O sistema movimenta o helicóptero de cima para baixo. 3. O sistema aguarda o tempo para disparar o helicóptero ser igual ou superior a 2 segundos.		
[Fluxo Alternativo 1]	O helicóptero ataca. 1. No passo 3 do fluxo básico o tempo de disparo do helicóptero é igual a 2 segundos. 2. O sistema verifica se o estado de jogo é “jogando”. 3. O sistema verifica se o helicóptero possui mais que zero de pontos de vida. 4. O sistema gera um novo projétil na tela a partir da posição do helicóptero. 5. O sistema permite que o projétil seja desenhado na tela. 6. O sistema permite que o projétil siga uma trajetória em linha reta para baixo na tela. 7. O sistema continua no passo 2 do fluxo básico.		
Exceções	Não tem.		
Inclusões	O caso de uso Controlar Helicóptero não inclui outros Casos de Uso.		
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando.		
Requisitos	Os requisitos funcionais implementados total ou parcialmente são: RF004, RF005. Os requisitos não realizados funcionais implementados total ou parcialmente são: RNF006.		
Considerações	Considera-se que: • O helicóptero não pode se movimentar ou disparar projétil quando seus pontos de vida forem menores ou iguais a zero.		

Tabela 84 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Controlar Helicóptero

	Valores limite		Classes de equivalência	
Condições ou Variáveis Operacionais	Válido	Inválido	Válidas	Inválidas
Ter pontos de vida	1	0	Pontos de vida>0	Pontos de vida<=0
Estado de jogo jogando	2	1; 3	2	Estados diferentes de 2
Tempo para disparar	2	1; 3	Tempo p/ disparar>=2	Tempo p/ disparar<2

Tabela 85 – Matriz de Casos de Teste com valores (caso de uso Controlar Helicóptero)

ID caso de teste	Cenário/ Condição	Pontos de vida	Estado de jogo	Tempo para disparar	Resultado esperado
CT01	Cenário 1 Helicóptero ataca	1	2	2	Cria um projétil; Cria *Rule* para desenhar; Cria *Rule* para movimentar
CT02	Cenário 2- Helicóptero se movimenta	1	2	N/A	Movimenta helicóptero de cima para baixo

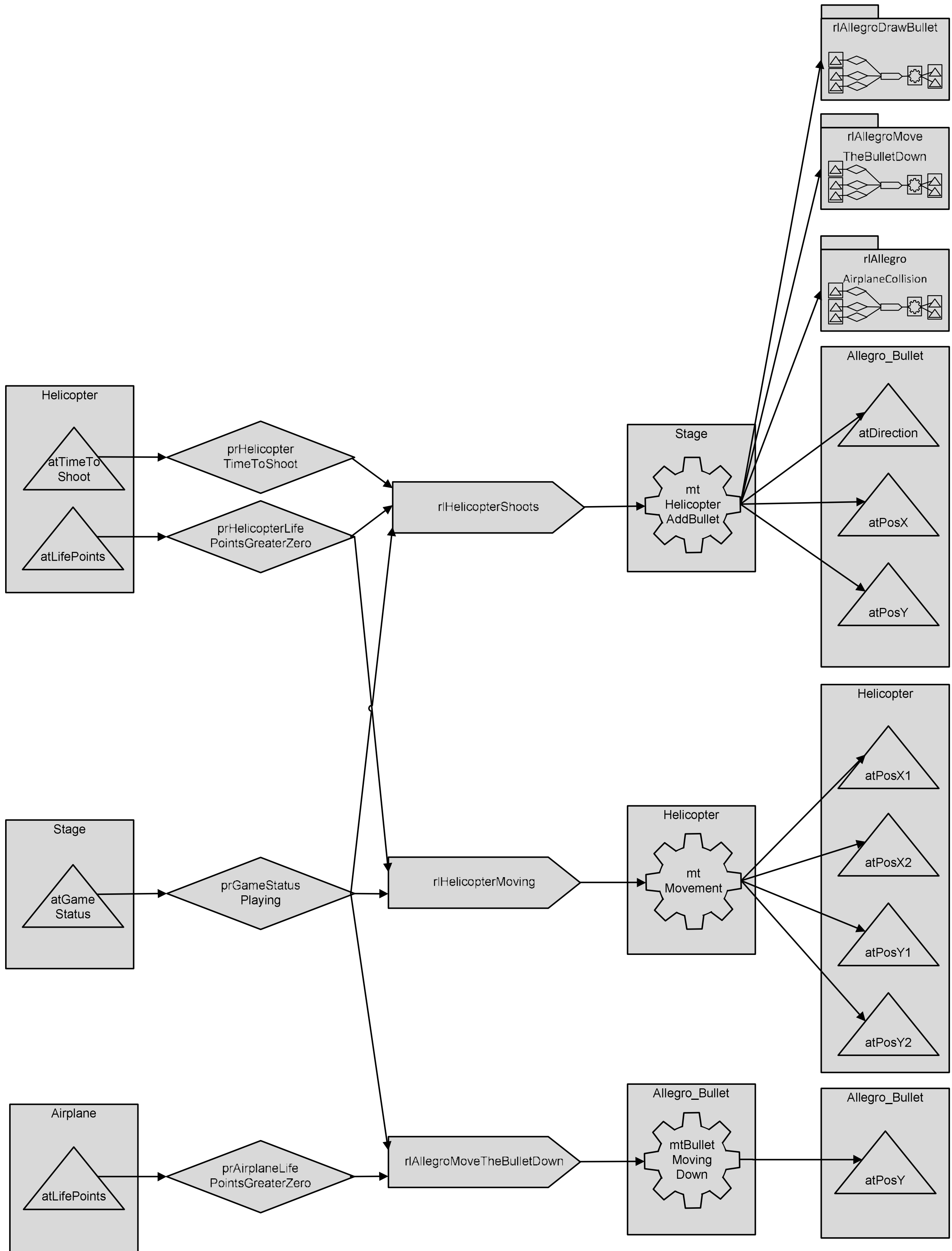


Figura 39 – Diagrama de objetos do caso de uso Controlar Helicóptero

Tabela 86 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Controlar Helicóptero

Variáveis operacionais	*Attributes*
Pontos de vida	atLifePoints
Estado de jogo	atGameStatus
Tempo para disparar	atTimeToShoot

Tabela 87 – Matriz de Casos de Teste para o Caso de Uso Controlar Helicóptero

		Variáveis operacionais				Resultados esperados				
		Attributes				*Premisses*				*Rule*
ID caso de teste	**Cenário/ Condição**	**atTime ToSho ot**	**atLife-Points**	**atGame Status**	**atLife-Points (airplane)**	**prHelic opterTi meToS hoot**	**prHelico pterLifeP ointsGre aterZero**	**prGame StatusP laying**	**prAirpl aneLife Points Greater Zero**	
CT01	Cenário 1 Helicóp-tero ataca	2	2	2	1	V	V	V	V	Aprova a *Rule* rlHelicopterShoots
CT02	Cenário 2- Helicóp-tero se movi-menta	2	2	2	1	V	V	V	V	Aprova a *Rule* rlHelicopterMovem ent

A Tabela 88 apresenta a determinação do número de casos de teste mínimo que pode ser gerado para exercitar diretamente as entidades que compõem este caso de uso.

Tabela 88 – Determinação do número de casos de teste possíveis com análise de valores limite

atGameStatus	atTimeToShoot	atLifePoints (avião)	atLifePoints (helicóptero)
4 estados possíveis (N/A, 1 (pausado), 2 (jogando) ou 3 (parado))	4 valores possíveis (N/A, 1, 2 ou 3)	3 valores possíveis (N/A, 0 ou 1)	3 valores possíveis (N/A, 0 ou 1)
Total de casos de teste: 4*4*3*3 = 144 casos de teste diferentes (possíveis) apenas com valores limite			

A execução do caso de teste CT01 revelou um erro na definição da *Premise* prHelicopterTimeToShoot. Ao invés da *Premise* avaliar o *Attribute* atTimeToShoot, estava sendo avaliado o *Attribute* atLifePoints. Esse erro foi corrigido.

5.1.2.4 Testes de integração no caso de uso Controlar Apresentação

A Tabela 89 apresenta a descrição do caso de uso Controlar Apresentação. A Tabela 88 apresenta a determinação de classes de equivalência e análise de valor limite para as variáveis operacionais identificados para este caso de uso. A Tabela

91 apresenta a matriz de casos de teste com valores válidos, inválidos e não avaliados para cada variável operacional identificada. A Figura 40 apresenta o diagrama de objetos. A Tabela 92 apresenta uma relação entre as variáveis operacionais e os *Attributes* deste caso de uso. A Tabela 93 apresenta a matriz de casos de teste com valores que exercitem diretamente as entidades deste caso de uso.

Tabela 89 – Descrição do caso de uso Controlar Apresentação

ID Caso de Uso	UC4		
Nome do Caso de Uso	Controlar Apresentação		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Não tem.		
Descrição	Este caso de uso controla as funções da biblioteca Allegro como desenhar/redesenhar os personagens e mostrar a tela de jogo.		
Disparador	O caso de uso inicia quando o jogador iniciar o jogo.		
Pré-condições	O jogo precisa estar em estado jogando ou pausado.		
Pós-condições	Pós-condição 1) Deverá ser desenhado um frame com todos os personagens e cenário Pós-condição 2) Deverá ser apagado o frame.		
Fluxo básico	1. O jogo fica com estado jogando ou pausado. 2. O sistema continuamente apaga e redesenha o cenário e os personagens 3. Retorna para o passo 1.		
[Fluxo Alternativo 1]	O jogo está em estado de jogo jogando. 1. No passo 1 do fluxo básico, se o estado de jogo é jogando, o cenário deve movimentar-se de cima para baixo. 2. O caso de uso continua no passo 2 do fluxo básico.		
Exceções	Exceção 1) O personagem não pode ser desenhado 1. No passo 2 do fluxo básico, se o personagem (avião ou helicóptero) não tiver mais pontos de vida, não pode ser desenhado na tela. 2. O caso de uso continua no passo 3 do fluxo básico.		
Inclusões	O caso de uso Controlar Apresentação não inclui outros Casos de Uso.		
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando.		
Requisitos	Os requisitos funcionais implementados total ou parcialmente são: RF003, RF014, RF015. Os requisitos não realizados funcionais implementados total ou parcialmente são: RNF002, RNF003.		
Considerações	Considera-se que: • Desenhar é o ato de mostrar um personagem ou o cenário • Apagar é a ação de escurecer a tela para que possa permitir o redesenho.		

Tabela 90 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Controlar Apresentação

	Valores limite		Classes de equivalência	
Condições ou Variáveis Operacionais	Válido	Inválido	Válidas	Inválidas
Avião ter pontos de vida	1	0	Pontos de vida>0	Pontos de vida<=0
Helicóptero ter pontos de vida	1	0	Pontos de vida>0	Pontos de vida<=0
Estado de jogo “jogando”	2	1;3	2	Estados diferentes de 2
Estado de jogo “pausado”	1	2; 3	1	Estados diferentes de 1

Tabela 91 – Matriz de Casos de Teste com valores (caso de uso Controlar Apresentação)

ID caso de teste	Cenário/ Condição	Ponto de vida avião	Pontos de vida helicóptero	Estado de jogo jogando	Estado de jogo pausado	Resultado esperado
CT01	Cenário 1 Desenhar avião	1	N/A	V	N/A	Executa *Method* mtDraw (Airplane)
CT02	Cenário 2 Desenhar helicóptero	N/A	1	V	N/A	Executa *Method* mtDraw (Helicopter)
CT03	Cenário 3 Redesenha a tela	N/A	N/A	N/A	V	Executa *Method* mtDraw (Scenario)

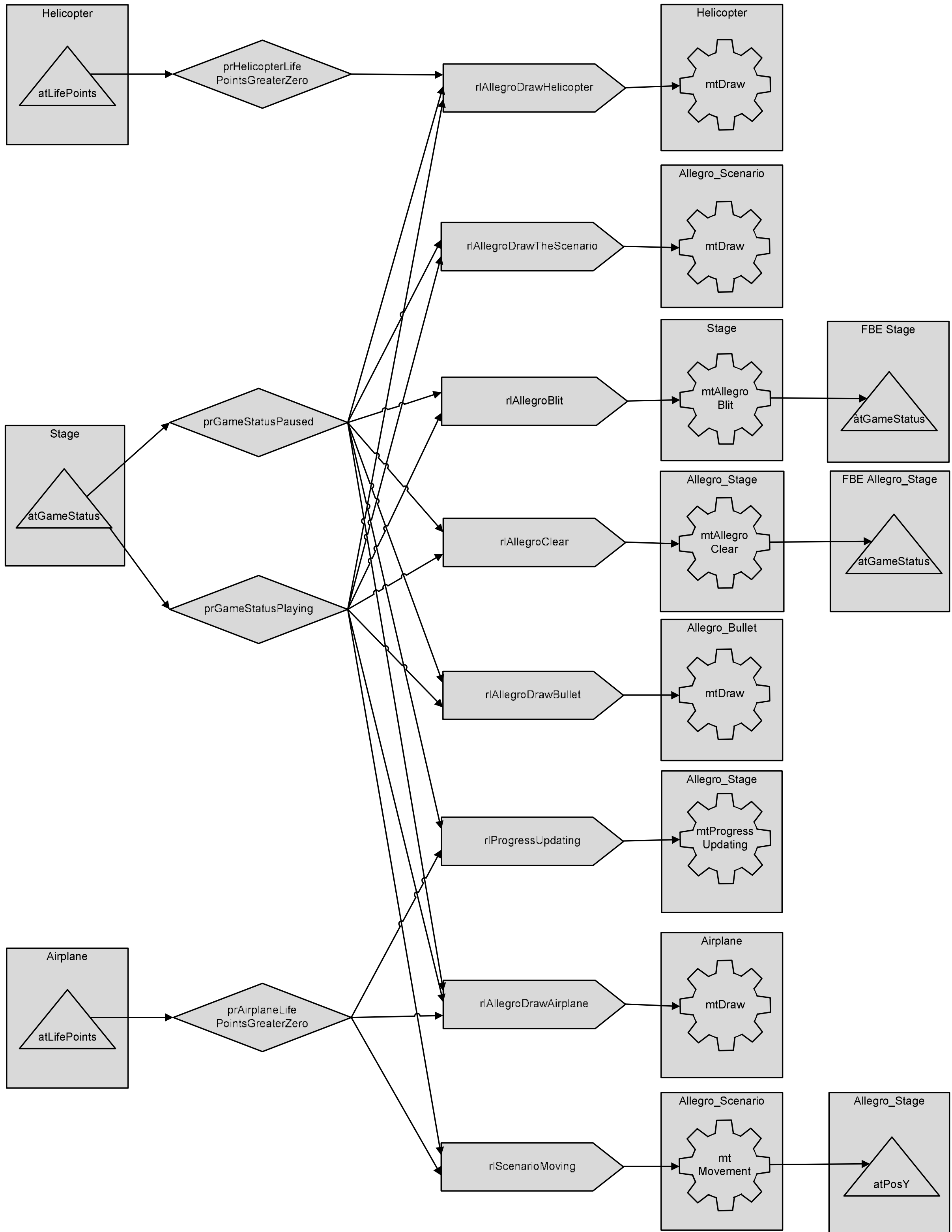


Figura 40 – Diagrama de objetos do caso de uso Controlar Apresentação

Tabela 92 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Controlar Apresentação

Variáveis operacionais	*Attributes*
Avião ter pontos de vida	atLifePoints (avião)
Helicóptero ter pontos de vida	atLifePoints (helicóptero)
Estado de jogo "jogando"	atGameStatus
Estado de jogo "pausado"	atGameStatus

Tabela 93 – Matriz de Casos de Teste para o Caso de Uso Controlar Apresentação

		Variáveis operacionais			Resultados esperados			
		Attributes			*Premisses*			*Rule*
ID caso de teste	**Cenário/ Condição**	**atLife Points (Avião)**	**atLifePoints (helicóptero)**	**atGameStatus**	**prHelicopterLifePointsGreaterZero**	**prAirplaneLifePointsGreaterZero**	**prGame Status Playing**	
CT01	Cenário 1 Desenhar avião	1	N/A	2	N/A	V	V	Aprova *Rule* rlAllegroDrawAirplane
CT02	Cenário 2 Desenhar helicóptero	N/A	1	2	V	N/A	V	Aprova *Rule* rlAllegroDrawHelicopter
CT03	Cenário 3 Redesenha a tela	N/A	N/A	N/A	N/A	N/A	N/A	Aprova *Rule* rlAllegoDrawScenario

A Tabela 94 apresenta a determinação do número de casos de teste mínimo que pode ser gerado para exercitar diretamente as entidades que compõem este caso de uso.

Tabela 94 – Determinação do número de casos de teste possíveis com análise de valores limite

atGameStatus	atLifePoints (avião)	atLifePoints (helicóptero)
4 estados possíveis (N/A, 1 (pausado), 2 (jogando) ou 3 (parado))	3 valores possíveis (N/A, 0 ou 1)	3 valores possíveis (N/A, 0 ou 1)
Total de casos de teste: 4*3*3 = 36 casos de teste diferentes (possíveis) apenas com valores limite		

5.1.2.5 Testes de integração no caso de uso Pausar e Continuar Jogo

A Tabela 95 apresenta a descrição do caso de uso Pausar e Continuar Jogo. A Tabela 96 apresenta a determinação de classes de equivalência e análise de valor limite para as variáveis operacionais identificadas para este caso de uso. A Tabela 97 apresenta a matriz de casos de teste com valores válidos, inválidos e não avaliados para cada variável operacional identificada. A Figura 41 apresenta o diagrama de objetos. A Tabela 98 apresenta uma relação entre as variáveis operacionais e os *Attributes* deste caso de uso. A Tabela 99 apresenta a matriz de

casos de teste com valores que exercitem diretamente as entidades deste caso de uso.

Tabela 95 – Descrição do caso de uso Pausar e Continuar Jogo

ID Caso de Uso	UC5		
Nome do Caso de Uso	Pausar e Continuar Jogo		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Jogador		
Descrição	Este caso de uso controla as funções pausar e continuar o jogo		
Disparador	O caso de uso inicia quando o jogador iniciar o jogo.		
Pré-condições	O jogo precisa estar em estado jogando ou pausado.		
Pós-condições	Pós-condição 1) Deverá ser pausado o jogo com estado “jogando”. Pós-condição 2) Deverá ser continuado um jogo com estado “pausado”		
Fluxo básico	1. O jogo fica com estado jogando ou pausado. 2. O sistema aguarda o jogador pressionar algum botão.		
[Fluxo Alternativo 1]	O jogador pressiona o botão de pausa. 1. No passo 2 do fluxo básico o jogador pressiona o botão de pausa. 2. O sistema verifica se o estado de jogo é “jogando”. 3. O sistema verifica se o jogador possui mais que zero de pontos de vida. 4. O jogo será pausado. 5. O sistema continua no passo 1 do fluxo básico.		
[Fluxo Alternativo 2]	O jogador pressiona o botão para continuar o jogo. 1. Tempo para disparar no passo 2 do fluxo básico o jogador pressiona o botão para continuar o jogo. 2. O sistema verifica se o estado de jogo é “pausado”. 3. O sistema verifica se o jogador possui mais que zero de pontos de vida. 6. O jogo será continuado. 4. O sistema continua no passo 1 do fluxo básico.		
Inclusões	O caso de uso Controlar Pausar e Continuar Jogo inclui o caso de uso Realizar Leitura de Teclado.		
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando ou pausado.		
Requisitos	Os requisitos funcionais implementados total ou parcialmente são: RF009.		
Considerações	Considera-se que: • O botão de pausar ou continuar jogo é a tecla Enter do teclado.		

Tabela 96 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Pausar e Continuar Jogo

	Valores limite		Classes de equivalência	
Condições ou Variáveis Operacionais	Válido	Inválido	Válidas	Inválidas
Pressionar botão de pausar	Verdadeiro	Falso	Verdadeiro	Falso
Pressionar botão para continuar	Verdadeiro	Falso	Verdadeiro	Falso

Tabela 97 – Matriz de Casos de Teste com valores (caso de uso Pausar e Continuar Jogo)

ID caso de teste	Cenário/ Condição	Botão pausar	Botão continuar	Resultado esperado
CT01	Cenário 1 – Pausar jogo	V	N/A	Pausa o jogo
CT02	Cenário 2 – Continuar o jogo	N/A	V	Continua o jogo

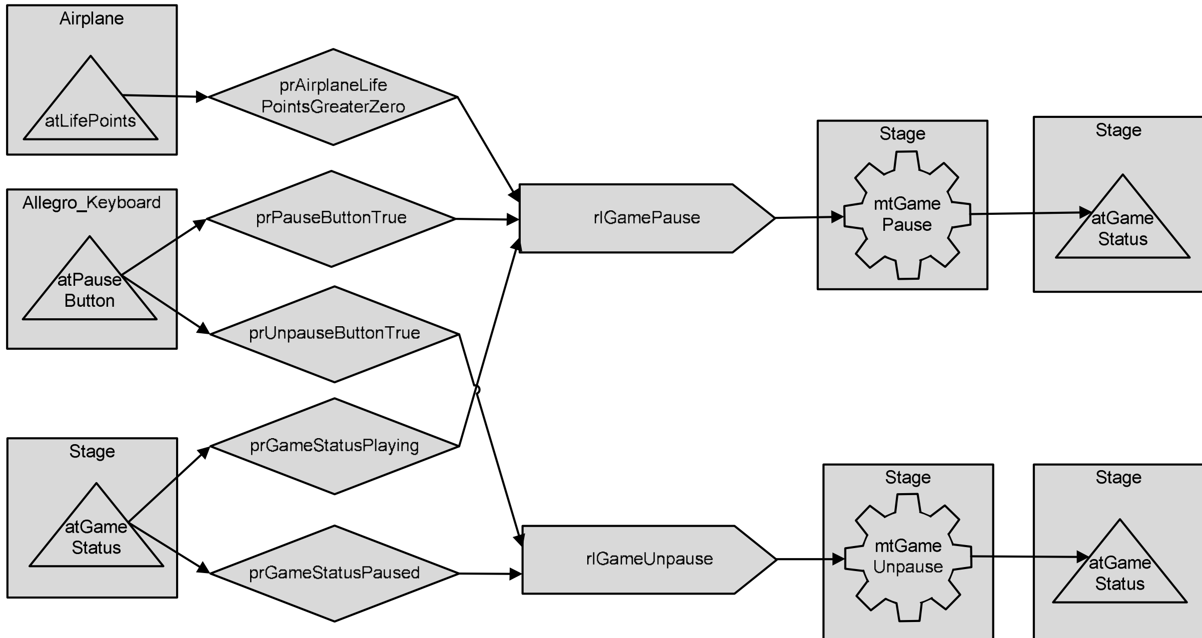


Figura 41 – Diagrama de objetos do caso de uso Pausar e Continuar Jogo

Tabela 98 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Pausar e Continuar Jogo

Variáveis operacionais	*Attributes*
Botão pausar	atPauseButton
Botão continuar	atUnpauseButton
Pontos vida avião	atLifePoints
Jogo pausado	atGameStatus
Jogo estado jogando	atGameStatus

Tabela 99 – Matriz de Casos de Teste para o Caso de Uso Pausar e Continuar Jogo

		Variáveis operacionais				Resultados esperados					
		Attributes				*Premisses*					*Rule*
ID caso de teste	**Cenário/ Condição**	**atPauseButton**	**atUnpauseButton**	**atLifePoints**	**atGameStatus**	**prAirplaneLifePointsGreaterZero**	**prPauseButtonTrue**	**prUnpauseButtonTrue**	**prGameStatusPlaying**	**prGameStatusPaused**	
CT01	Cenário 1 – Pausar jogo	V	N/A	1	2	V	V	N/A	V	N/A	Aprova *Rule* rlGame-Pause
CT02	Cenário 2 – Conti-nuar o jogo	N/A	V	1	1	V	N/A	V	N/A	V	Aprova *Rule* rlGame-Unpause

A Tabela 100 apresenta a determinação do número de casos de teste mínimo que pode ser gerado para exercitar diretamente as entidades que compõem este caso de uso.

Tabela 100 – Determinação do número de casos de teste possíveis com análise de valores limite

atGameStatus	atLifePoints	atPauseButton	atUnpauseButton
4 estados possíveis (N/A, 1 (pausado), 2 (jogando) ou 3 (parado))	3 valores possíveis (N/A, 0 ou 1)	3 valores possíveis (N/A, verdadeiro ou falso)	3 valores possíveis (N/A, verdadeiro ou falso)
Total de casos de teste: 4*3*3*3 = 108 casos de teste diferentes (possíveis) apenas com valores limite			

A execução do caso de teste CT01 revelou um erro na definição da *Premise* prGameStatusPlaying. Ao invés de avaliar o *Attribute* atGameStatus, estava avaliando o *Attribute* atPauseButton. Esse erro foi corrigido.

5.1.2.6 Testes de integração no caso de uso Parar Jogo

A Tabela 101 apresenta a descrição do caso de uso Parar Jogo. A Tabela 102 apresenta a determinação de classes de equivalência e análise de valor limite para as variáveis operacionais identificados para este caso de uso. A Tabela 103 apresenta a matriz de casos de teste com valores válidos, inválidos e não avaliados para cada variável operacional identificada. A Figura 41 apresenta o diagrama de objetos do caso de uso Parar Jogo. A Tabela 104 apresenta uma relação entre as

variáveis operacionais e os *Attributes* deste caso de uso. A Tabela 105 apresenta a matriz de casos de teste com valores que exercitem diretamente as entidades deste caso de uso.

Tabela 101 – Descrição do caso de uso Parar Jogo

ID Caso de Uso	UC6		
Nome do Caso de Uso	Parar Jogo		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Jogador		
Descrição	Este caso de uso controla as funções parar o jogo e finalizar o software.		
Disparador	O caso de uso inicia quando o jogador pressionar o botão para parar o jogo.		
Pré-condições	O jogo precisa estar em estado jogando ou pausado.		
Pós-condições	Pós-condição 1) O jogo será fechado.		
Fluxo básico	1. O jogo fica com estado jogando ou pausado. 2. O sistema aguarda o jogador pressionar algum botão.		
[Fluxo Alternativo 1]	O jogador pressiona o botão para parar o jogo. 1. No passo 2 do fluxo básico o jogador pressiona o botão para parar o jogo. 2. O sistema verifica se o estado de jogo é “jogando” ou “pausado”. 3. O jogo será parado. 4. O sistema será fechado.		
Inclusões	O caso de uso Parar Jogo inclui o caso de uso Realizar Leitura de Teclado.		
Frequência de Uso	Este caso de uso é usado quando o jogador pressionar o botão para parar.		
Requisitos	O requisito funcional implementado totalmente é: RF014.		
Considerações	Considera-se que: • O botão de parar o jogo é a tecla de Esc do teclado padrão.		

Tabela 102 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Parar Jogo

	Valores limite		Classes de equivalência	
Condições ou Variáveis Operacionais	Válido	Inválido	Válidas	Inválidas
Pressionar o botão para parar o jogo	Verdadeiro	Falso	Verdadeiro	Falso

Tabela 103 – Matriz de Casos de Teste com valores (caso de uso Parar Jogo)

ID caso de teste	Cenário/ Condição	Botão parar	Resultado esperado
CT01	Cenário 1 Avião não realiza ação	V	O jogo será finalizado

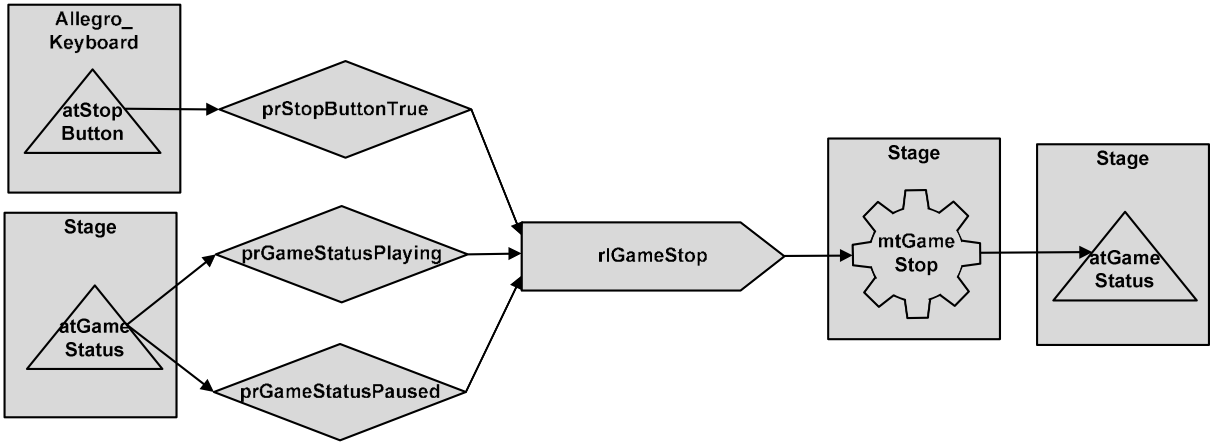


Figura 42 – Diagrama de objetos do caso de uso Parar Jogo

Tabela 104 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Parar Jogo

Variáveis operacionais	*Attributes*
Botão parar	atStopButton
Estado de jogo	atGameStatus

Tabela 105 – Matriz de Casos de Teste para o Caso de Uso Parar Jogo

		Variáveis operacionais		Resultados esperados			
		Attributes		*Premisses*			*Rule*
ID caso de teste	Cenário/ Condição	atGame-Status	atStop-Button	prStop Button True	prGame Status Playing	prGame Status Paused	rlGameStop
CT01	Cenário 1 Avião não realiza ação	2	V	V	V	N/A	Aprova a *Rule* rlGameStop

A Tabela 106 apresenta a determinação do número de casos de teste mínimo que pode ser gerado para exercitar diretamente as entidades que compõem este caso de uso.

Tabela 106 – Determinação do número de casos de teste possíveis com análise de valores limite

atGameStatus	atStopButton
4 estados possíveis (N/A, 1 (pausado), 2 (jogando) ou 3 (parado))	4 valores possíveis (N/A, 2, 3, 4)
Total de casos de teste: 4*4 = 16 casos de teste diferentes (possíveis) apenas com valores limite	

5.1.2.7 Testes de integração no caso de uso Detectar Colisão de Projétil

A Tabela 107 apresenta a descrição do caso de uso Detectar Colisão de Projétil. A Tabela 108 apresenta a determinação de classes de equivalência e análise de valor limite para as variáveis operacionais identificados para este caso de uso. A Tabela 109 apresenta as variáveis operacionais e *Attributes* correspondentes.

A Figura 43 apresenta o diagrama de objetos do caso de uso Detectar Colisão de Projétil. A Tabela 110 apresenta a matriz de casos de teste com valores válidos, inválidos e não avaliados. A Tabela 111 apresenta a matriz de casos de teste com valores que exercitem diretamente as entidades deste caso de uso.

Tabela 107 – Descrição do caso de uso Detectar Colisão de Projétil

ID Caso de Uso	UC7		
Nome do Caso de Uso	Detectar Colisão de Projétil		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Não tem.		
Descrição	Este caso de uso é responsável por detectar colisão entre projétil e avião ou projétil e helicóptero.		
Disparador	O caso de uso inicia quando um projétil colide com o avião ou helicóptero.		
Pré-condições	O jogo precisa estar em estado jogando.		
Pós-condições	Pós-condição 1) Será decrementado o valor de pontos de vida do personagem que colidiu com o projétil.		
Fluxo básico	1. O jogo fica com estado jogando. 2. O jogo aguarda o disparo de um projétil		
[Fluxo Alternativo 1]	O projétil colide com um personagem 1. No passo 2 do fluxo básico, o avião ou helicóptero disparam um projétil. 2. O projétil segue sua trajetória. 3. O projétil colide com o avião ou helicóptero. 4. O sistema decrementa o valor dos pontos de vida do personagem que colidiu com o projétil. 5. O caso de uso continua no passo 2 do fluxo básico.		
Inclusões	O caso de uso Detectar Colisão de Projétil não inclui outros Casos de Uso.		
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando e há colisão.		
Requisitos	O requisito funcional implementado total ou parcialmente são: RF006.		
Considerações	Considera-se que: • Uma colisão ocorre quando um projétil disparado encosta na área reservada de algum personagem na tela.		

Tabela 108 – Classes de equivalência e análise de valores limite para as variáveis operacionais para o caso de uso Detectar Colisão de Projétil

<table>
<tr><th></th><th colspan="2">Valores limite</th><th colspan="2">Classes de equivalência</th></tr>
<tr><th>Condições ou Variáveis Operacionais</th><th>Válido</th><th>Inválido</th><th>Válidas</th><th>Inválidas</th></tr>
<tr><td>Pontos de vida (avião)</td><td>1</td><td>0</td><td>Pontos de vida >0</td><td>Pontos de vida <=0</td></tr>
<tr><td>Pontos de vida (helicóptero)</td><td>1</td><td>0</td><td>Pontos de vida >0</td><td>Pontos de vida <=0</td></tr>
<tr><td>Colisão</td><td>Verdadeiro</td><td>Falso</td><td>Verdadeiro</td><td>Falso</td></tr>
<tr><td>Posição em X (avião)</td><td rowspan="6">O projétil precisa estar contido dentro da delimitação do personagem</td><td rowspan="6">O projétil não pode estar contido dentro da delimitação do personagem</td><td></td><td></td></tr>
<tr><td>Posição em Y (avião)</td><td></td><td></td></tr>
<tr><td>Posição em Y (helicóptero)</td><td></td><td></td></tr>
<tr><td>Posição em Y (helicóptero)</td><td></td><td></td></tr>
<tr><td>Posição em X (projétil)</td><td></td><td></td></tr>
<tr><td>Posição em Y (projétil)</td><td></td><td></td></tr>
<tr><td>Estado de jogo</td><td>2</td><td>1; 3</td><td>Estado de jogo=2</td><td>Estado de jogo<>2</td></tr>
</table>

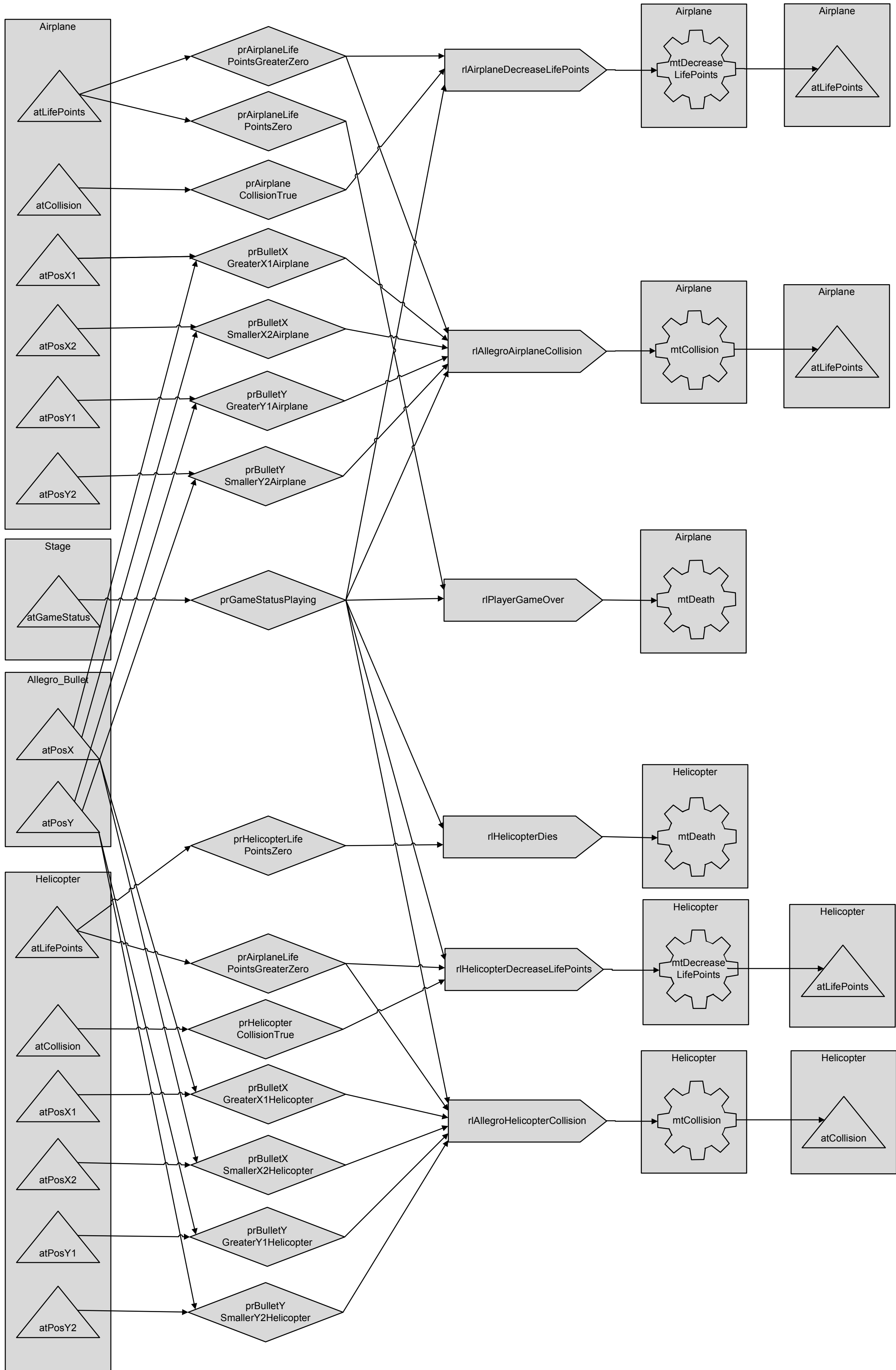

Airplane
atLifePoints
atCollision
atPosX1
atPosX2
atPosY1
atPosY2
Stage
atGameStatus
Allegro_Bullet
atPosX
atPosY
Helicopter
atLifePoints
atCollision
atPosX1
atPosX2
atPosY1
atPosY2
prAirplaneLife PointsGreaterZero
prAirplaneLife PointsZero
prAirplane CollisionTrue
prBulletX GreaterX1Airplane
prBulletX SmallerX2Airplane
prBulletY GreaterY1Airplane
prBulletY SmallerY2Airplane
prGameStatusPlaying
prHelicopterLife PointsZero
prAirplaneLife PointsGreaterZero
prHelicopter CollisionTrue
prBulletX GreaterX1Helicopter
prBulletX SmallerX2Helicopter
prBulletY GreaterY1Helicopter
prBulletY SmallerY2Helicopter
rlAirplaneDecreaseLifePoints
rlAllegroAirplaneCollision
rlPlayerGameOver
rlHelicopterDies
rlHelicopterDecreaseLifePoints
rlAllegroHelicopterCollision
Airplane
mtDecrease LifePoints
Airplane
atLifePoints
Airplane
mtCollision
Airplane
atLifePoints
Airplane
mtDeath
Helicopter
mtDeath
Helicopter
mtDecrease LifePoints
Helicopter
atLifePoints
Helicopter
mtCollision
Helicopter
atCollision

Figura 43 – Diagrama de objetos do caso de uso Detectar Colisão de Projétil

Para mostrar, optou-se por apresentar apenas as variáveis operacionais relacionadas ao avião e ao projétil, conforme apresentado na Tabela 109.

Tabela 109 – Variáveis operacionais e *Attributes* correspondentes para o caso de uso Detectar Colisão de Projétil

Variáveis operacionais	*Attributes*	Número
Pontos de vida	atLifePoints	1
Colisão	atCollision	2
Posição em X (avião)	atPosX1	3
Posição em X (avião)	atPosX2	4
Posição em Y (avião)	atPosY1	5
Posição em Y (avião)	atPosY2	6
Estado de jogo	atGameStatus	7
Posição em X (projétil)	atPosX	8
Posição em Y (projétil)	atPosY	9

Tabela 110 – Matriz de Casos de Teste com valores (caso de uso Detectar Colisão de Projétil)

		Variáveis operacionais							Resultados esperados			
		Attributes							*Premisses*			*Rule*
ID caso de teste	**Cenário/ Condição**	**Pontos de vida**	**Estado de jogo**	**Colisão**	**Posição X (avião)**	**Posição Y (avião)**	**Posição X (projétil)**	**Posição Y (projétil)**	**prAirplaneLifePointsGreaterZero**	**prGame Status Playing**	**prAirplaneCollisionTrue**	
CT01	Cenário 1 – Colisão com Avião	V	V	V	V	V	V	V	V	V	V	Aprova *Rule* rlAirplaneDecrease LifePoints
CT02	Cenário 2 não ocorre colisão com avião	V	**I**	V	V	V	V	V	V	V	V	Não aprova *Rule*

Tabela 111 – Matriz de Casos de Teste para o Caso de Uso Detectar Colisão de Projétil

		Variáveis operacionais									Resultados esperados			
		Attributes									*Premisses*			*Rule*
ID caso de teste	**Cenário/ Condição**	**atLife Points**	**atGame Status**	**atCollision**	**atPosX1**	**atPosX2**	**atPosY1**	**atPosX2**	**atPosX**	**atPosY**	**prAirplane LifePoints GreaterZero**	**prGame Status Playing**	**prAirplaneCollisionTrue**	
CT01	Cenário 1 – Colisão com Avião	1	2	V	10	70	10	70	11	11	V	V	V	Aprova *Rule* rlAirplaneDecreaseLifePoints
CT02	Cenário 2 não ocorre colisão com avião	1	**1**	V	10	70	10	70	11	11	V	V	V	Não aprova *Rule*

Considerando que na Tabela 110 cada variável operacional pode assumir, no mínimo, um dentre três valores e que há 7 variáveis operacionais, então podem ser gerados 2187 casos de teste diferentes.

A execução desses casos de teste resultou na descoberta de alguns erros. Quatro *Premisses* não estavam avaliando corretamente os *Attributes* de posicionamento do avião em relação a um projétil. O operador lógico era de “Greater Than”, sendo que deveria ser “Greater Or Equal”. Com esses erros, havia uma tolerância de um pixel em relação à colisão com projétil.

5.1.2.8 Testes de integração no caso de uso Iniciar Jogo

A Tabela 112 apresenta a descrição do caso de uso Iniciar Jogo.

Tabela 112 – Descrição do caso de uso Iniciar Jogo

ID Caso de Uso	UC8		
Nome do Caso de Uso	Iniciar Jogo		
Criado por	Clayton	Última vez atualizado por	Clayton
Data de Criação	01/05/15	Data da última revisão	01/05/15
Atores	Jogador.		
Descrição	Este caso de uso apenas inicia um novo jogo		
Disparador	O caso de uso inicia quando o jogador clicar na opção iniciar jogo no menu principal.		
Pré-condições	Não tem.		
Pós-condições	Pós-condição 1) O sistema deverá iniciar o caso de uso Realizar Leitura de Teclado.		
Fluxo básico	3. O jogo fica com estado jogando. 4. O sistema aguarda o jogador pressionar algum botão.		
Inclusões	O caso de uso Realizar Leitura de Teclado não inclui outros Casos de Uso.		
Frequência de Uso	Este caso de uso é usado sempre que o jogo está com estado jogando ou pausado.		
Requisitos	O requisito funcional implementado totalmente: RF017		

Observação: O único teste a ser realizado é conferir se o menu funciona de acordo com o esperado, i.e. se o clique na opção “Iniciar Jogo” inicia o jogo em estado jogando.

5.2 CONSIDERAÇÕES SOBRE O CASO DE ESTUDO

Nesse capítulo, foi apresentado um caso de estudo em que foi aplicado o método de teste de software proposto no Capítulo 4 no software apresentado no Capítulo 3.

Esee método permite a realização da atividade de teste em PON de maneira sistemática, metodológica e fundamentada em conceitos teóricos e práticos do teste de software. A utilização desse método requer a especificação adequada dos requisitos e dos casos de uso, para que a atividade de teste possa ser realizada satisfatoriamente. O caso de estudo mostrou que o método de teste possibilita a identificação de erros que possam ocorrer durante o desenvolvimento de uma aplicação PON.

Embora o método tenha se mostrado útil para o teste de software em aplicação PON, se for realizado manualmente, pode ser uma atividade demorada e propensa a erros. Por isto, faz-se necessário o desenvolvimento de uma ferramenta para realização de testes em PON. Além disto, adaptações e melhorias podem ser desenvolvidas para aperfeiçoar o método e complementar esses critérios propostos. Ademais, é necessário validar o critério de teste que cobre “caminhos” do software (caixa branca) PON para que possa ser adequadamente aplicada no método de teste.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta a conclusão final dessa dissertação e aponta perspectivas para trabalhos futuros que podem ser desenvolvidos a partir dela. A Seção 6.1 apresenta conclusão desta dissertação de mestrado relacionado às contribuições deste trabalho. A Seção 6.2, por sua vez, apresenta proposta para trabalhos futuros que contribuirão ainda mais para o amadurecimento do método de teste de software para o PON.

6.1 CONCLUSÃO

O teste de software é uma etapa integrante das principais metodologias de desenvolvimento de software existentes. Entretanto, ainda não existia um método para teste de software em PON. O estudo e proposição de um método para teste de software em PON resultou em várias atividades que culminaram no desenvolvimento dessa dissertação.

Inicialmente, foram realizadas pesquisas em trabalhos do grupo do PON para conhecer o atual estado da arte e da técnica desse paradigma. A partir disso, para aplicar o conhecimento adquirido sobre o estado da técnica do PON, foi modelado e desenvolvido um software de combate aéreo seguindo as etapas do DON. É importante ressaltar que até então poucas aplicações haviam sido concebidas sob os princípios desse paradigma, com escopo relativamente modesto, que, de certa forma, não contribuía para um aprendizado orientado a exemplos, oferecendo poucas possibilidades de desenvolvimento aos desenvolvedores e estudantes do PON. Então, esse software auxiliou também na visão de análise e projeto de uma aplicação em PON.

Constatou-se também, que o PON tem sido materializado em termos de programação e modelagem, porém não possuía um método formalizado para orientar a atividade de teste. O PON é fundamentado nos melhores conceitos do PI e PD, no entanto, ele difere substancialmente em relação à definição de instruções, sintaxe e funcionamento. Por isso, técnicas e critérios de teste comuns não podem

ser simplesmente aplicadas, sem levar em consideração as particularidades desse paradigma.

Com isso, foi realizada uma ampla pesquisa para conhecer as técnicas de teste de software mais comuns e representativas que possibilitassem a compreensão mais aprofundada deste tema. Constatou-se que, em geral, o PI é que mais possui trabalhos relacionados a teste de software por ter a maioria das linguagens de programação mais utilizadas. Por sua vez, o teste de software no PD (principalmente SBRs) envolve, principalmente, a representação do sistema por meio de uma rede de Petri. Porém, a representação do PON por meio de redes de Petri ainda não está plenamente definida e formalizada o que inviabiliza o estudo e proposição de testes com esta técnica.

Então, buscou-se compreender como são realizadas as principais atividades de teste de software. Constatou-se com base na literatura que o teste de software ocorre, geralmente, em quatro fases:

- Teste unitário: testa a menor unidade ou bloco de instrução de um programa;
- Teste de integração: testa a integração das unidades que foram testadas individualmente;
- Teste de validação: testa o software de acordo com o documento de requisitos;
- Teste de sistema: testa o sistema por completo incluindo a interação com outros softwares, banco de dados, conexões de rede, etc.

Notou-se que apenas as fases de teste unitário e teste de integração buscam exercitar o código e a lógica interna do software. As fases de teste de validação e teste de sistema não consideram o código fonte ou estruturas internas do software. Assim, para o contexto do PON, optou-se por estudar as fases de teste unitário e teste de integração, sendo que as fases de teste de validação e teste de sistema, a priori, podem ser realizadas no PON da mesma maneira que se faz em outros paradigmas.

A proposta de teste de unidade em PON considera as menores unidades testáveis do paradigma que são *Premise*, *Condition e SubCondition*, *Rule* e *FBE* (que envolve *Method*). Cada uma destas unidades possui particularidades de implementação e critérios de teste. *Premise* utiliza a determinação de classes de equivalência e análise de valor limite para determinar casos de teste. *Condition*

avalia o estado de *SubConditions* ou *Premisses* para que ela possa ser aprovada. *Rule* avalia o estado de sua *Condition*. *Methods* são testados como métodos OO e sugere-se o uso do critério de teste funcional.

A proposta de teste de integração em PON considera o funcionamento e o comportamento de todas as entidades em conjunto na realização de casos de uso. Testes de integração podem ser realizados por meio de duas abordagens que se complementam: (1) Teste sobre a descrição e funcionalidades do caso de uso e (2) Teste diretamente sobre as notificações entre as entidades PON que implementam o caso de uso.

Com isso, essa dissertação preencheu uma lacuna que havia neste paradigma: o desenvolvimento de um método de teste para processos de desenvolvimento de software usando o PON. As contribuições desta dissertação estão voltadas para a determinação de teste de software nas fases de teste unitário e teste de integração e são utilizados critérios de caixa-branca e caixa-preta para a geração de casos de teste.

O caso de estudo permitiu a aplicação da proposta do método de teste de software em toda a aplicação apresentada no Capítulo 3, aprofundando mais o conteúdo e aprendizado sobre o método. Também, foram apresentadas demonstrações, comentários e recomendações sobre o teste de software em PON. O método de teste de software mostrou-se útil para geração de casos de teste para as fases de teste unitário e teste de integração.

O PON permite a alteração do estado de *Attributes*, *Premisses*, *Rules* e *Methods* em qualquer parte do programa. Cada um desses elementos pode influenciar no comportamento de outras entidades já existentes. Certamente, esta característica exige maior atenção do programador, tendo em vista o potencial impacto que decorre da execução de cada entidade, tornando obrigatório levar isto em consideração.

Assim, a união das contribuições desta dissertação e outros trabalhos aqui apresentados, bem como as contribuições apresentadas nos demais trabalhos relacionados ao PON, certamente abrirão margem para outros trabalhos que contribuirão ainda mais com o amadurecimento do paradigma em questão, o qual atualmente é considerado um paradigma emergente. Certamente, com novos avanços nessas áreas em estudo será possível vislumbrar novos métodos de teste de software em PON.

6.2 TRABALHOS FUTUROS

Este trabalho é pioneiro na apresentação de uma estratégia de teste de software em aplicações PON, e abre perspectivas de pesquisa para amadurecer ainda mais esta área. A seguir são apresentados trabalhos futuros que podem ser desenvolvidos a partir dessa dissertação.

Desenvolvimento de Ferramenta para Realização de Testes de Software em PON

A atividade de testes em qualquer paradigma está propensa a erros humanos na sua elaboração. Além disso, é causadora de fadiga e custosa com relação a tempo para sua execução (DELAMARO; MALDONADO; JINO, 2007). Por isto, faz-se necessário desenvolver uma ferramenta que possa automatizar esta tarefa, a exemplo de outras existentes para como o JUnit (Java), Mothra (C++), LDRA, entre outras. Assim, o testador concentraria esforços em escolher casos de teste que melhor se adéquam a necessidade e observaria os resultados produzidos pela ferramenta de testes. Além disto, essa ferramenta poderia disponibilizar recurso de gerar casos de teste para entidades que são alteradas em tempo de execução.

Aprimoramento dos critérios de teste apresentados e proposição de novos critérios

Nesse trabalho, foi apresentado um método de teste para as fases de teste unitário e teste de integração. O teste nestas duas fases foi realizado por meio de alguns critérios de caixa-preta e caixa-branca adaptados para o PON. Possivelmente, cabe mais investigação para melhorar este método e propor novos critérios específicos para o PON ou utilizar outros critérios que também possam ser adequadas a esse paradigma e que não foram apresentados nessa dissertação.

Teste em PON por meio de técnicas de teste de SBRs com redes de Petri

É preciso formalizar e validar a representação do PON por meio de redes de Petri para que seja possível desenvolver uma técnica de teste de software utilizando esta ferramenta. Alguns trabalhos que investigaram SBRs e apresentaram técnicas

de teste com redes de Petri podem ser úteis para o desenvolvimento de uma técnica de teste de software com o uso de redes de Petri:

- Nazareth e Kennedy (1991) apresentam uma abordagem alternativa para verificação em SBR em que o sistema é modelado como um grafo orientado. Muitas proposições para detecção de erros foram formuladas e provadas. Um algoritmo foi desenvolvido para detectar erros.
- Nazareth (1993) apresentou outra abordagem alternativa para verificação de SBR, onde o sistema é modelado como rede de Petri, expondo as possibilidades de se fazer verificação de erros. O conjunto de preposições foi formulado para expor erros como circularidade, redundância, conflito e caminhos vazios na base de conhecimentos. Também foram discutidas dificuldades para implementação de redes de Petri.
- Weyuker (1996) propôs um algoritmo que seleciona um caso de teste de cobertura para um grande número de estados relevantes. A abordagem identifica os estados que mais podem ser executados usando informação de distribuição operacional. Depois de aplicar seu algoritmo, é verificada a corretude de cada estado testado por meio resultados probabilísticos da suíte de testes utilizada. Resultados matemáticos empíricos foram apresentados.
- Cardoso e Valette (1997) destacam que a vantagem da utilização da rede de Petri para representar um sistema de regras é devido ao seu formalismo que permite análise e verificação.
- Andrews, O’fallon e Chen (2000) propuseram um método de teste de SBR para modelos VHDL com objetivo de verificar o comportamento dos modelos antes de transferir o design para um hardware. É apresentada uma maneira mais sistemática para endereçar a geração de testes padrões. O método discutido neste artigo é baseado em teste caixa-branca, onde que as estruturas internas, comportamento e fluxo de dados do modelo são analisadas. Novos casos de testes são gerados baseados em regras heurísticas.
- Döll (2002) ressalta que algumas das características de redes de Petri as tornam propícias para capturar especificações comportamentais, orientadas a objetos e concorrentes. Redes de Petri permitem a modelagem de

concorrência, sincronização e compartilhamento de recursos em um sistema; além de que existem muitos resultados teóricos associados a redes de Petri para a análise comportamental, tais como detecção de bloqueio e análise de desempenho.

Além dos trabalhos futuros supramencionados, também se apontam novas linhas de pesquisa para futuros estudos:

- Melhoria nos critérios de teste apresentados nesta dissertação;
- Teste em hardware próprio para o PON;
- Teste na linguagem PON (LingPON) que ainda está em desenvolvimento e possivelmente substituirá a linguagem do *framework* no desenvolvimento em PON;
- Estudo e proposição do teste de mutação para o PON e apresentação de operadores de mutação para a linguagem do Framework PON e, futuramente, para a linguagem de programação PON;
- Melhoria na representação por meio do diagrama de objetos PON, permitindo maior correspondência entre diagrama e código fonte.

REFERÊNCIAS

AGRAWAL, H. et al. **Design of Mutant Operators for the C Programming Language**. Software Engineering Research Center, Purdue University. West Lafayette, IN. 1989.

ANDREWS, A.; O'FALLON, A.; CHEN, T. A rule-based software testing method for VHDL models. **The School of Electrical Engineering and Computer Science**, Washington, 2000.

ATARI2600. River Raid. **Atari 2600**, 2015. Disponivel em: <http://www.atari2600.com.br/Atari/Roms/01nD/River_Raid>. Acesso em: 22 abr. 2015.

AVRITZER, A.; ROS, J. P.; WEYUKER, E. J. Reliability testing of rule-based systems. **International Symposium on Software Reliability Engineering**, White Plains, New York, Outubro 1996.

BANASZEWSKI, R. F. **Paradigma Orientado a Notificações - Avanços e Comparações**. Dissertação (Mestrado) - Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2009.

BANASZEWSKI, R. F. et al. Notification Oriented Paradigm (NOP) – A Software Development Approach based on Artificial Intelligence Concepts. **VI Congress of Logic Applied to Technology - LAPTEC**, 2007.

BATISTA, M. V. et al. Uma comparação entre o Paradigma Orientado a Notificações (PON) e o Paradigma Orientado a Objetos (POO) realizado por meio da implementação de um Sistema de Vendas. **III Congreso Internacional de Computación y Telecom - COMTEL**, 2011.

BEIZER, B. **Black-box testing:** techniques for functional testing of software and systems. [S.l.]: John Wiley & Sons, Inc., 1995.

BELMONTE, D.; SIMAO, J. M.; STADZISZ, P. C. Proposta de um método para distribuição de carga de trababalho usando o Paradigma Orientado a Notificações (PON). **Revista SODEBRAS**, v. 8, n. 84, dez 2012.

BINDER, R. V. **Testing Object-oriented Systems:** Models, Patterns, and Tools. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-80938-9.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)**. [S.l.]: Addison-Wesley Professional, 2005. ISBN 0321267974.

BOOKSHEAR, J. G. **Computer Science:** An Overview. [S.l.]: Addison Wesley, 2006.

BRIAND, L.; LABICHE, Y. A UML-based approach to system testing. In: HARTMANN, J. et al. **A UML-based approach to system testing**. [S.l.]: Springer, 2001. p. 194-208.

BROOKSHEAR, J. G. **Computer Science:** An Overview. 11. ed. [S.l.]: Prentice Hall, 2011.

BUDD, T. A. **Mutation Analysis of Program Test Data**. New Haven, CT, USA. 1980. AAI8025191.

CAPUANO, F. G.; IDOETA, I. V. **Elementos de eletrônica digital**. 40. ed. [S.l.]: [s.n.], 2000.

CARDOSO, J.; VALETTE, R. **Redes de Petri.** [S.l.]: Editora UFSC, 1997.

CHAIM, M. L. **Poke-tool: uma ferramenta para suporte ao teste estrutural de programas baseado em analise de fluxo de dados**. Faculdade de Engenharia Elétrica e de Computação FEEC, UNICAMP. [S.l.]. 1991.

CHENG, A. M. K.; CHEN, J. R. Response Time Analysis of OPS5 Production Systems. **IEEE Transaction on Software Engineering**, v. 12, p. 391-409, 2000.

COPELAND, L. **A Practitioner's Guide to Software Test Design**. Norwood, MA, USA: Artech House, Inc., 2004.

DEEN, S. M. **Agent-Based Manufacturing:** Advances in the Holonic Approach. [S.l.]: Springer, 2003. ISBN 9783540440697 LCCN: 2003042700.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. 4. ed. Rio de Janeiro: Elsevier, 2007.

DEMILLO, R. A. et al. **Software Testing and Evaluation**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1987. ISBN 0-8053-2535-2.

DEMILLO, R. A. et al. **An extended overview of the Mothra software testing environment**. Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on. [S.l.]: [s.n.]. 1988. p. 142-151.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. **IEEE Computer**, v. 11, n. 4, p. 34-41, April 1978. ISSN 0018-9162 DOI: 10.1109/C-M.1978.218136.

DEREZINSKA, A. **Object-oriented mutation to assess the quality of tests**. Euromicro Conference, 2003. Proceedings. 29th. [S.l.]: [s.n.]. Sept 2003. p. 417-420.

DÍAZ, M. et al. A component-based nuclear power plant simulator kernel. **Concurrency and Computation: Practice and Experience**, v. 19, n. 5, p. 593-607, 2007.

DÖLL, L. M. **Proposta de uma Metodologia para a Modelagem da Dinâmica de Sistemas Orientados a Objetos usando Redes de Petri Predicado/Transição**. Dissertação de Mestrado. CEFET-PR. [S.l.]. 2002.

FAISON, T. **Event-Based Programming Taking Events to the Limit**. [S.l.]: Springer, 2006.

FERREIRA, C. A. **Linguagem e compilador para o paradigma orientado a notificações (pon): avanços e comparações**. Seminário III, Programa de Pós-Graduação em Computação Aplicada (PPGCA), Universidade Tecnológica Federal do Paraná (UTFPR). [S.l.]. 2014.

FORGY, C. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. **Artificial Intelligences**, v. 19, n. 1, p. 17-37, 1982. ISSN 0004-3702. Disponivel em: <http://dx.doi.org/10.1016/0004-3702(82)90020-0>.

FRANKL, P. G.; WEYUKER, E. J. An Applicable Family of Data Flow Testing Criteria. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v. 14, n. 10, p. 1483-1498, out. 1988. ISSN 0098-5589 DOI: 10.1109/32.6194. Disponivel em: <http://dx.doi.org/10.1109/32.6194>.

GABBRIELLI, M.; MARTINI, S. **Programming Languages:** Principles and Paradigms. 1. ed. [S.l.]: Springer, 2010.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GAUDIOT, J.-L.; SOHN, A. Data-driven parallel production systems. **Software Engineering, IEEE Transactions on**, v. 16, n. 3, p. 281-293, 1990.

GELPERIN, D.; HETZEL, B. The Growth of Software Testing. **Commun. ACM**, New York, NY, USA, v. 31, n. 6, p. 687-695, jun. 1988. ISSN 0001-0782 DOI: 10.1145/62959.62965. Disponivel em: <http://doi.acm.org/10.1145/62959.62965>.

HARBOUR, J. S. **Game programming all in one third edition**. 3. ed. [S.l.]: Cengage Learning PTR, 2006.

HEUMANN, J. Generating test cases from use cases. **The Rational Edge**, v. 6, n. 01, 2001.

HOWDEN, W. E. Functional Program Testing. **Software Engineering, IEEE Transactions on**, v. SE-6, n. 2, p. 162-169, March 1980. ISSN 0098-5589 DOI: 10.1109/TSE.1980.230467.

IBM. **IBM Rational Software**, 2015. Disponivel em: <http://www-01.ibm.com/software/br/rational/>. Acesso em: 7 fev. 2015.

IEEE Standard for Software Test Documentation. **IEEE Std 829-1998**, p. i-, 1998. ISSN DOI: 10.1109/IEEESTD.1998.88820.

JACOBSON, I. The use-case construct in object-oriented software engineering. In: DUFFY, T. M. et al. **Scenario-based Design:** Envisioning Work and Technology in System Development. [S.l.]: John Wiley & Sons, 1995.

JACOBSON, I. et al. **Object-oriented software engineering:** a use case driven approach. [S.l.]: Addison Wesley, 1992.

JOHNSTON, W. M.; HANNA, J. R. P.; MILLAR, R. J. Advances in Dataflow Programming Languages. **ACM Computing Surveys**, New York, NY, USA, v. 36, n. 1, p. 1-34, mar. 2004. ISSN 0360-0300 DOI: 10.1145/1013208.1013209. Disponivel em: <http://doi.acm.org/10.1145/1013208.1013209>.

KAISLER, S. **Software Paradigm**. 1. ed. [S.l.]: Wiley-Interscience, 2005.

KANG, J. A.; CHENG, A. M. K. Shortening Matching Time in OPS5 Production Systems. **IEEE Transaction on Software Engineering**, v. 30, 2004.

KOSCIANSKI, A.; SOARES, M. D. S. **Qualidade de software:** aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. [S.l.]: Novatec Editora, 2007.

KOSSOSKI, C.; SIMÃO, J. M.; STADZISZ, P. C. Introdução ao teste funcional de software no Paradigma Orientado a Notificações. **VI Congreso Internacional de Computación y Telecomunicaciones**, v. 1, p. 136-143, out. 2014.

KUMAR, V.; LEONARD, N.; MORSE, A. S. **Cooperative Control:** A Post-Workshop Volume, 2003 Block Island Workshop on Cooperative Control. [S.l.]: Springer, 2004. ISBN 9783540228615 LCCN: 2004112294.

LDRA. LDRA Tool Suite, 2014. Disponivel em: <http://www.ldra.com/en/software-quality-test-tools/group/by-product-module/ldra-tool-suite>. Acesso em: 14 dez. 2014.

LEE, P. Y.; CHENG, A. M. HAL - A Faster Match Algorithm. **IEEE Transaction on Knowledge and Data Engineering**, p. 1047-1058, 2002.

LIBALLEG. Allegro: A game programming library. **Liballeg**, 2014. Disponivel em: <http://www.liballeg.org/readme.html>. Acesso em: 1 Outubro 2014.

LIMA, A. S. **UML 2.0:** do requisito à solução. 3. ed. [S.l.]: [s.n.], 2008.

LINHARES, R. R. **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao Paradigma Orientado a Notificações**. Tese (Doutorado) - Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). [S.l.]. 2014.

LINHARES, R. R. et al. **Comparações entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sob o contexto de um simulador de sistema telefônico**. III Congresso Intern. de Computación y Telecom - COMTEL. [S.l.]: [s.n.]. 2011.

LINHARES, R. R.; SIMÃO, J. M.; STADZISZ, P. C. **Pedido de patente:** Arquitetura de Computador Orientada a Notificações - ARQPON. BR1020140040706, 2014.

MA, Y.-S.; KWON, Y.-R.; OFFUTT, J. **Inter-Class Mutation Operators for Java**. Proceedings of the 13th International Symposium on Software Reliability Engineering. Washington, DC, USA: IEEE Computer Society. 2002. p. 352--.

MALDONADO, J. C. **Critérios potenciais usos: Uma contribuição ao teste estrutural de software**. Faculdade de Engenharia Elétrica, UNICAMP. [S.l.]. 1991.

MARETIS, D. K. Highly Parallel Architectures in Meteorological Applications. In: HOFFMANN, G.-R.; MARETIS, D. **The Dawn of Massively Parallel Processing in Meteorology**. [S.l.]: Springer Berlin Heidelberg, 1990. p. 372-376. ISBN DOI: 10.1007/978-3-642-84020-3_24. Disponivel em: <http://dx.doi.org/10.1007/978-3-642-84020-3_24>.

MCCABE, T. J. A Complexity Measure. **Software Engineering, IEEE Transactions on**, v. SE-2, n. 4, p. 308-320, Dec 1976. ISSN 0098-5589 DOI: 10.1109/TSE.1976.233837.

MELO, L. C. V. **Relatório da adaptação do Paradigma Orientado a Notificações - PON para suporte a desenvolvimento de sistemas de lógica fuzzy**. Programa de Pós-Graduação em Computação Aplicada (PPGCA), Universidade Tecnológica Federal do Paraná (UTFPR). [S.l.]. 2013.

MYERS, G. J. **Art of Software Testing**. New York, NY, USA: John Wiley & Sons, Inc., 1979. ISBN 0471043281.

MYERS, G. J. et al. **The Art of Software Testing**. 2. ed. New York, NY, EUA: John Wiley and Sons, 2004.

NAZARETH, D. L. Investigating the applicability of Petri nets for rule-based system verification. **Knowledge and Data Engineering, IEEE Transactions on**, v. 5, n. 3, p. 402-415, jun. 1993. ISSN 1041-4347 DOI: 10.1109/69.224193.

NAZARETH, D. L.; KENNEDY, M. H. Verification of rule-based knowledge using directed graphs. **Kwowledge Acquisition**, p. 339-360, ago. 1991.

OFFUTT, A. J. Investigations of the Software Testing Coupling Effect. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v. 1, n. 1, p. 5-20, jan. 1992. ISSN 1049-331X DOI: 10.1145/125489.125473. Disponivel em: <http://doi.acm.org/10.1145/125489.125473>.

OFFUTT, A. J. et al. An Experimental Determination of Sufficient Mutant Operators. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v. 5, n. 2, p. 99-118, abr. 1996. ISSN 1049-331X DOI: 10.1145/227607.227610. Disponivel em: <http://doi.acm.org/10.1145/227607.227610>.

OFFUTT, A. J.; HAYES, J. H. A Semantic Model of Program Faults. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v. 21, n. 3, p. 195-200, may 1996. ISSN 0163-5948 DOI: 10.1145/226295.226317. Disponivel em: <http://doi.acm.org/10.1145/226295.226317>.

PAN, J.; SOUZA, G. N. D.; KAK, A. C. FuzzyShell: a large-scale expert system shell using fuzzy logic for uncertainty reasoning. **Fuzzy Systems, IEEE Transactions on**, v. 6, n. 4, p. 563-581, Nov 1998. ISSN 1063-6706 DOI: 10.1109/91.728455.

PETERS, E. **Coprocessador para aceleração de aplicações desenvolvidas utilizando paradigma orientado a notificações**. Dissertação (Mestrado) - Curso de

Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2012.

POTTER, M. D. **Set Theory and Its Philosophy:** A Critical Introduction. [S.l.]: Oxford University Press, 2004.

PRESSMAN, R. **Software Engineering:** A Practitioner's Approach. 7. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN 0073375977, 9780073375977.

RAPPS, S.; WEYUKER, E. Data Flow Analysis Techniques for Test Data Selection. **Department of Computer Science, Courant Institute of Mathematical Sciences**, New York, 1982.

RAPPS, S.; WEYUKER, E. J. Selecting Software Test Data Using Data Flow Information. **Software Engineering, IEEE Transactions on**, v. SE-11, n. 4, p. 367-375, April 1985. ISSN 0098-5589 DOI: 10.1109/TSE.1985.232226.

RONSZCKA, A. F. **Contribuição para a Concepção de Aplicações no Paradigma Orientado a Notificações (PON) sob o viés de Padrões**. Dissertação (Mestrado) - Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2012.

RONSZCKA, A. F. et al. Comparações quantitativas e qualitativas entre o Paradigma Orientado a Objetos e o Paradigma Orientado a Notificações sobre um simulador de jogo. **III Congresso Intern. de Computación y Telecom - COMTEL**, 2011.

RONSZCKA, A. F. et al. **Compilador para o Paradigma Orientado a Notificações**. UTFPR (CPGEI/PPGCA). [S.l.]. 2013.

ROSS, D. T. Structured Analysis (SA): A Language for Communicating Ideas. In: GRIES, D. **Programming Methodology**. [S.l.]: Springer New York, 1978. p. 388-421. ISBN DOI: 10.1007/978-1-4612-6315-9_27. Disponivel em: <http://dx.doi.org/10.1007/978-1-4612-6315-9_27>.

ROY, P. V. Programming paradigms for dummies: What every programmer should know. **New Computational Paradigms for Computer Music**, p. 9, 2009.

ROY, P. V.; HARIDI, S. **Concepts, Techniques, and Models of Computer Programming**. Cambridge, MA, USA: MIT Press, 2004. ISBN 0262220695.

RUSSEL, S.; NORVIG, P. **Artificial Intelligence:** a modern approach. 2. ed. [S.l.]: Prentice Hall, 2003.

SCOTT, M. L. **Programming Language Pragmatics**. 2. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 2000.

SCOTT, M. L. **Programming Language Pragmatics, Third Edition**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN 0123745144, 9780123745149.

SEBESTA, R. W. **Concepts of Programming Languages**. 10th. ed. [S.l.]: Pearson, 2012. ISBN 0273769103, 9780273769101.

SIMÃO, J. M. **Proposta de uma Arquitetura de Controle para Sistemas Flexíveis de Manufatura Baseada em Regras e Agentes**. Dissertação (Mestrado) - Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2001.

SIMÃO, J. M. **A Contribution to the Development of a HMS Simulation Tool and Proposition of a Metal-Model for Holonic Control**. Centro Federal de Educação Tecnológica do Paraná (CEFET-PR), Université Henri Poincaré (UHP). Curitiba, Brazil. 2005.

SIMÃO, J. M. et al. Comparações entre duas materializações do Paradigma Orientado a Notificações (PON): Framework PON Prototipal versus Framework PON Primário. **IV Congresso Intern. de Computación y Telecom - COMTEL**, 2012.

SIMÃO, J. M. et al. Notification Oriented and Object Oriented Paradigm comparison via Sale System. **Journal of Software Engineering and Applications – JSEA**, 2012.

SIMÃO, J. M. et al. Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study. **Journal of Software Engineering and Applications (JSEA)**, v. 5, p. 402, 2012.

SIMÃO, J. M.; STADZISZ, P. C. **Paradigma Orientado a Notificações (PON) Uma Técnica de Composição e Execução de Software Orientado a Notificações**. PI015080004262, 2008.

SIMÃO, J. M.; STADZISZ, P. C. Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues. **IEEE Transactions on Systems, Man and Cybernetics**, v. 39, p. 238-250, 2009.

SIMÃO, J. M.; STADZISZ, P. C. **Pedido de patente:** Mecanismo de Resolução de Conflito e Garantia de Determinismo para o Paradigma Orientado a Notificações (PON). PI10002960, fev. 2010.

SOMMERVILLE, I. **Software Engineering (9th Edition)**. [S.l.]: Addison-Wesley, 2011.

TAHCHIEV, P. et al. **JUnit in Action, Second Edition**. 2nd. ed. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN 1935182021, 9781935182023.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

THAKUR, D. Software Testing Techniques. **Computer Notes**, 2014. Disponivel em: <http://ecomputernotes.com/software-engineering/testing-techniques>. Acesso em: 09 set. 2014.

TIANFIELD, H. **A new framework of holonic self-organization for multi-agent systems**. Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on. [S.l.]: [s.n.]. Oct 2007. p. 753-758.

TUTTLE, S. M.; EICK, C. F. **Suggesting causes of faults in data-driven rule-based systems**. Tools with Artificial Intelligence, 1992. TAI '92, Proceedings., Fourth International Conference on. [S.l.]: [s.n.]. Nov 1992. p. 413-416.

VALENÇA, G. Z. **Contribuição para materialização do Paradigma Orientado Notificações (PON) via Framework e Wizard**. Dissertação (Mestrado) - Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2012.

VALENÇA, G. Z. et al. Framework PON, Avanços e Comparações. **III Simpósio de Computação Aplicada**, 2011.

WATT, D. A. **Programming Language Design Concepts**. [S.l.]: Wiley India Pvt. Limited, 2004. ISBN 9788126505272 LCCN: 2003026236.

WEISSENGEIST, T. Allegro para iniciantes. **Centro de informática - UFPE**, 2014. Disponivel em: <http://www.cin.ufpe.br/~csa3/Arquivos/Monitoria%20Software/Allegro/allegro_manual.asp.htm>. Acesso em: 12 set. 2014.

WIECHETECK, L. V. B. **Método para Projeto de Software Usando o Paradigma Orientado a Notificações - PON**. Dissertação (Mestrado) - Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, Brasil. 2011.

WIECHETECK, L. V. B.; STADZISZ, P. C.; SIMÃO, J. M. Um Perfil UML para o Paradigma Orientado a Notificações (PON). **III Congresso Intern. de Computación y Telecom - COMTEL**, 2011.

XAVIER, R. D. **Paradigmas de desenvolvimento de software: comparação entre abordagens orientada a eventos e orientada a notificações**. Seminário de qualificação III (Mestrado) - Programa de Pós-graduação em Computação Aplicada (PPGCA), Universidade Tecnológica Federal do Paraná (UTFPR). [S.l.]. 2014.

APÊNDICE A – Diagrama de classes completo

APÊNDICE B – *Rules* desenvolvidas para o software

Este apêndice apresenta a continuação do Levantamento de *Rules* criadas para o software apresentado no Capítulo 3.

A Tabela 113 apresenta a descrição da *Rule* que movimentará o avião para a direita. Esta *Rule* implementará, parcialmente, o caso de uso Controlar Avião por meio dos requisitos RF008, RNF004 e RNF005.

Tabela 113 – *Rule* 5: Movimentar Avião à direita

Passo	Resposta
1 – Propósito	Movimentar Avião à direita
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero O Avião não pode estar no limite da borda direita da tela Jogador precisa ter pressionado o botão direito
3 – Consequência	O Avião move-se para a direita quando o botão direito for pressionado

A Tabela 114 apresenta a descrição da *Rule* que permitirá o avião disparar projéteis contra os inimigos (helicópteros). Esta *Rule* implementará, integralmente, o caso de uso "Controlar Avião" por meio do requisito RF002.

Tabela 114 – *Rule* 6: Disparar projétil contra o Helicóptero

Passo	Resposta
1 – Propósito	Disparar projétil contra o Helicóptero
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero Jogador precisa ter pressionado o botão de disparo
3 – Consequência	O Avião dispara projétil

A Tabela 115 apresenta a descrição da *Rule* que permitirá os inimigos dispararem contra o avião. Esta *Rule* implementará, parcialmente, o caso de uso "Controlar Helicóptero" por meio do requisito RF005.

Tabela 115 – *Rule* 7: Disparar projétil contra o Avião

Passo	Resposta
1 – Propósito	Helicóptero dispara projétil contra Avião
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Helicóptero deve ser maior que zero O Helicóptero não pode ter disparado um projétil até dois segundos depois do último disparo
3 – Consequência	O Helicóptero dispara projétil

A Tabela 116 apresenta a descrição da *Rule* que apresentará (desenho) de cada helicóptero na tela. Esta *Rule* implementará, parcialmente, o caso de uso "Controlar Apresentação" por meio do requisito RF013.

Tabela 116 – *Rule* 8: Apresentar o Helicóptero no tela

Passo	Resposta
1 – Propósito	Desenhar o Helicóptero na tela
2 – Fatores	O jogo precisa estar com estado jogando ou pausado O Helicóptero precisa ter mais que zero de pontos de vida
3 – Consequência	O Helicóptero será desenhado na tela

A Tabela 117 apresenta a descrição da *Rule* que decrementará os pontos de vida do avião. Esta *Rule* implementará, parcialmente, o caso de uso "Detectar Colisão de Projétil" por meio do requisito RF006.

Tabela 117 – *Rule* 9: Decrementar pontos de vida do Avião

Passo	Resposta
1 – Propósito	Decrementar pontos de vida do Avião
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero O Avião precisa ter colidido com um projétil que fora disparado
3 – Consequência	Os pontos de vida do Avião serão decrementados no valor de 10

A Tabela 118 apresenta a descrição da *Rule* que decrementará os pontos de vida do helicóptero. Esta *Rule* implementará, parcialmente, o caso de uso "Detectar Colisão de Projétil" por meio do requisito RF006.

Tabela 118 – *Rule* 10: Decrementar pontos de vida do Helicóptero

Passo	Resposta
1 – Propósito	Decrementar pontos de vida do Helicóptero
2 – Fatores	O sistema precisa estar com estado jogando O Helicóptero precisa ter mais que zero de pontos de vida O Helicóptero precisa ter colidido com um projétil que fora disparado
3 – Consequência	Os pontos de vida do Helicóptero serão decrementados no valor de 10

A Tabela 119 apresenta a descrição da *Rule* que eliminará os helicópteros abatidos. Esta *Rule* implementará, parcialmente, o caso de uso "Controlar Helicóptero" por meio do requisito RF010.

Tabela 119 – *Rule* 11: Eliminar o Helicóptero

Passo	Resposta
1 – Propósito	Eliminar o Helicóptero
2 – Fatores	O sistema precisa estar com estado jogando O Helicóptero precisa ter zero ou menos de pontos de vida
3 – Consequência	O Helicóptero é eliminado do jogo

A Tabela 120 apresenta a descrição da *Rule* que eliminará o jogador. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Avião” por meio do requisito RF010.

Tabela 120 – *Rule* 12: Eliminar o jogador

Passo	Resposta
1 – Propósito	Eliminar o jogador
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião precisa ser igual ou menor que zero
3 – Consequência	O Avião é eliminado do jogo, causando o fim de jogo

A Tabela 121 apresenta a descrição da *Rule* que permitirá ao jogador pausar o jogo. Esta *Rule* implementará, parcialmente, o caso de uso “Pausar e Continuar Jogo” por meio do requisito RF009.

Tabela 121 – *Rule* 13: Pausar o jogo

Passo	Resposta
1 – Propósito	Pausar o jogo
2 – Fatores	O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero O jogador precisa pressionar o botão de pausa
3 – Consequência	O jogo é pausado.

A Tabela 122 apresenta a descrição da *Rule* que permitirá o jogador continuar um jogo pausado. Esta *Rule* implementará, parcialmente, o caso de uso “Pausar e Continuar Jogo”, por meio do requisito RF009.

Tabela 122 – *Rule* 14: Continuar o jogo

Passo	Resposta
1 – Propósito	Continuar o jogo
2 – Fatores	O sistema precisa estar pausado O jogador precisa pressionar o botão para continuar
3 – Consequência	O jogo é continuado

A Tabela 123 apresenta a descrição da *Rule* que irá terminar o jogo. Esta *Rule* implementará, integralmente, o caso de uso "Parar Jogo" por meio do requisito RF014.

Tabela 123 – *Rule* 15: Parar o jogo

Passo	Resposta
1 – Propósito	Parar o jogo
2 – Fatores	O sistema precisa estar com estado pausado ou jogando O jogador pressiona o botão parar
3 – Consequência	O jogo é parado

A Tabela 124 apresenta a descrição da *Rule* que limpará a tela do jogo durante a execução. Esta *Rule* implementará, parcialmente, o caso de uso "Controlar Apresentação" por meio do requisito RF015.

Tabela 124 – *Rule* 16: Limpar a tela (Allegro)

Passo	Resposta
1 – Propósito	Allegro – limpar a tela
2 – Fatores	O sistema precisa estar com estado pausado ou jogando
3 – Consequência	A tela é apagada completamente

A Tabela 125 apresenta a descrição da *Rule* que desenhará todos os elementos do jogo. Esta *Rule* implementará, parcialmente, o caso de uso "Controlar Apresentação" por meio do requisito RF015.

Tabela 125 – *Rule* 17: Desenhar todos os elementos (Allegro)

Passo	Resposta
1 – Propósito	Allegro – Desenhar todos os elementos
2 – Fatores	O sistema precisa estar com estado pausado ou jogando
3 – Consequência	Todos os elementos bitmaps serão desenhados

A Tabela 126 apresenta a descrição da *Rule* que aguardará o envio de comandos de teclado feitos pelo jogador. Esta *Rule* implementará, integralmente, o caso de uso "Realizar Leitura de Teclado" por meio dos requisitos RF015 e RF016.

Tabela 126 – *Rule* 18: Comandos do teclado (Allegro)

Passo	Resposta
1 – Propósito	Allegro – Aguardar comandos enviados pelo jogador
2 – Fatores	O sistema precisa estar com estado pausado ou jogando
3 – Consequência	O sistema realiza ações de acordo com os comandos enviados pelo jogador

A Tabela 127 apresenta a descrição da *Rule* que desenhará o avião enquanto ele tiver pontos de vida superiores à zero. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Apresentação” por meio do requisito RF015.

Tabela 127 – *Rule* 19: Desenhar o Avião (Allegro)

Passo	Resposta
1 – Propósito	Allegro – desenhar o Avião na tela
2 – Fatores	O sistema precisa estar com estado pausado ou jogando O Avião precisa ter mais que zero de pontos de vida
3 – Consequência	O Avião é desenhado na tela

A Tabela 128 apresenta a descrição da *Rule* que desenhará o cenário de fundo. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Apresentação” por meio do requisito RF015.

Tabela 128 – *Rule* 20: Desenhar o cenário de fundo

Passo	Resposta
1 – Propósito	Allegro – desenhar o cenário de fundo
2 – Fatores	O sistema precisa estar com estado pausado ou jogando
3 – Consequência	O sistema desenha o cenário de fundo da fase

A Tabela 129 apresenta a descrição da *Rule* que desenhará os projéteis disparados. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Apresentação” por meio do requisito RF015.

Tabela 129 – *Rule* 21: Desenhar projétil

Passo	Resposta
1 – Propósito	Allegro – Desenhar projétil na tela
2 – Fatores	O sistema pode estar com estado jogando ou pausado
3 – Consequência	O projétil é desenhado na tela

A Tabela 130 apresenta a descrição da *Rule* que irá detectar a colisão entre projétil e helicóptero. Esta *Rule* implementará, parcialmente, o caso de uso “Detectar Colisão de Projétil” por meio do requisito RF006.

Tabela 130 – *Rule* 22: Detectar colisão com Helicóptero (Allegro)

Passo	Resposta
1 – Propósito	Allegro – detectar colisão entre projétil e um Helicóptero
2 – Fatores	Na tela precisa haver algum projétil que fora disparado O sistema precisa estar com estado jogando O Helicóptero precisa ter mais que zero de pontos de vida O projétil deve estar contido dentro da área delimitada do Helicóptero na tela
3 – Consequência	O sistema detecta colisão quando um projétil é desenhado na área delimitada do Helicóptero

A Tabela 131 apresenta a descrição da *Rule* que irá detectar a colisão entre projétil e avião. Esta *Rule* implementará, parcialmente, o caso de uso “Detectar Colisão de Projétil” por meio do requisito RF006.

Tabela 131 – *Rule* 23: Detectar colisão de projétil com Avião (Allegro)

Passo	Resposta
1 – Propósito	Allegro – detectar colisão entre projétil e o Avião (jogador)
2 – Fatores	Na tela precisa haver algum projétil que fora disparado O sistema precisa estar com estado jogando O valor dos pontos de vida do Avião deve ser maior que zero O projétil deve estar contido dentro da área delimitada do Helicóptero na tela
3 – Consequência	O sistema detecta colisão quando um projétil é desenhado na área delimitada do Avião

A Tabela 132 apresenta a descrição da *Rule* que irá apresentar a trajetória ascendente de projétil disparado pelo avião. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Apresentação” por meio do requisito RF012.

Tabela 132 – *Rule* 24: Apresentar a trajetória ascendente de um projétil disparado

Passo	Resposta
1 – Propósito	Allegro – apresentar trajetória de projétil na tela (ascendente)
2 – Fatores	O sistema precisa estar com estado jogando O Avião precisa ter mais que zero de pontos de vida (ascendente porque o Avião dispara de baixo para cima)
3 – Consequência	O sistema apresenta um projétil seguindo sua trajetória ascendente

A Tabela 133 apresenta a descrição da *Rule* que irá movimentar o helicóptero. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Helicóptero” por meio do requisito RF004.

Tabela 133 – *Rule* 25: Movimentar Helicóptero na tela

Passo	Resposta
1 – Propósito	Allegro – desenhar a helicóptero na tela
2 – Fatores	O sistema precisa estar com estado jogando O Helicóptero precisa estar com valor de pontos de vida maior que zero
3 – Consequência	O Helicóptero é desenhado na tela

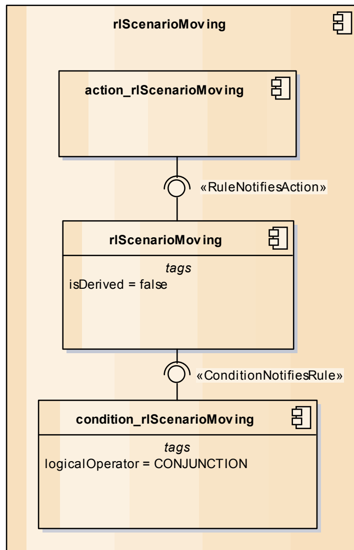
A Tabela 134 apresenta a descrição da *Rule* que irá apresentar a trajetória descendente de um projétil disparado pelo helicóptero. Esta *Rule* implementará, parcialmente, o caso de uso “Controlar Apresentação” por meio do requisito RF012.

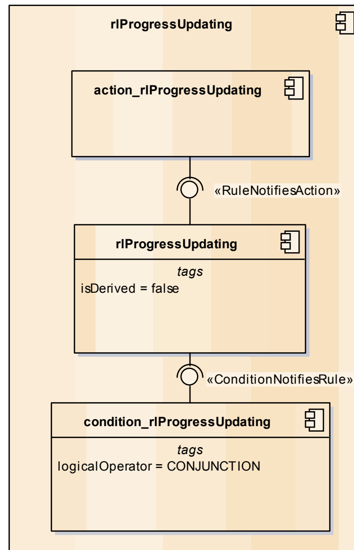
Tabela 134 – *Rule* 26: Apresentar a trajetória descendente de um projétil disparado

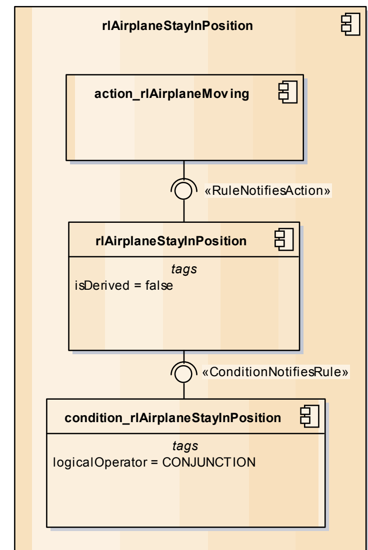
Passo	Resposta
1 – Propósito	Allegro – apresentar trajetória de projétil na tela (descendente)
2 – Fatores	O sistema precisa estar com estado jogando O Helicóptero precisa ter mais que zero de pontos de vida (descendente porque o Helicóptero dispara de cima para baixo)
3 – Consequência	O sistema apresenta um projétil seguindo trajetória descendente

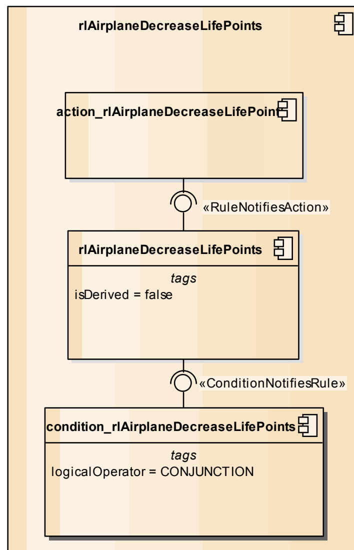
APÊNDICE C - Diagramas de componentes

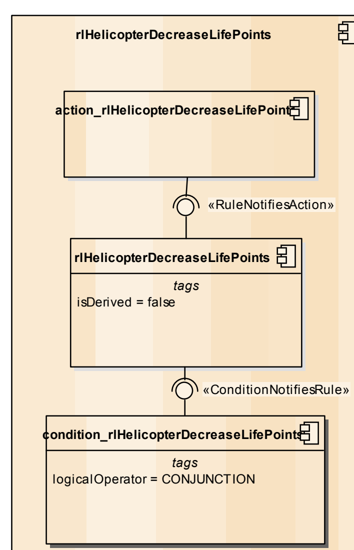
composite structure Estruturas Internas - Passo 1

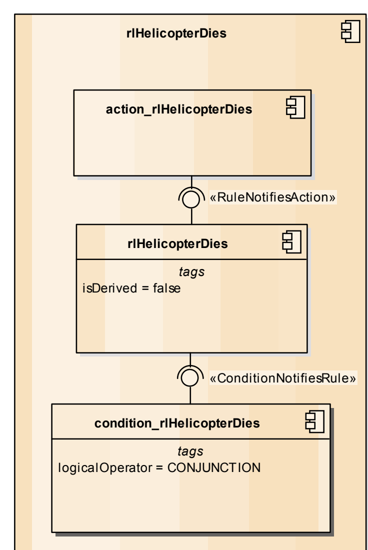


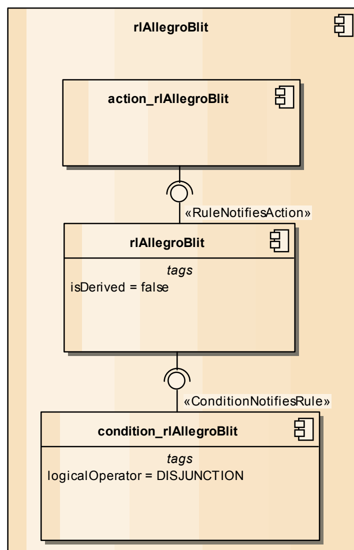


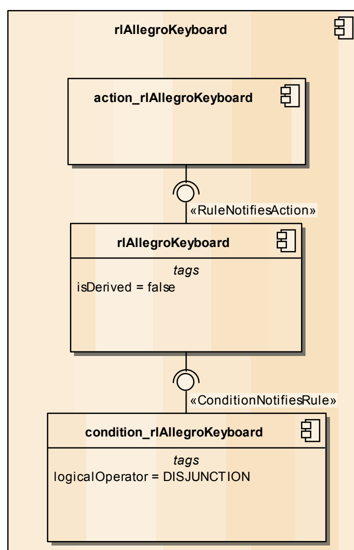


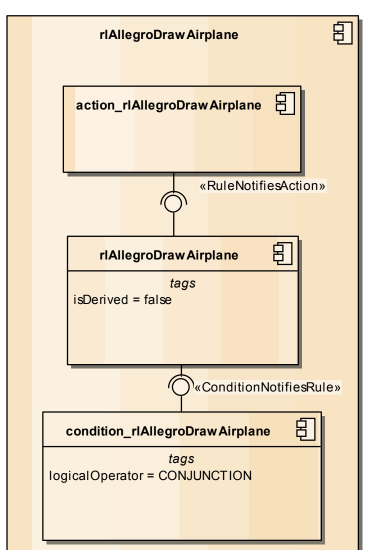




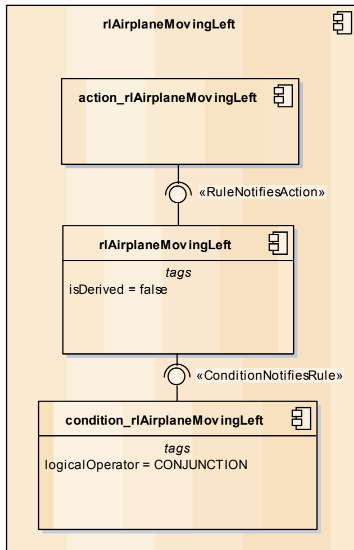


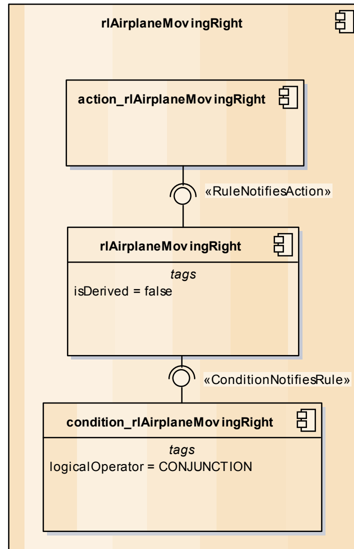


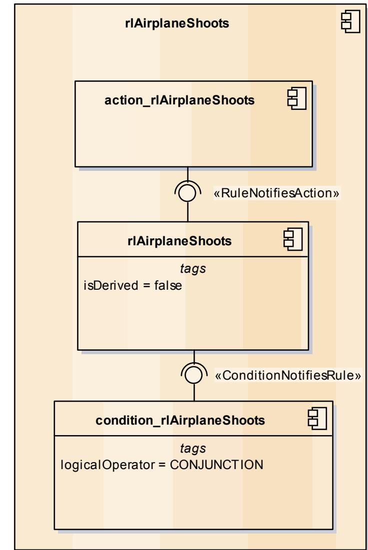


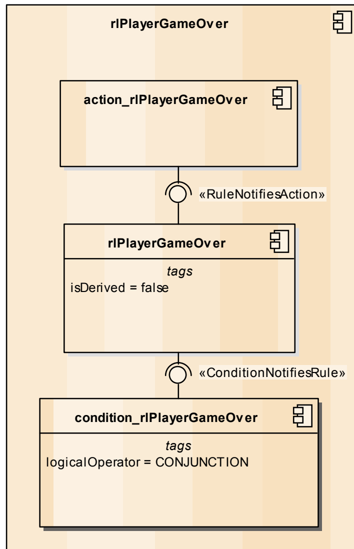


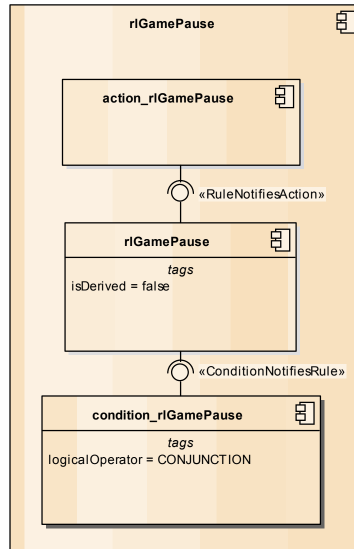
composite structure Estruturas Internas - Passo 1

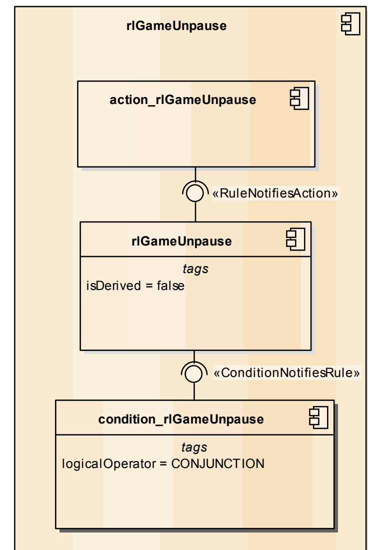


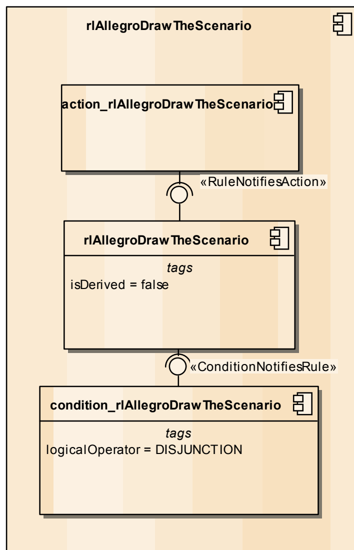


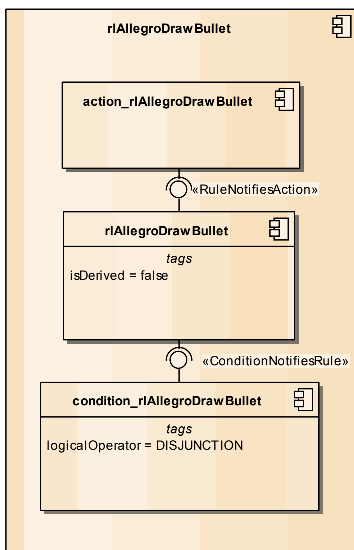


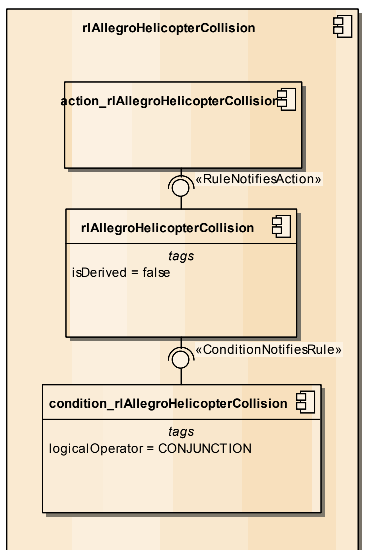




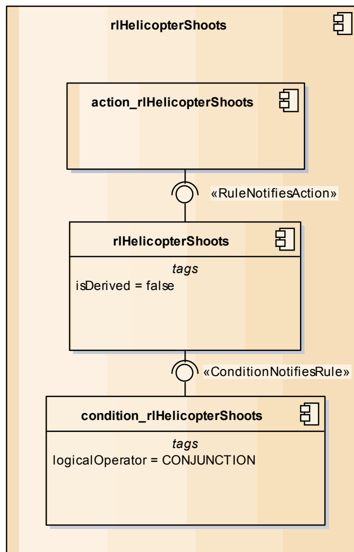


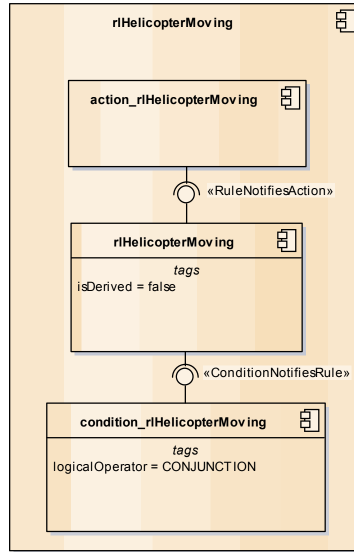


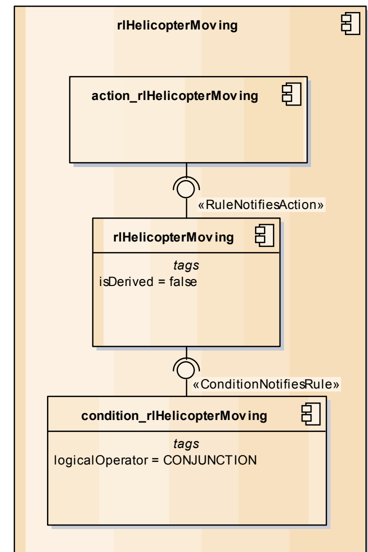


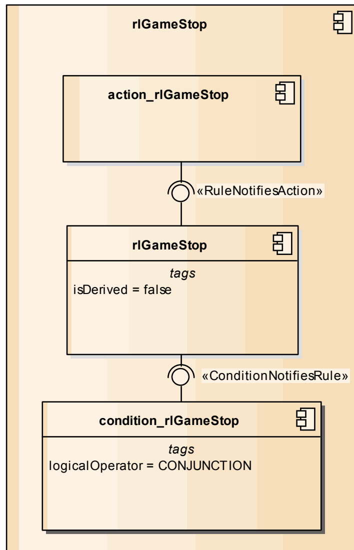


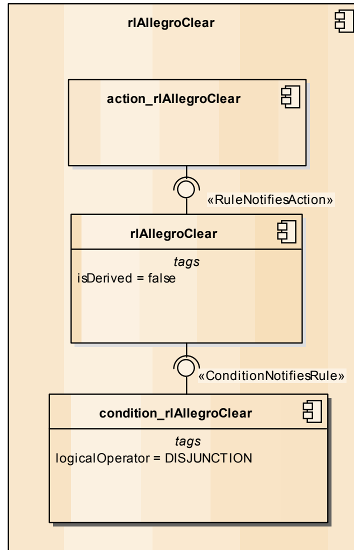
composite structure Estruturas Internas - Passo 1

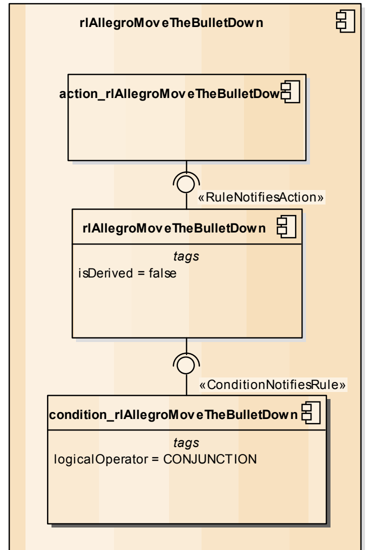


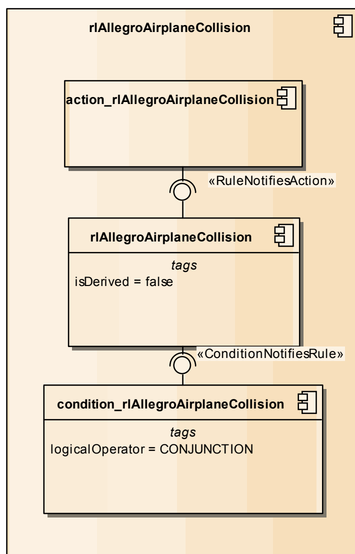


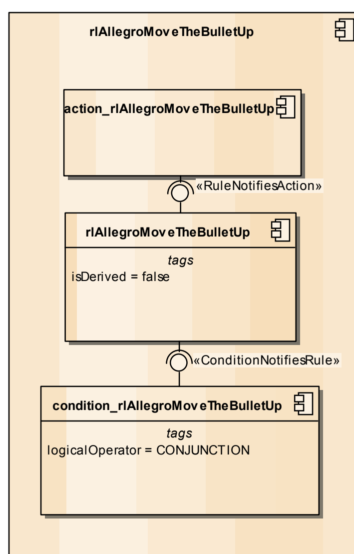




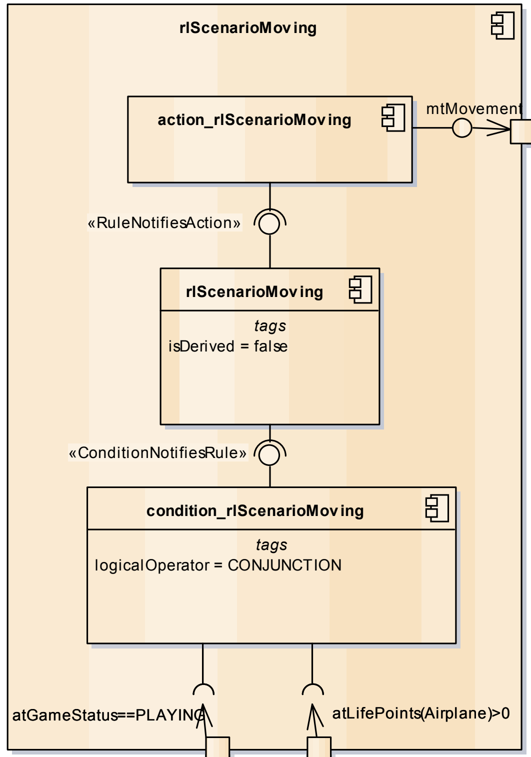


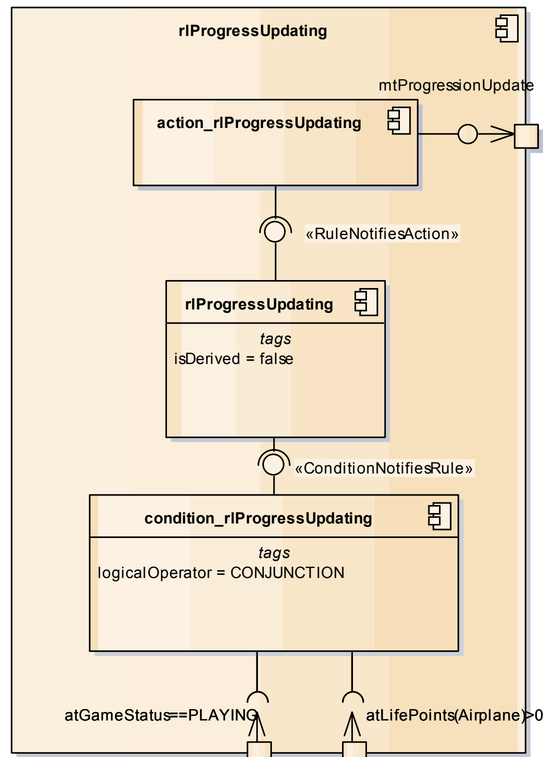


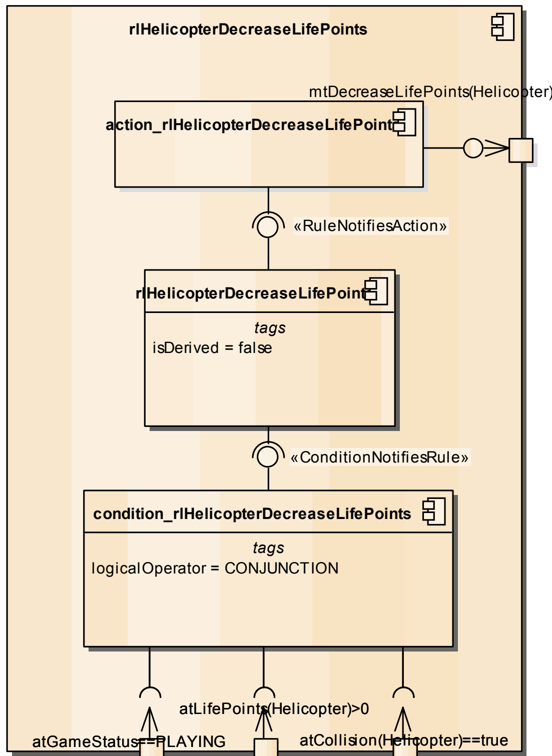


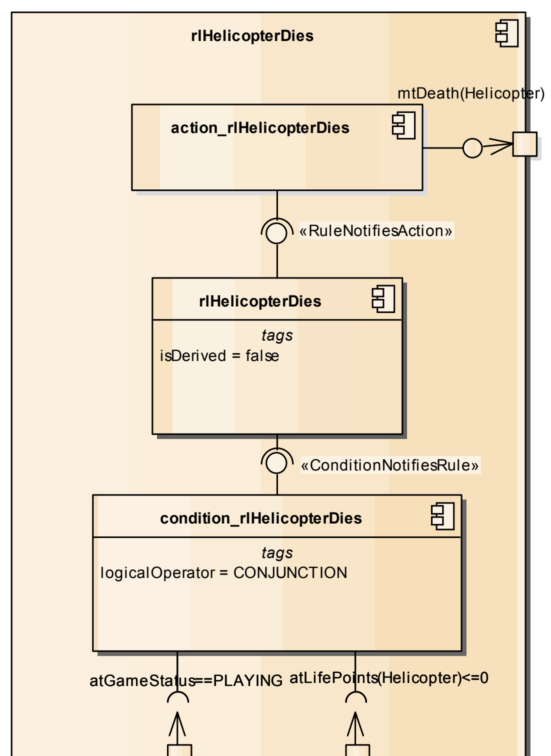


composite structure Estruturas Internas - Passo 2

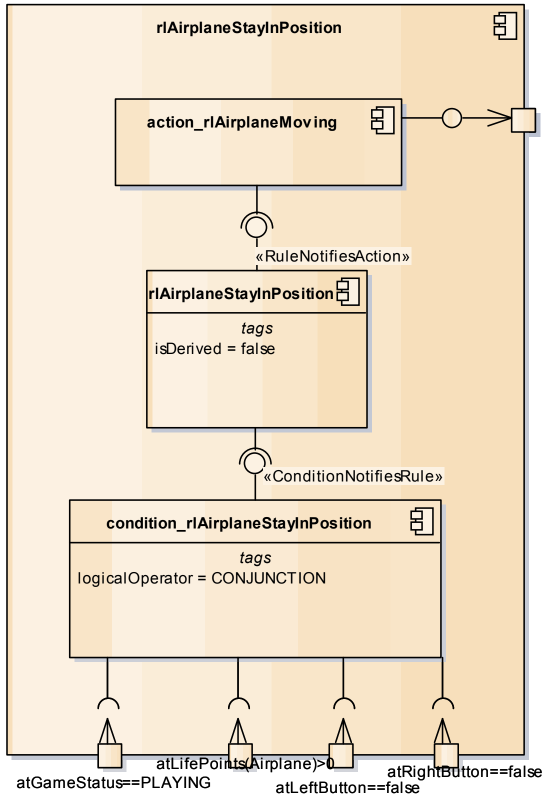


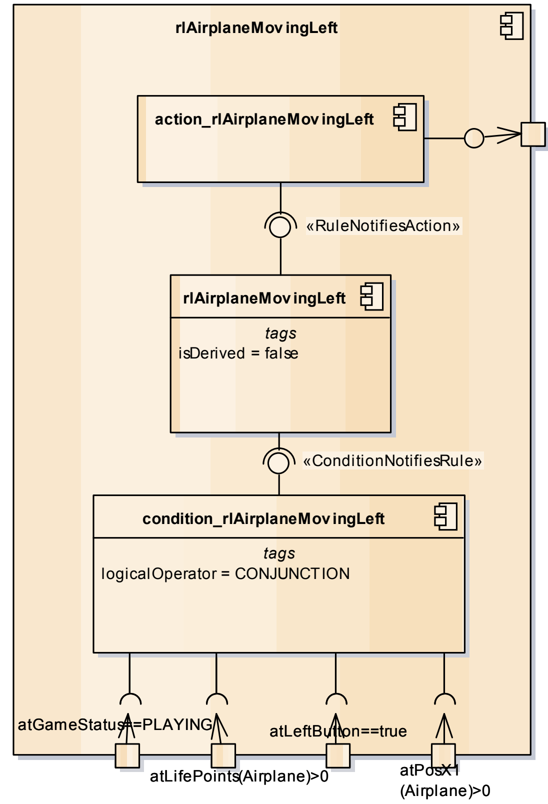


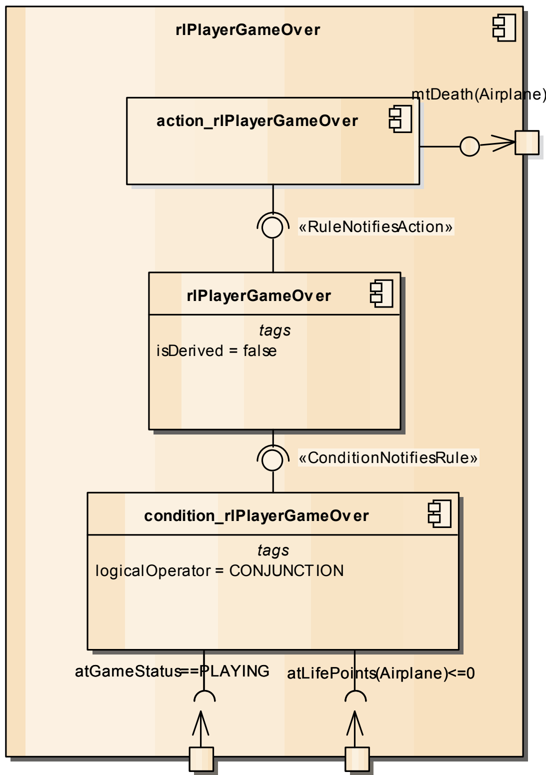


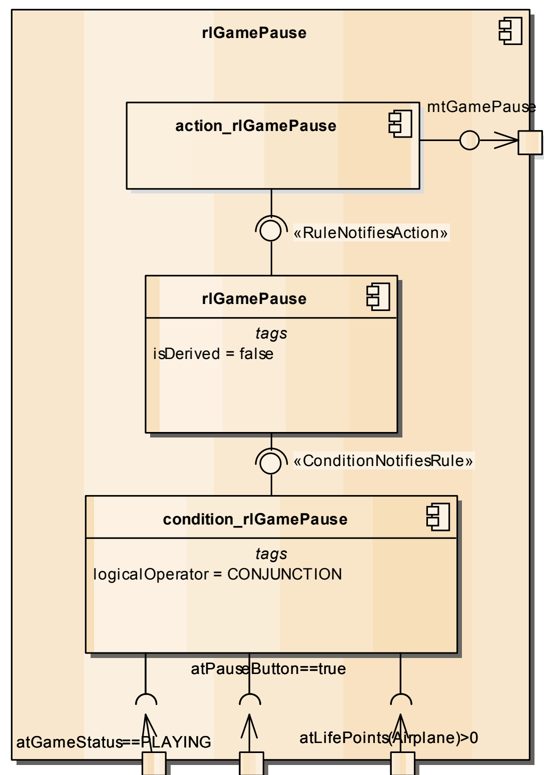


composite structure Estruturas Internas - Passo 2

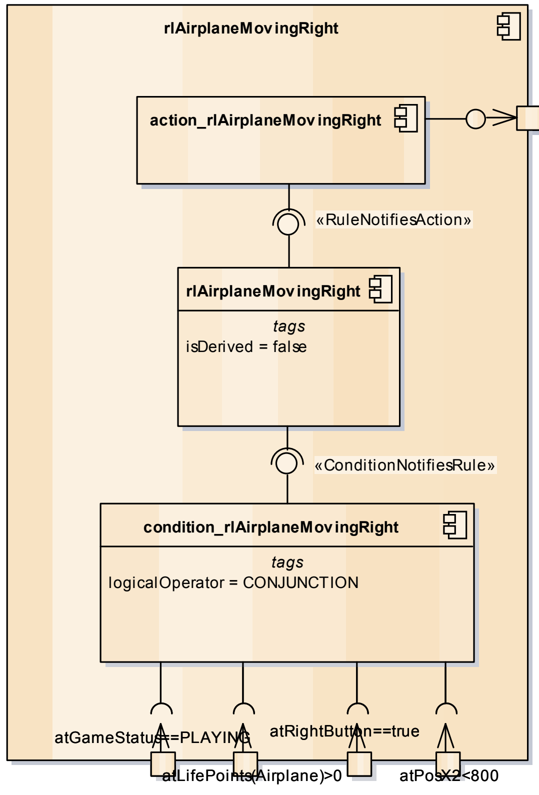


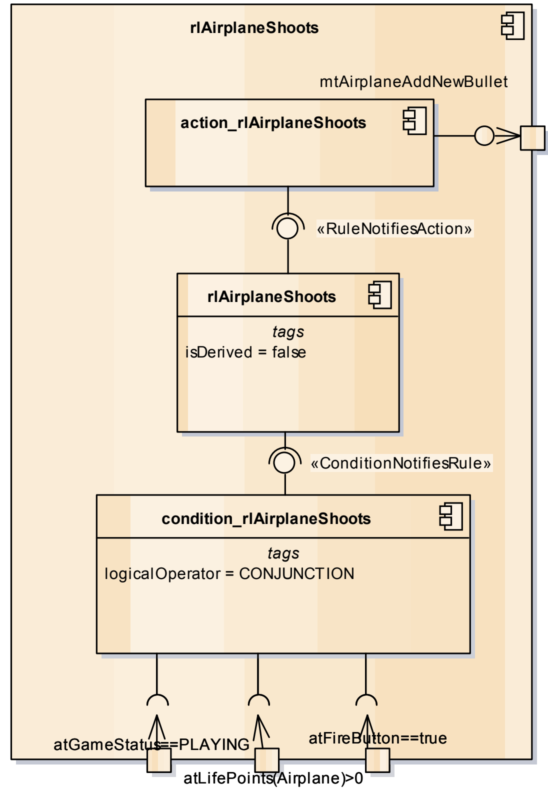


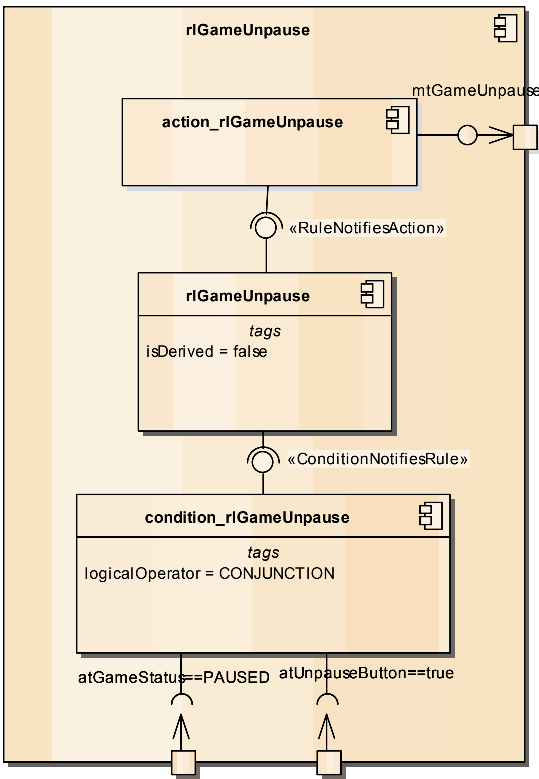


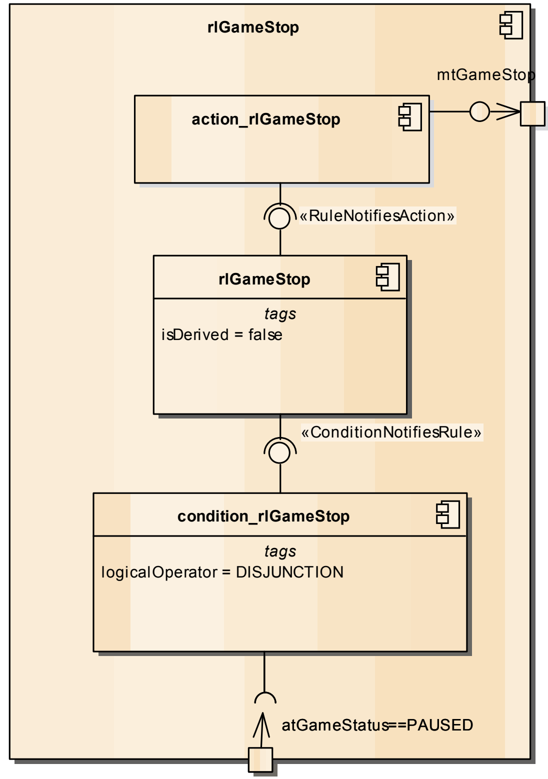


composite structure Estruturas Internas - Passo 2

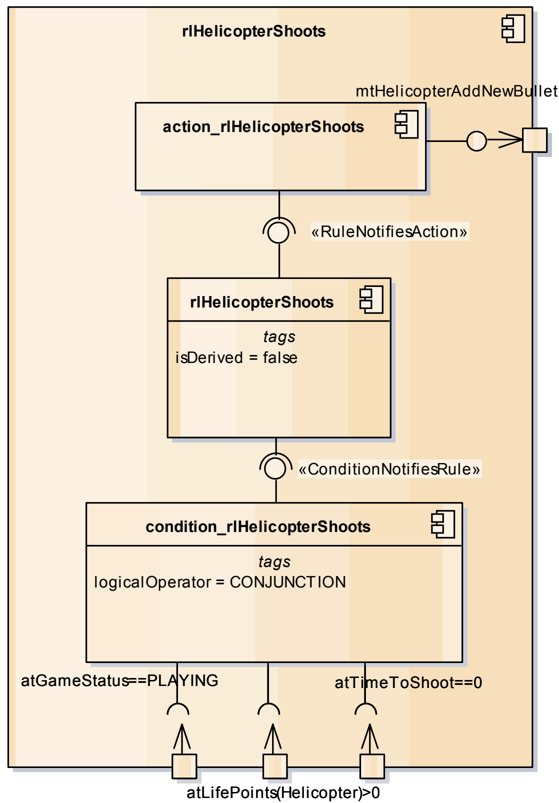


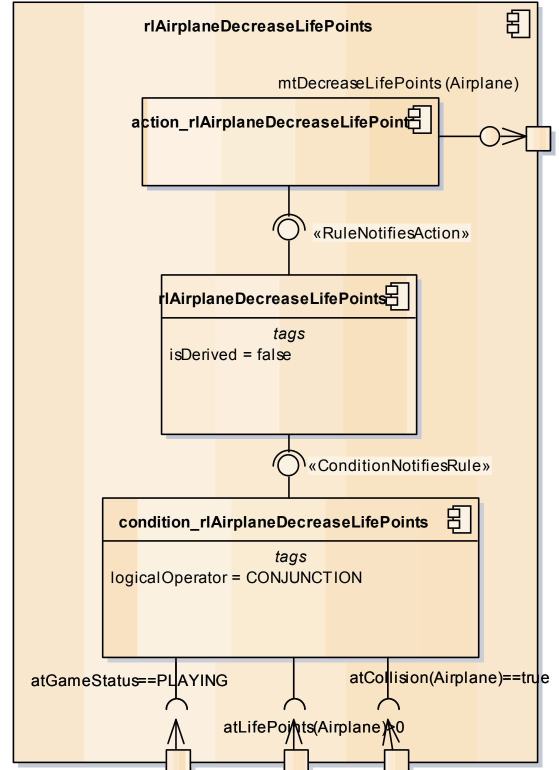


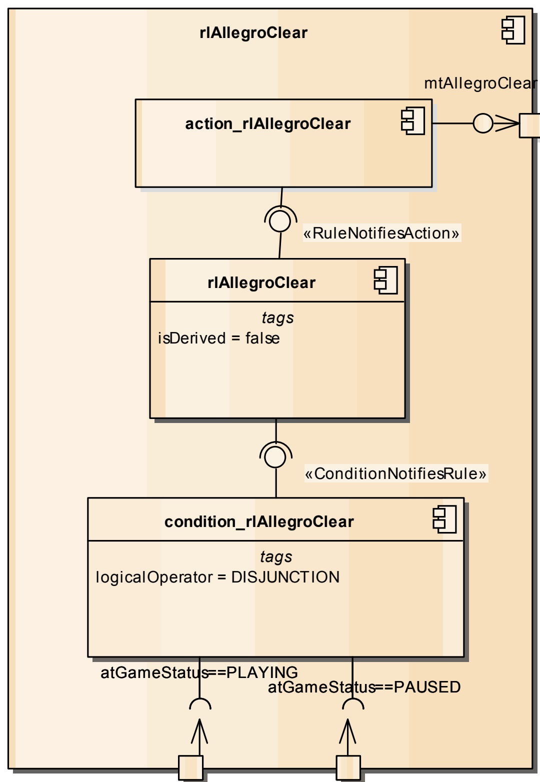


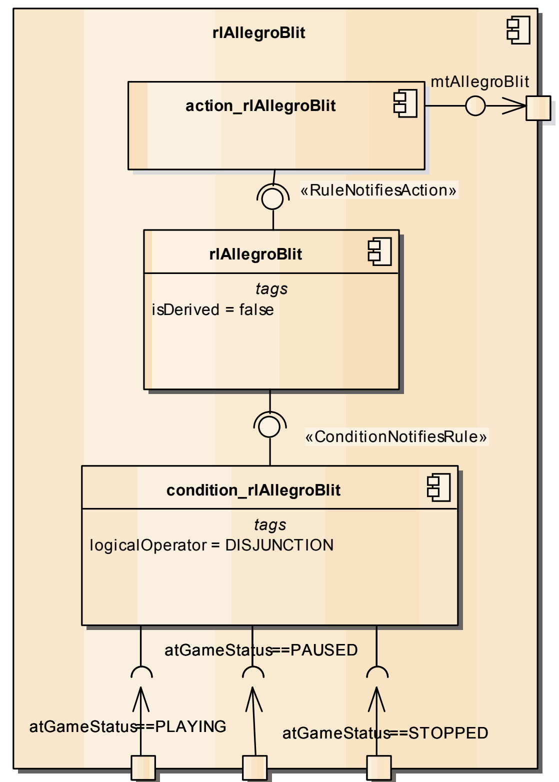


composite structure Estruturas Internas - Passo 2

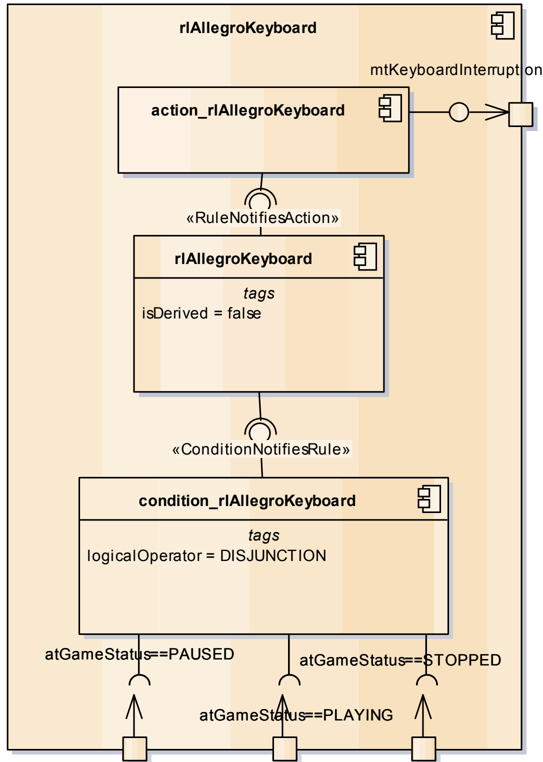


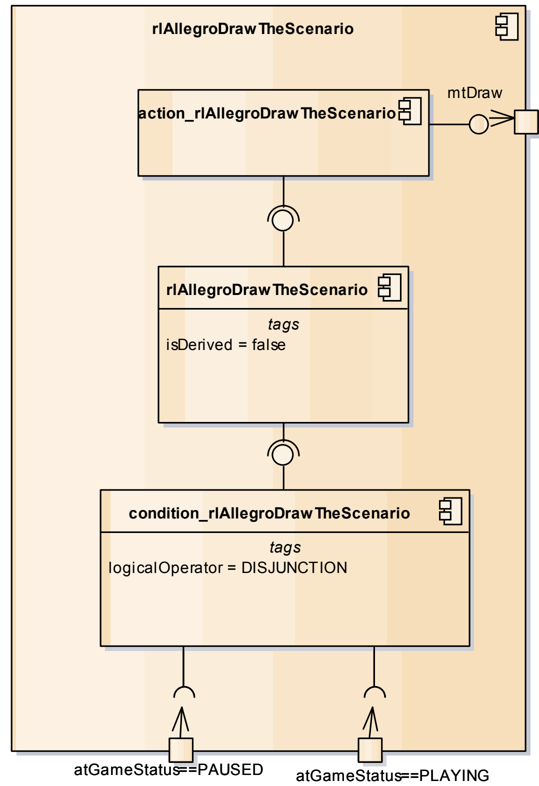


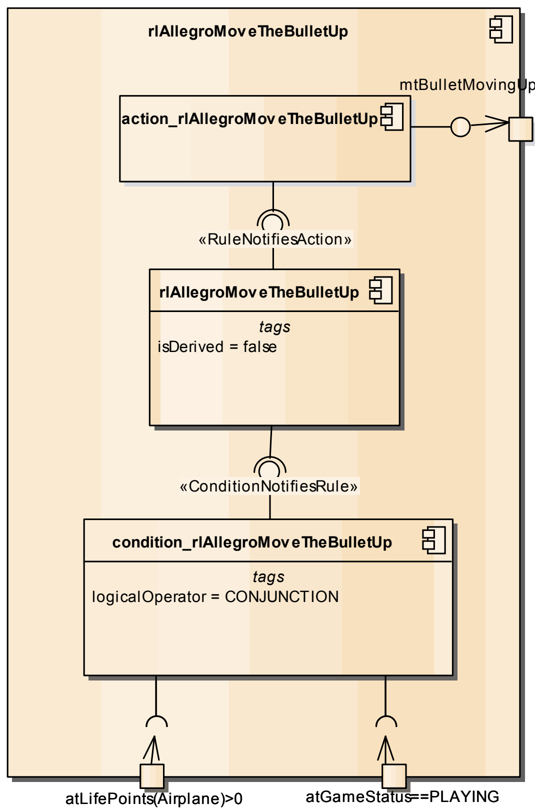


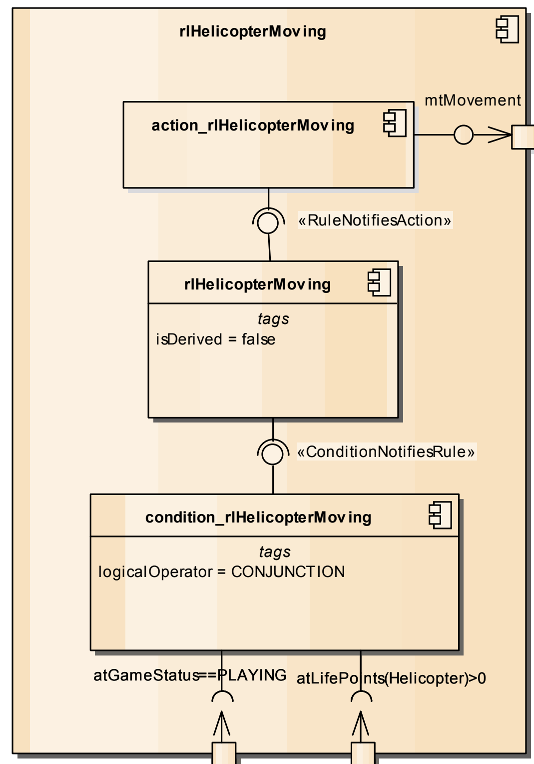


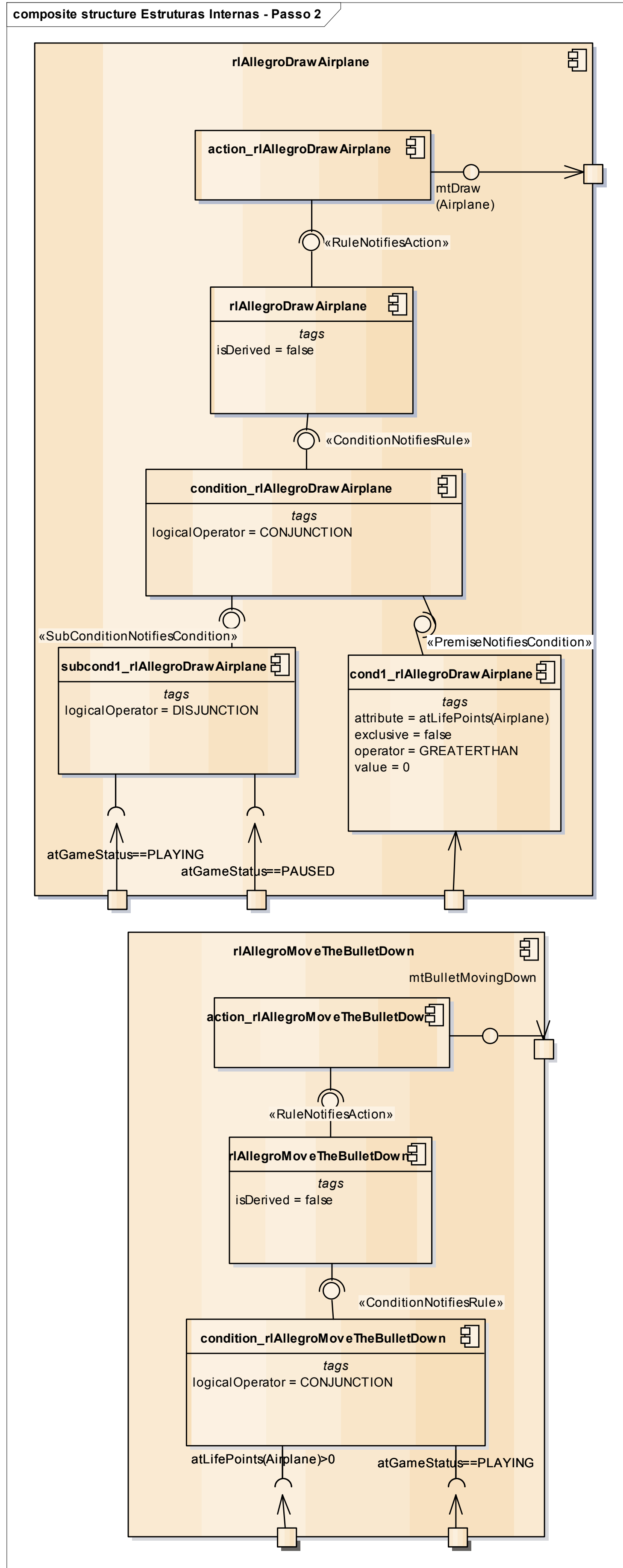
composite structure Estruturas Internas - Passo 2

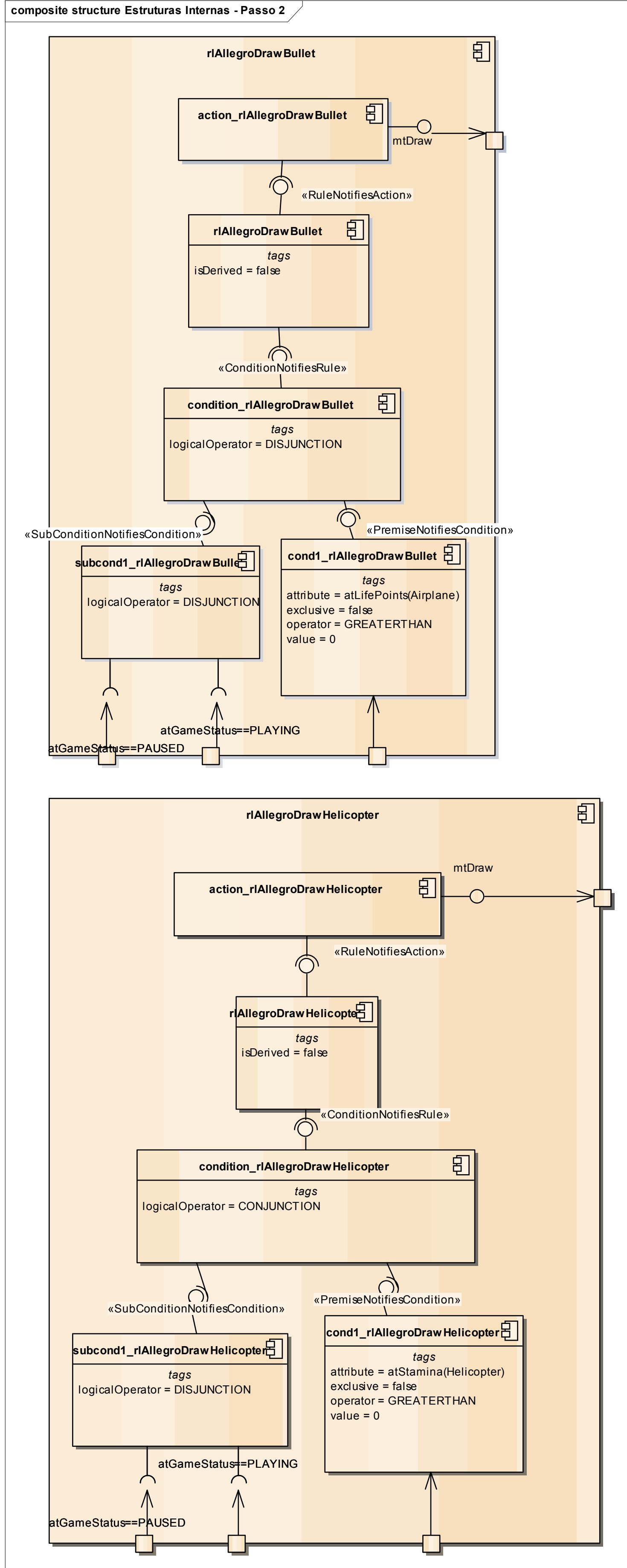


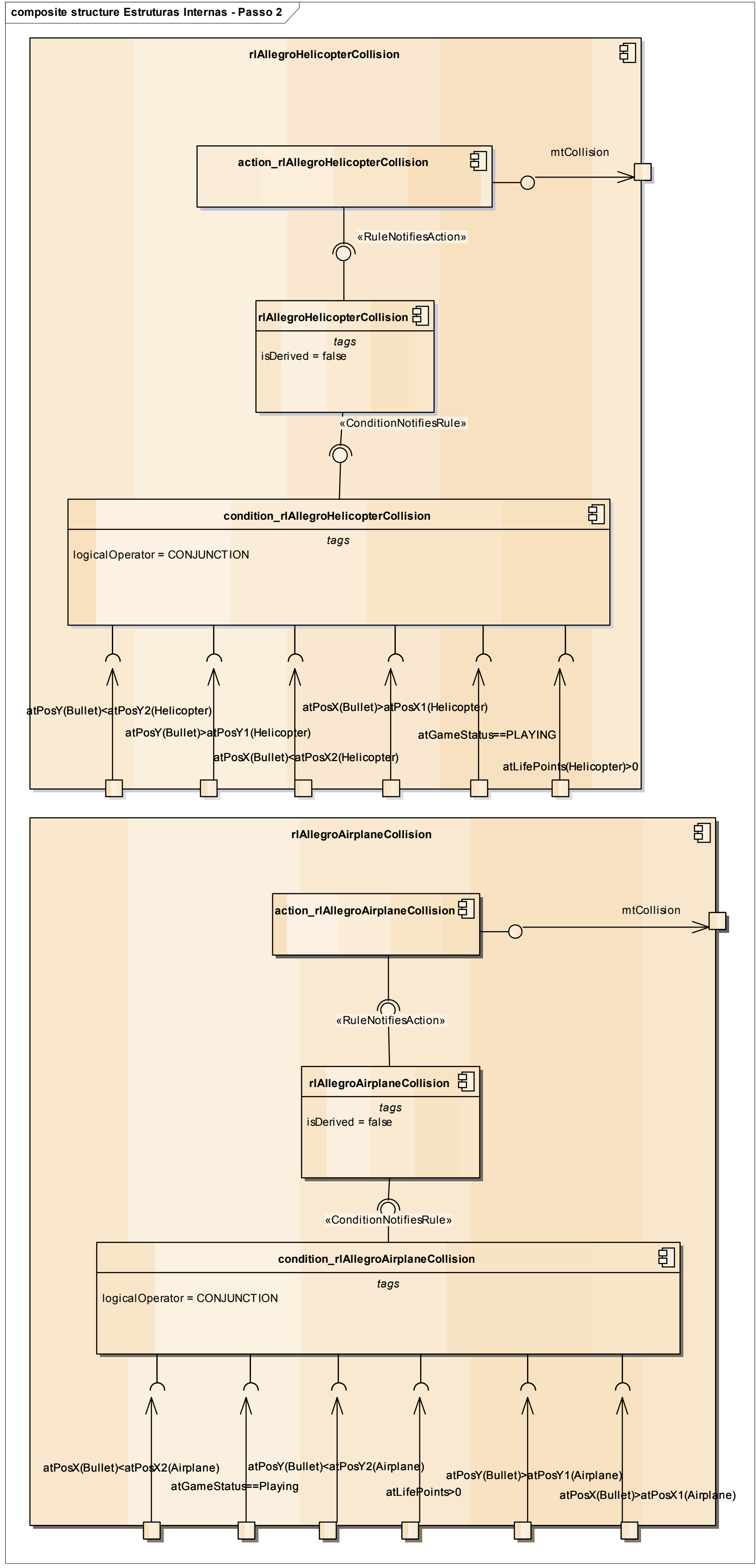


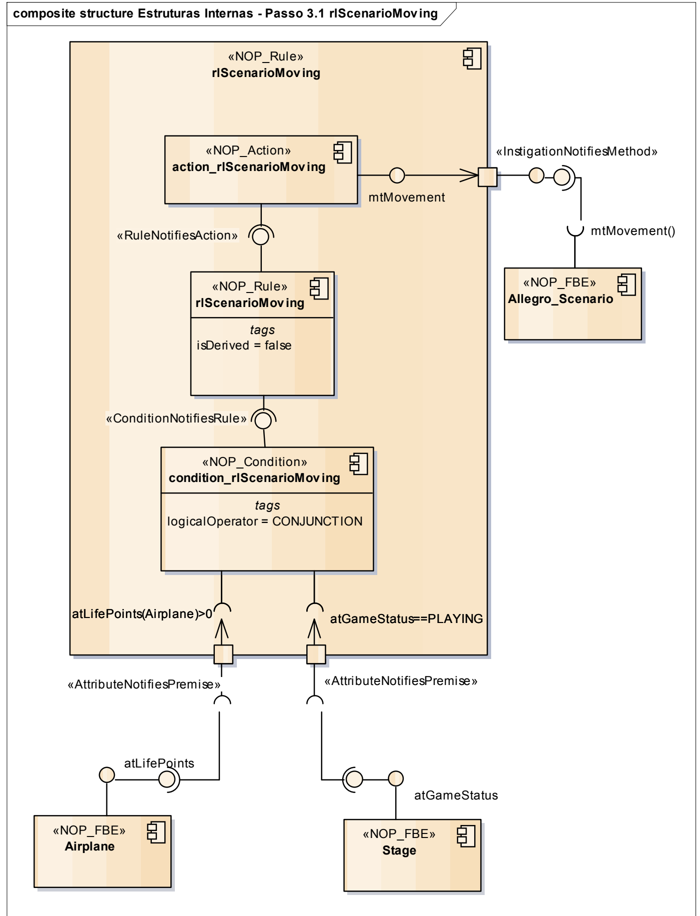


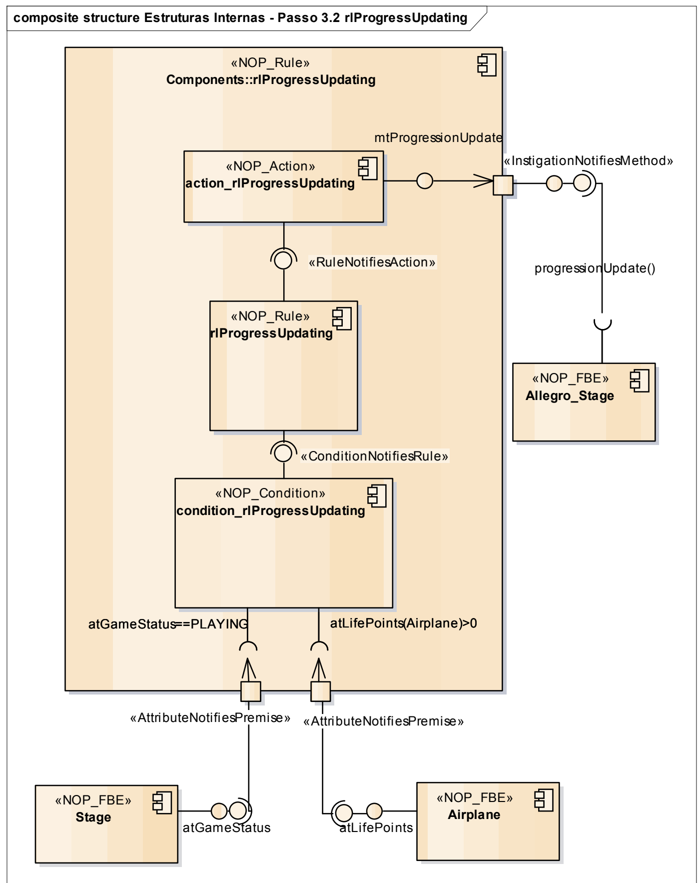


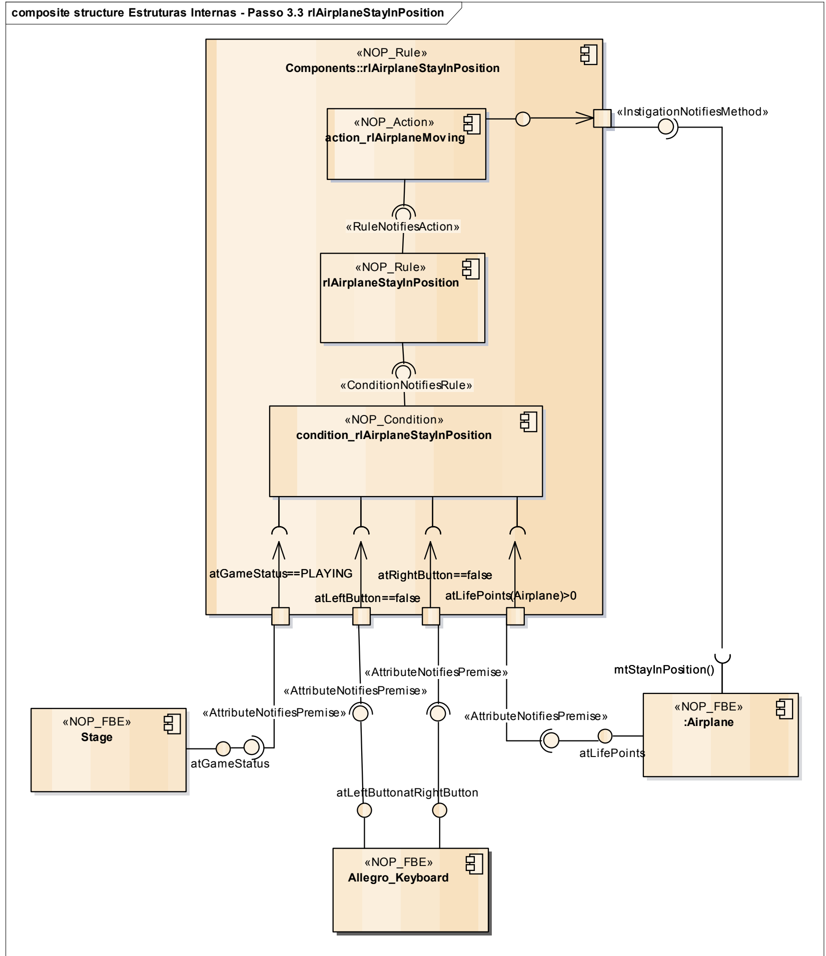

composite structure Estruturas Internas - Passo 2
rlAllegroDrawAirplane
action_rlAllegroDrawAirplane
mtDraw
(Airplane)
«RuleNotifiesAction»
rlAllegroDrawAirplane
tags
isDerived = false
«ConditionNotifiesRule»
condition_rlAllegroDrawAirplane
tags
logicalOperator = CONJUNCTION
«SubConditionNotifiesCondition»
«PremiseNotifiesCondition»
subcond1_rlAllegroDrawAirplane
tags
logicalOperator = DISJUNCTION
cond1_rlAllegroDrawAirplane
tags
attribute = atLifePoints(Airplane)
exclusive = false
operator = GREATERTHAN
value = 0
atGameStatus==PLAYING
atGameStatus==PAUSED
rlAllegroMoveTheBulletDown
mtBulletMovingDown
action_rlAllegroMoveTheBulletDown
«RuleNotifiesAction»
rlAllegroMoveTheBulletDown
tags
isDerived = false
«ConditionNotifiesRule»
condition_rlAllegroMoveTheBulletDown
tags
logicalOperator = CONJUNCTION
atLifePoints(Airplane)>0
atGameStatus==PLAYING

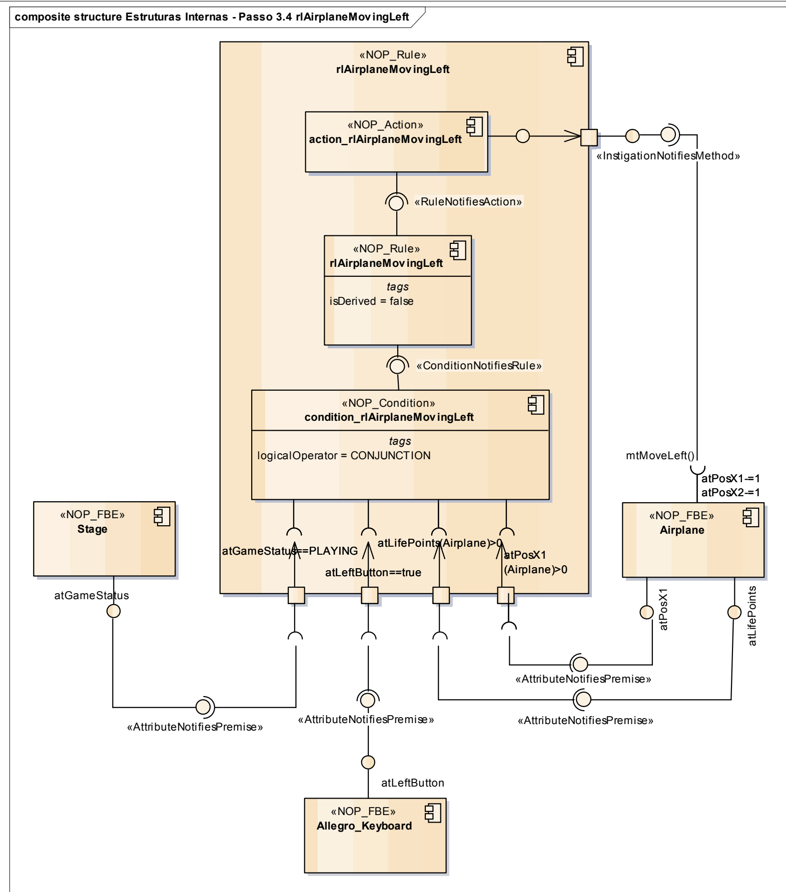

composite structure Estruturas Internas - Passo 2
rlAllegroDrawBullet
action_rlAllegroDrawBullet
mtDraw
«RuleNotifiesAction»
rlAllegroDrawBullet
tags
isDerived = false
«ConditionNotifiesRule»
condition_rlAllegroDrawBullet
tags
logicalOperator = DISJUNCTION
«SubConditionNotifiesCondition»
«PremiseNotifiesCondition»
subcond1_rlAllegroDrawBullet
tags
logicalOperator = DISJUNCTION
cond1_rlAllegroDrawBullet
tags
attribute = atLifePoints(Airplane)
exclusive = false
operator = GREATERTHAN
value = 0
atGameStatus==PLAYING
atGameStatus==PAUSED
rlAllegroDrawHelicopter
mtDraw
action_rlAllegroDrawHelicopter
«RuleNotifiesAction»
rlAllegroDrawHelicopter
tags
isDerived = false
«ConditionNotifiesRule»
condition_rlAllegroDrawHelicopter
tags
logicalOperator = CONJUNCTION
«PremiseNotifiesCondition»
«SubConditionNotifiesCondition»
cond1_rlAllegroDrawHelicopter
subcond1_rlAllegroDrawHelicopter
tags
logicalOperator = DISJUNCTION
tags
attribute = atStamina(Helicopter)
exclusive = false
operator = GREATERTHAN
value = 0
atGameStatus==PLAYING
atGameStatus==PAUSED

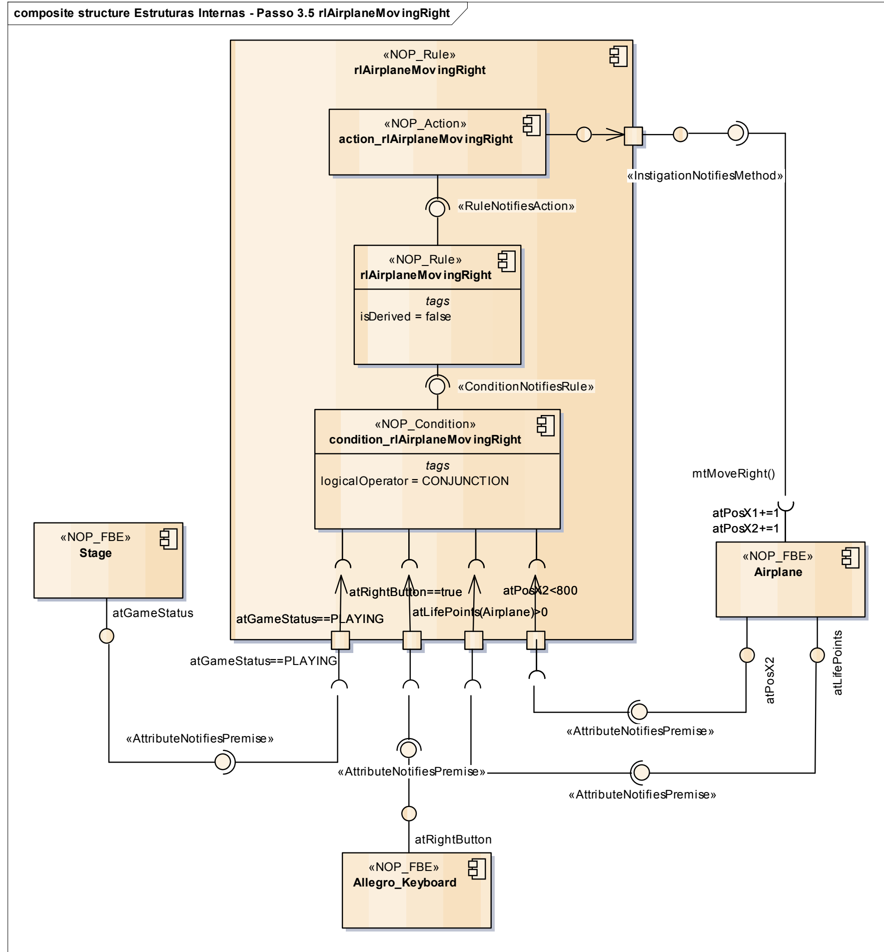

composite structure Estruturas Internas - Passo 2
rlAllegroHelicopterCollision
action_rlAllegroHelicopterCollision
mtCollision
«RuleNotifiesAction»
rlAllegroHelicopterCollision
tags
isDerived = false
«ConditionNotifiesRule»
condition_rlAllegroHelicopterCollision
tags
logicalOperator = CONJUNCTION
atPosY(Bullet)<atPosY2(Helicopter)
atPosX(Bullet)>atPosX1(Helicopter)
atPosY(Bullet)>atPosY1(Helicopter)
atGameStatus==PLAYING
atPosX(Bullet)<atPosX2(Helicopter)
atLifePoints(Helicopter)>0
rlAllegroAirplaneCollision
action_rlAllegroAirplaneCollision
mtCollision
«RuleNotifiesAction»
rlAllegroAirplaneCollision
tags
isDerived = false
«ConditionNotifiesRule»
condition_rlAllegroAirplaneCollision
tags
logicalOperator = CONJUNCTION
atPosX(Bullet)<atPosX2(Airplane)
atPosY(Bullet)<atPosY2(Airplane)
atGameStatus==Playing
atPosY(Bullet)>atPosY1(Airplane)
atLifePoints>0
atPosX(Bullet)>atPosX1(Airplane)

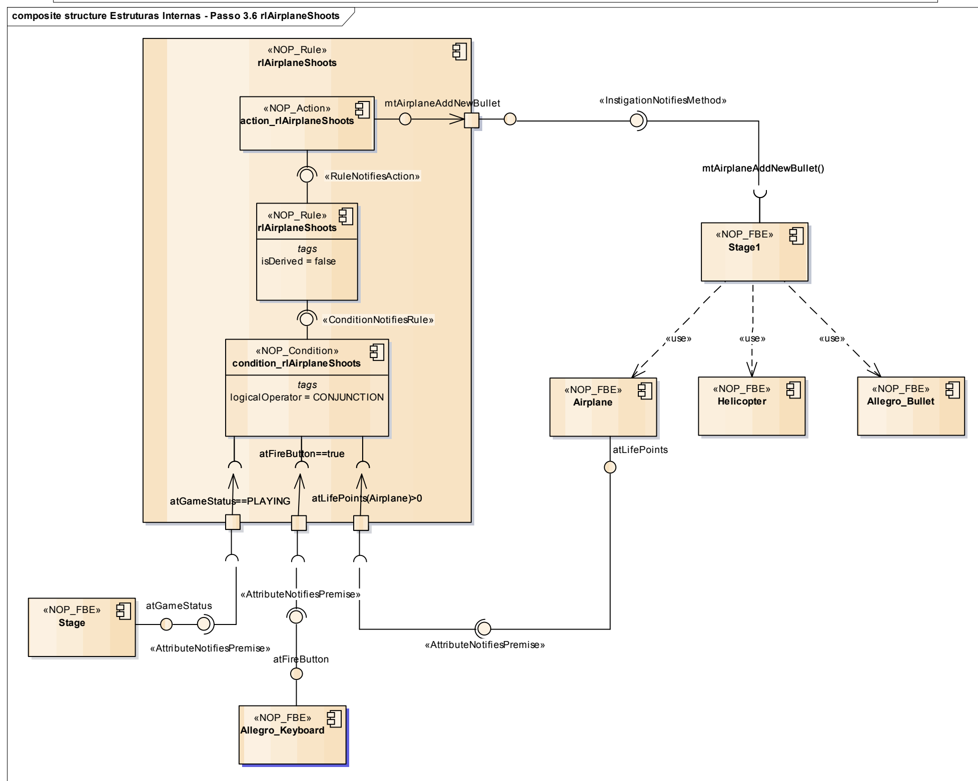

composite structure Estruturas Internas - Passo 3.1 rlScenarioMoving
«NOP_Rule»
rlScenarioMoving
«NOP_Action»
action_rlScenarioMoving
mtMovement
«InstigationNotifiesMethod»
mtMovement()
«NOP_FBE»
Allegro_Scenario
«RuleNotifiesAction»
«NOP_Rule»
rlScenarioMoving
tags
isDerived = false
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlScenarioMoving
tags
logicalOperator = CONJUNCTION
atLifePoints(Airplane)>0
atGameStatus==PLAYING
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atLifePoints
atGameStatus
«NOP_FBE»
Airplane
«NOP_FBE»
Stage

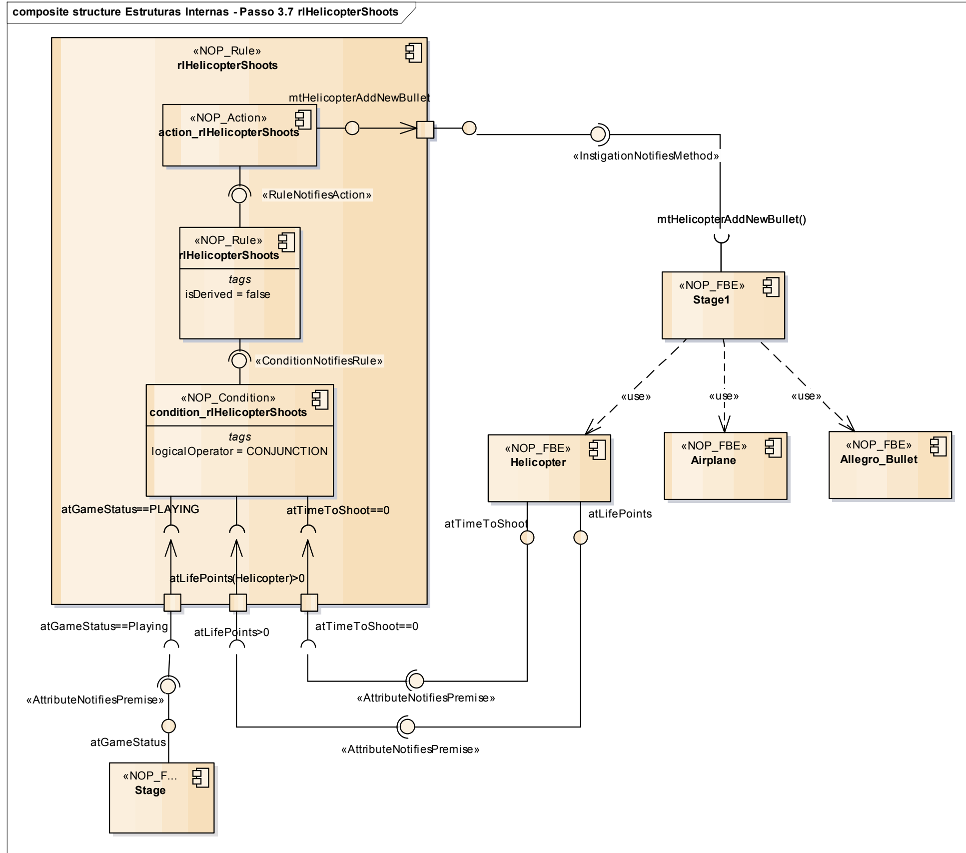

composite structure Estruturas Internas - Passo 3.2 rlProgressUpdating
«NOP_Rule»
Components::rlProgressUpdating
mtProgressionUpdate
«InstigationNotifiesMethod»
«NOP_Action»
action_rlProgressUpdating
«RuleNotifiesAction»
progressionUpdate()
«NOP_Rule»
rlProgressUpdating
«NOP_FBE»
Allegro_Stage
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlProgressUpdating
atGameStatus==PLAYING
atLifePoints(Airplane)>0
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«NOP_FBE»
Stage
atGameStatus
atLifePoints
«NOP_FBE»
Airplane

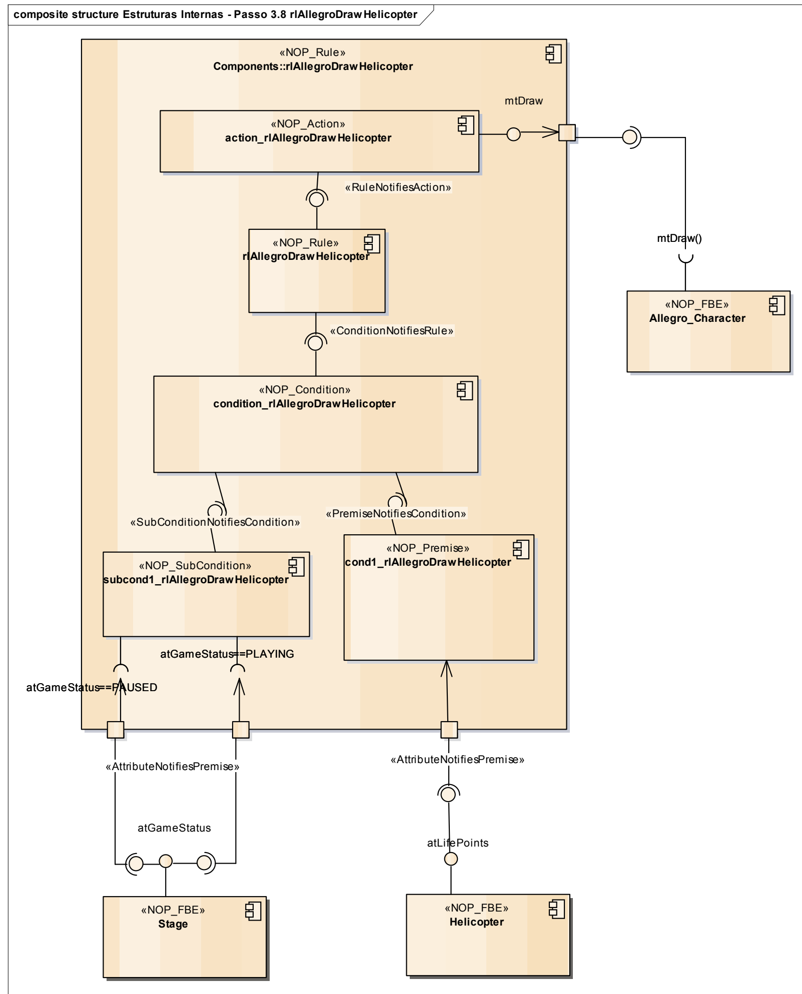

composite structure Estruturas Internas - Passo 3.3 rlAirplaneStayInPosition
«NOP_Rule»
Components::rlAirplaneStayInPosition
«NOP_Action»
action_rlAirplaneMoving
«InstigationNotifiesMethod»
«RuleNotifiesAction»
«NOP_Rule»
rlAirplaneStayInPosition
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAirplaneStayInPosition
atGameStatus==PLAYING
atRightButton==false
atLeftButton==false
atLifePoints(Airplane)>0
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
mtStayInPosition()
«NOP_FBE»
Stage
atGameStatus
«NOP_FBE»
:Airplane
atLifePoints
atLeftButtonatRightButton
«NOP_FBE»
Allegro_Keyboard

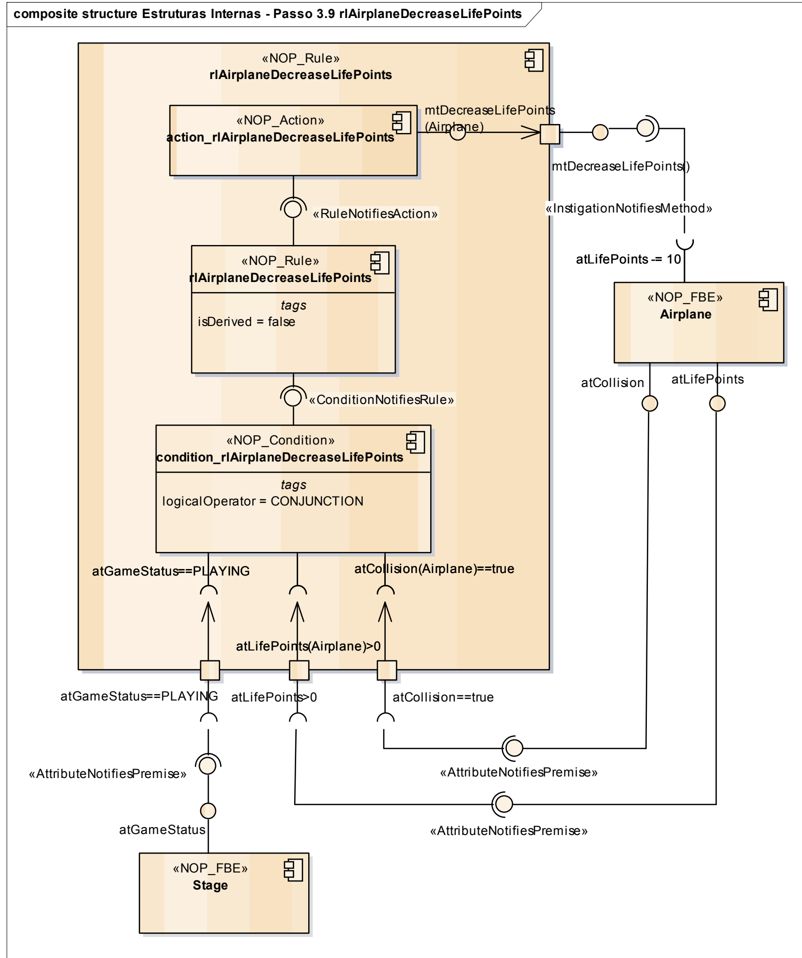

composite structure Estruturas Internas - Passo 3.4 rlAirplaneMovingLeft
«NOP_Rule»
rlAirplaneMovingLeft
«NOP_Action»
action_rlAirplaneMovingLeft
«InstigationNotifiesMethod»
«RuleNotifiesAction»
«NOP_Rule»
rlAirplaneMovingLeft
tags
isDerived = false
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAirplaneMovingLeft
tags
logicalOperator = CONJUNCTION
mtMoveLeft()
atPosX1-=1
atPosX2-=1
«NOP_FBE»
Stage
«NOP_FBE»
Airplane
atGameStatus==PLAYING
atLifePoints(Airplane)>0
atPosX1
(Airplane)>0
atLeftButton==true
atGameStatus
atPosX1
atLifePoints
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atLeftButton
«NOP_FBE»
Allegro_Keyboard

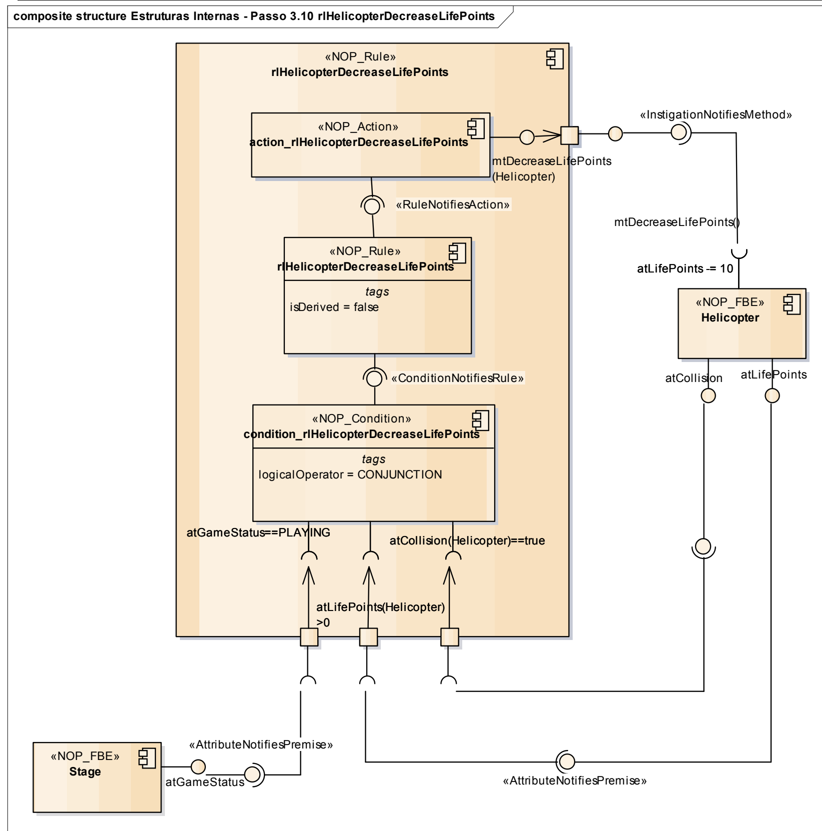

composite structure Estruturas Internas - Passo 3.5 rlAirplaneMovingRight
«NOP_Rule»
rlAirplaneMovingRight
«NOP_Action»
action_rlAirplaneMovingRight
«InstigationNotifiesMethod»
«RuleNotifiesAction»
«NOP_Rule»
rlAirplaneMovingRight
tags
isDerived = false
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAirplaneMovingRight
tags
logicalOperator = CONJUNCTION
mtMoveRight()
atPosX1+=1
atPosX2+=1
«NOP_FBE»
Stage
«NOP_FBE»
Airplane
atRightButton==true
atPosX2<800
atLifePoints(Airplane)>0
atGameStatus==PLAYING
atGameStatus
atGameStatus==PLAYING
atPosX2
atLifePoints
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atRightButton
«NOP_FBE»
Allegro_Keyboard

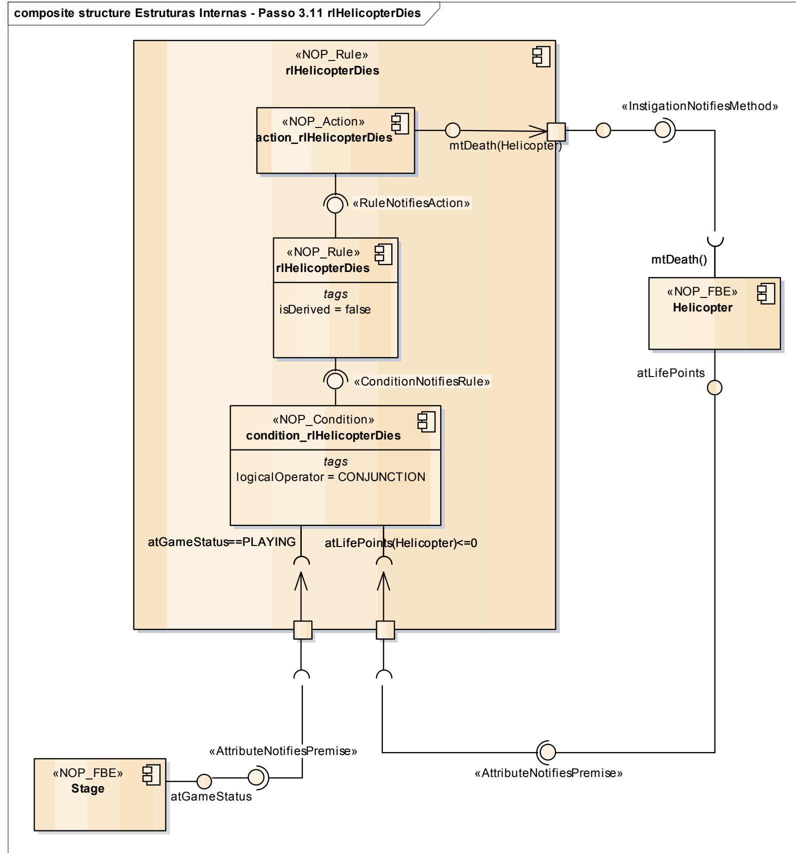

composite structure Estruturas Internas - Passo 3.6 rlAirplaneShoots
«NOP_Rule»
rlAirplaneShoots
«NOP_Action»
action_rlAirplaneShoots
mtAirplaneAddNewBullet
«InstigationNotifiesMethod»
«RuleNotifiesAction»
mtAirplaneAddNewBullet()
«NOP_Rule»
rlAirplaneShoots
tags
isDerived = false
«NOP_FBE»
Stage1
«ConditionNotifiesRule»
«use»
«use»
«use»
«NOP_Condition»
condition_rlAirplaneShoots
tags
logicalOperator = CONJUNCTION
«NOP_FBE»
Airplane
«NOP_FBE»
Helicopter
«NOP_FBE»
Allegro_Bullet
atLifePoints
atFireButton==true
atGameStatus==PLAYING
atLifePoints(Airplane)>0
«AttributeNotifiesPremise»
«NOP_FBE»
Stage
atGameStatus
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atFireButton
«NOP_FBE»
Allegro_Keyboard

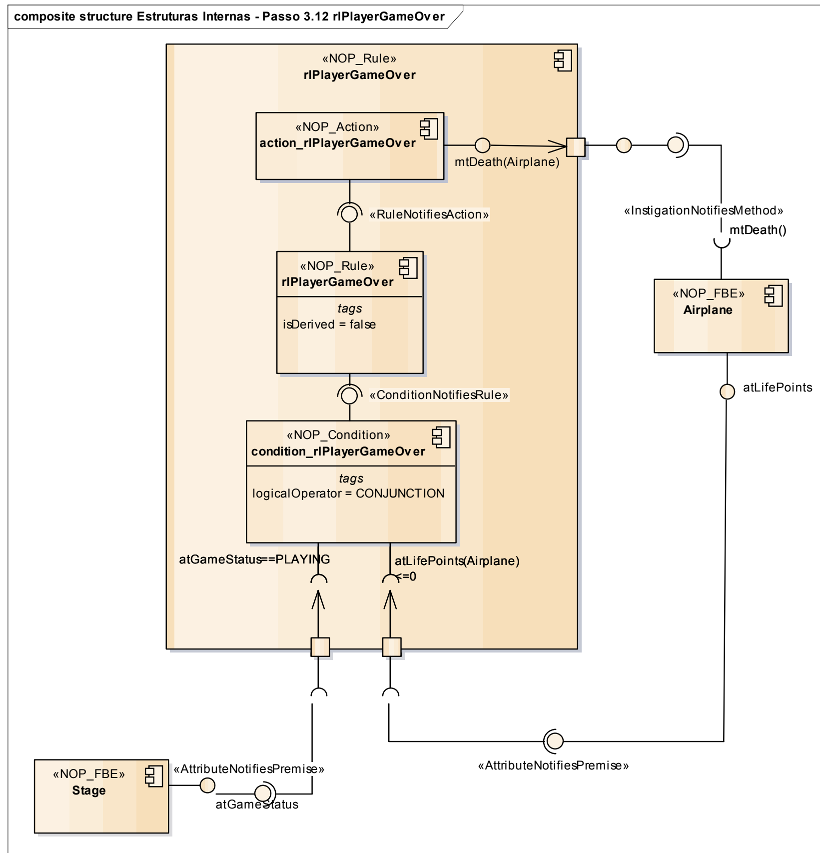

composite structure Estruturas Internas - Passo 3.7 rlHelicopterShoots
«NOP_Rule»
rlHelicopterShoots
mtHelicopterAddNewBullet
«NOP_Action»
action_rlHelicopterShoots
«InstigationNotifiesMethod»
«RuleNotifiesAction»
mtHelicopterAddNewBullet()
«NOP_Rule»
rlHelicopterShoots
tags
isDerived = false
«NOP_FBE»
Stage1
«ConditionNotifiesRule»
«use»
«use»
«use»
«NOP_Condition»
condition_rlHelicopterShoots
tags
logicalOperator = CONJUNCTION
«NOP_FBE»
Helicopter
«NOP_FBE»
Airplane
«NOP_FBE»
Allegro_Bullet
atGameStatus==PLAYING
atTimeToShoot==0
atTimeToShoot
atLifePoints
atLifePoints(Helicopter)>0
atGameStatus==Playing
atLifePoints>0
atTimeToShoot==0
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
«NOP_F...
Stage

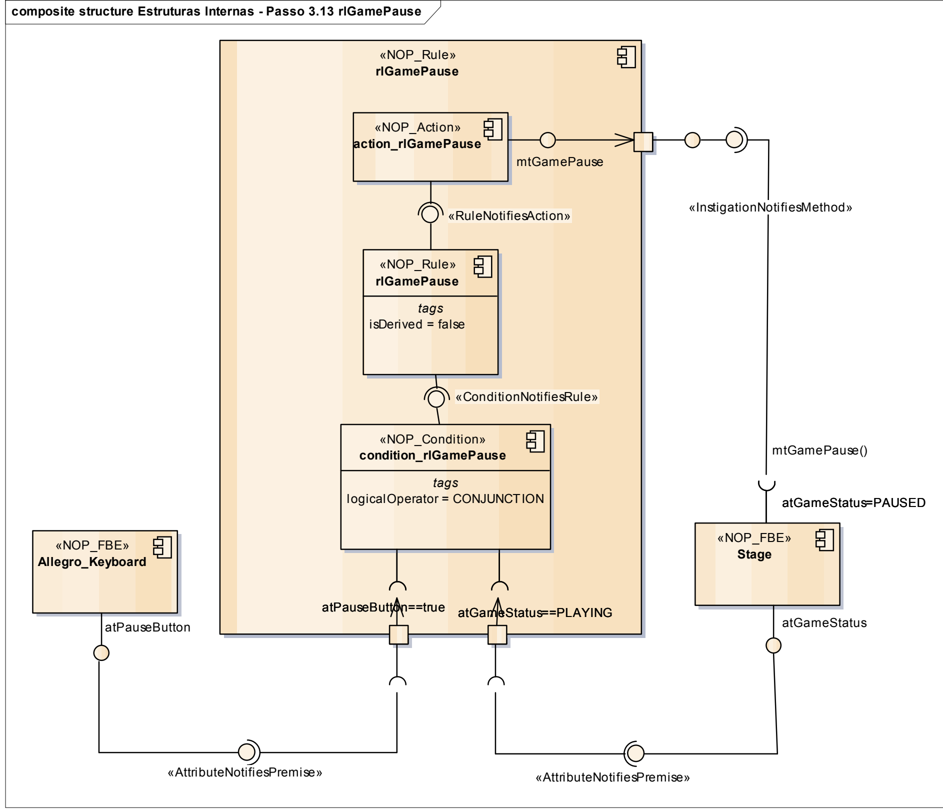

composite structure Estruturas Internas - Passo 3.8 rlAllegroDrawHelicopter
«NOP_Rule»
Components::rlAllegroDrawHelicopter
mtDraw
«NOP_Action»
action_rlAllegroDrawHelicopter
«RuleNotifiesAction»
mtDraw()
«NOP_Rule»
rlAllegroDrawHelicopter
«NOP_FBE»
Allegro_Character
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroDrawHelicopter
«PremiseNotifiesCondition»
«SubConditionNotifiesCondition»
«NOP_Premise»
cond1_rlAllegroDrawHelicopter
«NOP_SubCondition»
subcond1_rlAllegroDrawHelicopter
atGameStatus==PLAYING
atGameStatus==PAUSED
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
atLifePoints
«NOP_FBE»
Stage
«NOP_FBE»
Helicopter

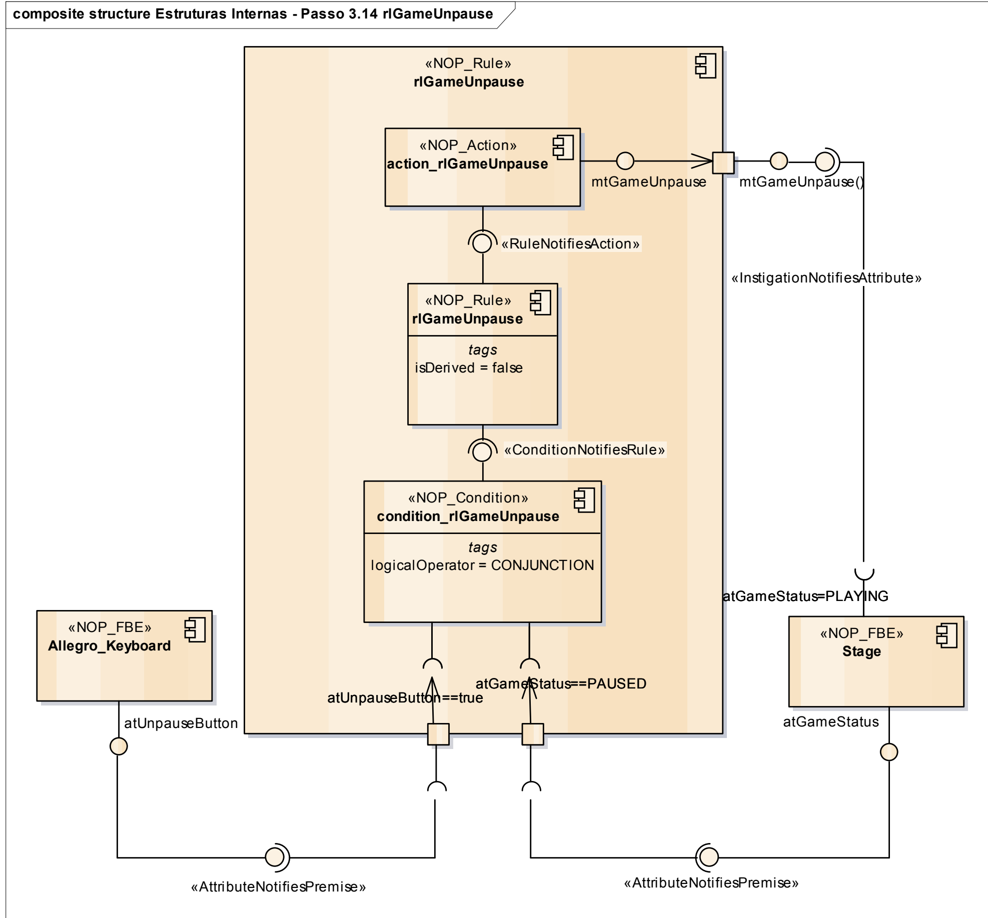

composite structure Estruturas Internas - Passo 3.9 rlAirplaneDecreaseLifePoints
«NOP_Rule»
rlAirplaneDecreaseLifePoints
«NOP_Action»
action_rlAirplaneDecreaseLifePoints
mtDecreaseLifePoints
(Airplane)
mtDecreaseLifePoints()
«InstigationNotifiesMethod»
«RuleNotifiesAction»
atLifePoints -= 10
«NOP_Rule»
rlAirplaneDecreaseLifePoints
tags
isDerived = false
«NOP_FBE»
Airplane
atCollision
atLifePoints
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAirplaneDecreaseLifePoints
tags
logicalOperator = CONJUNCTION
atGameStatus==PLAYING
atCollision(Airplane)==true
atLifePoints(Airplane)>0
atGameStatus==PLAYING
atLifePoints>0
atCollision==true
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
«NOP_FBE»
Stage

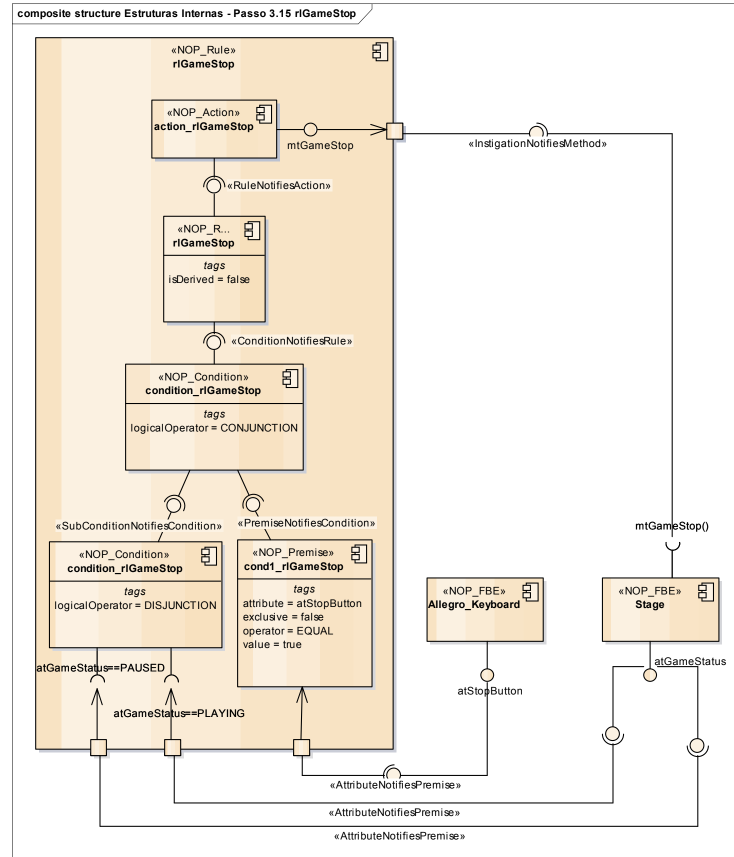

composite structure Estruturas Internas - Passo 3.10 rlHelicopterDecreaseLifePoints
«NOP_Rule»
rlHelicopterDecreaseLifePoints
«NOP_Action»
action_rlHelicopterDecreaseLifePoints
«InstigationNotifiesMethod»
mtDecreaseLifePoints
(Helicopter)
«RuleNotifiesAction»
mtDecreaseLifePoints()
«NOP_Rule»
rlHelicopterDecreaseLifePoints
tags
isDerived = false
atLifePoints -= 10
«NOP_FBE»
Helicopter
«ConditionNotifiesRule»
atCollision
atLifePoints
«NOP_Condition»
condition_rlHelicopterDecreaseLifePoints
tags
logicalOperator = CONJUNCTION
atGameStatus==PLAYING
atCollision(Helicopter)==true
atLifePoints(Helicopter)
>0
«NOP_FBE»
Stage
«AttributeNotifiesPremise»
atGameStatus
«AttributeNotifiesPremise»

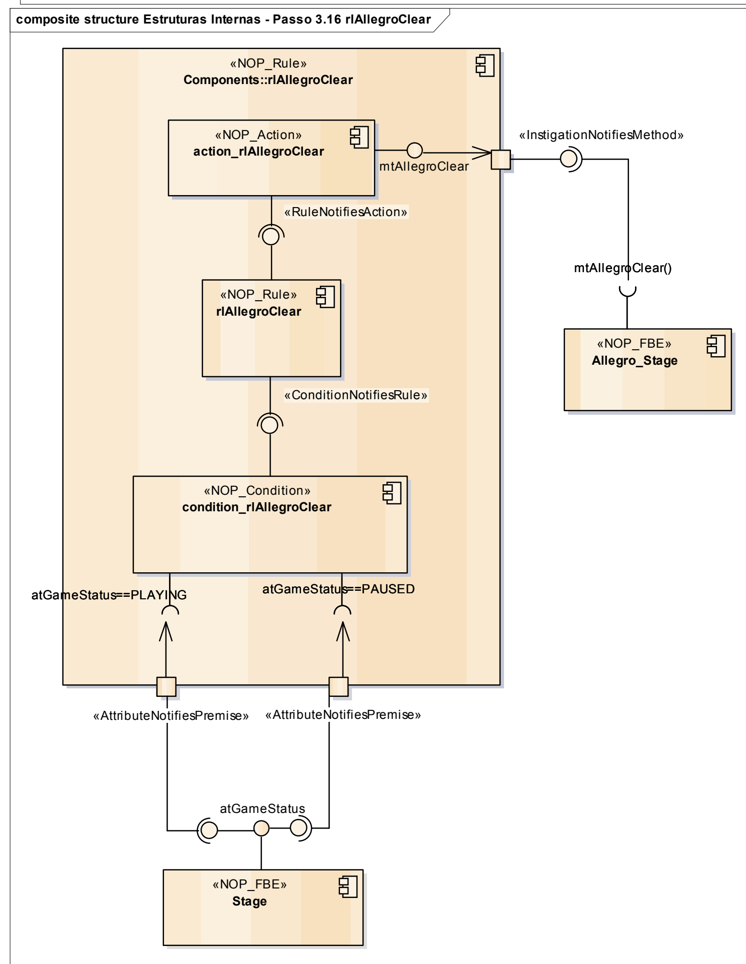

composite structure Estruturas Internas - Passo 3.11 rlHelicopterDies
«NOP_Rule»
rlHelicopterDies
«NOP_Action»
action_rlHelicopterDies
mtDeath(Helicopter)
«InstigationNotifiesMethod»
«RuleNotifiesAction»
«NOP_Rule»
rlHelicopterDies
tags
isDerived = false
mtDeath()
«NOP_FBE»
Helicopter
atLifePoints
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlHelicopterDies
tags
logicalOperator = CONJUNCTION
atGameStatus==PLAYING
atLifePoints(Helicopter)<=0
«AttributeNotifiesPremise»
«NOP_FBE»
Stage
atGameStatus
«AttributeNotifiesPremise»

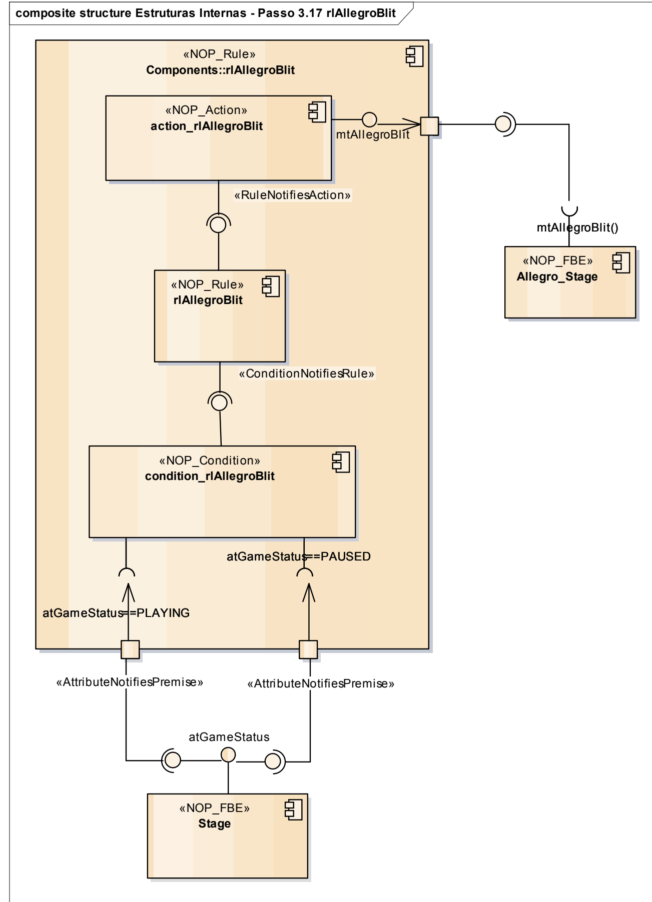

composite structure Estruturas Internas - Passo 3.12 rlPlayerGameOver
«NOP_Rule»
rlPlayerGameOver
«NOP_Action»
action_rlPlayerGameOver
mtDeath(Airplane)
«RuleNotifiesAction»
«InstigationNotifiesMethod»
mtDeath()
«NOP_Rule»
rlPlayerGameOver
tags
isDerived = false
«NOP_FBE»
Airplane
atLifePoints
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlPlayerGameOver
tags
logicalOperator = CONJUNCTION
atGameStatus==PLAYING
atLifePoints(Airplane)
<=0
«NOP_FBE»
Stage
«AttributeNotifiesPremise»
atGameStatus
«AttributeNotifiesPremise»

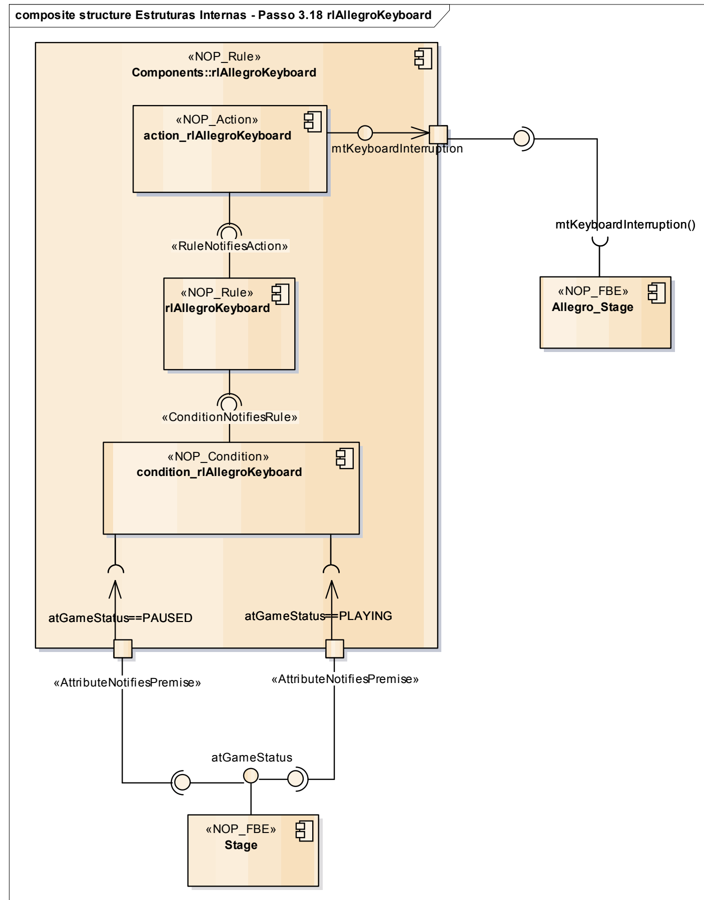

composite structure Estruturas Internas - Passo 3.13 rlGamePause
«NOP_Rule»
rlGamePause
«NOP_Action»
action_rlGamePause
mtGamePause
«RuleNotifiesAction»
«NOP_Rule»
rlGamePause
tags
isDerived = false
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlGamePause
tags
logicalOperator = CONJUNCTION
atPauseButton==true
atGameStatus==PLAYING
«InstigationNotifiesMethod»
mtGamePause()
atGameStatus=PAUSED
«NOP_FBE»
Stage
atGameStatus
«NOP_FBE»
Allegro_Keyboard
atPauseButton
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»

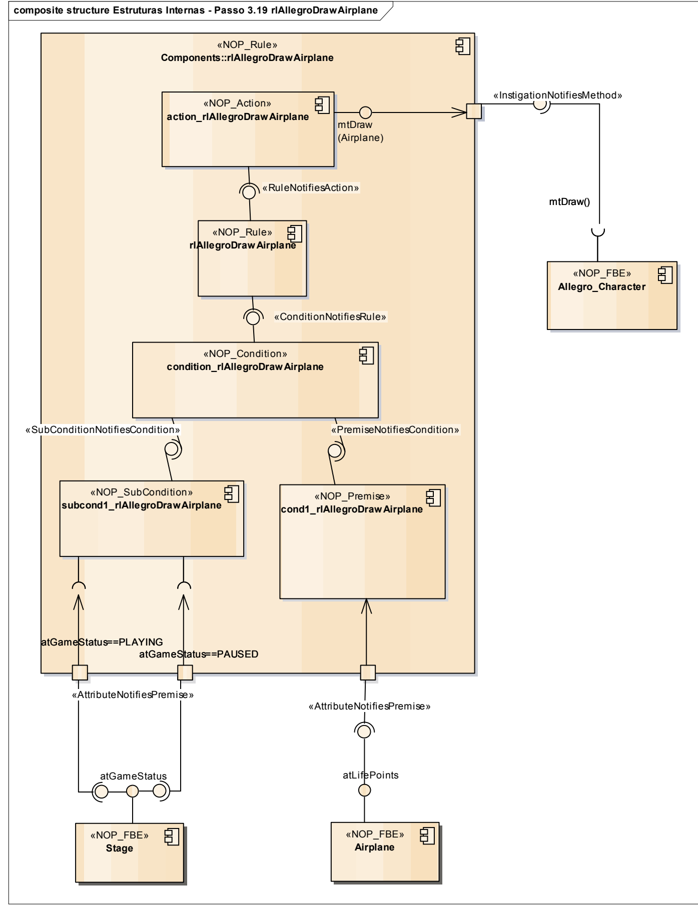

composite structure Estruturas Internas - Passo 3.14 rlGameUnpause
«NOP_Rule»
rlGameUnpause
«NOP_Action»
action_rlGameUnpause
mtGameUnpause
mtGameUnpause()
«RuleNotifiesAction»
«InstigationNotifiesAttribute»
«NOP_Rule»
rlGameUnpause
tags
isDerived = false
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlGameUnpause
tags
logicalOperator = CONJUNCTION
atGameStatus=PLAYING
«NOP_FBE»
Allegro_Keyboard
«NOP_FBE»
Stage
atGameStatus==PAUSED
atUnpauseButton==true
atUnpauseButton
atGameStatus
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»

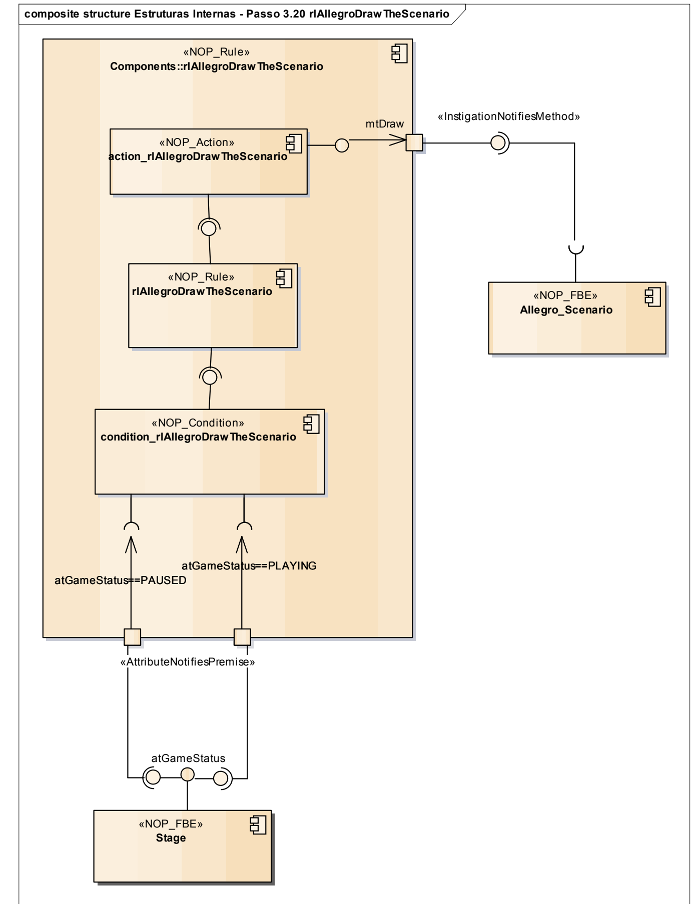

composite structure Estruturas Internas - Passo 3.15 rlGameStop
«NOP_Rule»
rlGameStop
«NOP_Action»
action_rlGameStop
mtGameStop
«InstigationNotifiesMethod»
«RuleNotifiesAction»
«NOP_R...
rlGameStop
tags
isDerived = false
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlGameStop
tags
logicalOperator = CONJUNCTION
«SubConditionNotifiesCondition»
«PremiseNotifiesCondition»
mtGameStop()
«NOP_Condition»
condition_rlGameStop
tags
logicalOperator = DISJUNCTION
«NOP_Premise»
cond1_rlGameStop
tags
attribute = atStopButton
exclusive = false
operator = EQUAL
value = true
«NOP_FBE»
Allegro_Keyboard
«NOP_FBE»
Stage
atGameStatus
atGameStatus==PAUSED
atStopButton
atGameStatus==PLAYING
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»

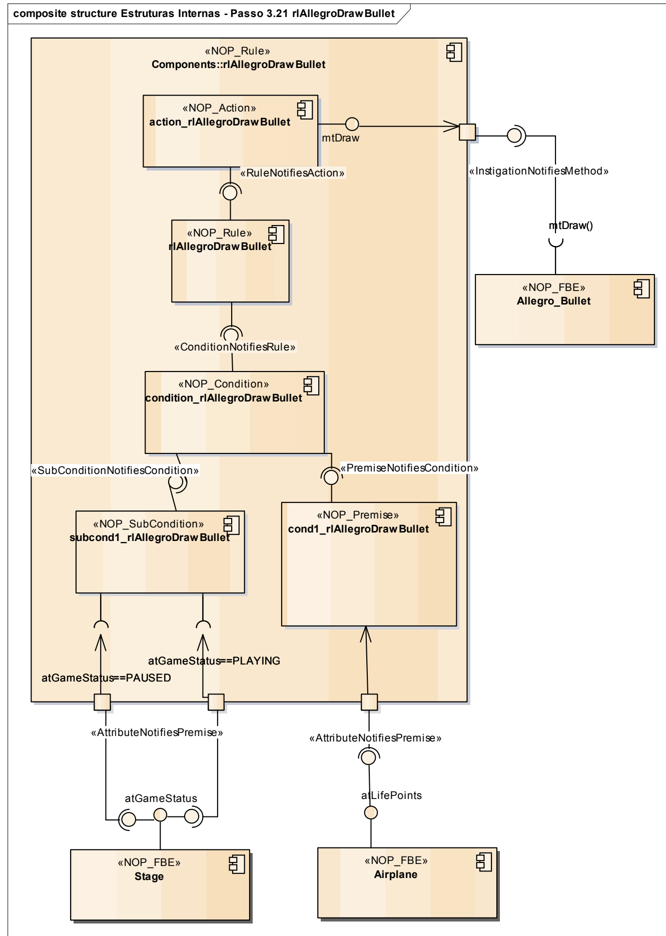

composite structure Estruturas Internas - Passo 3.16 rlAllegroClear
«NOP_Rule»
Components::rlAllegroClear
«NOP_Action»
action_rlAllegroClear
«InstigationNotifiesMethod»
mtAllegroClear
«RuleNotifiesAction»
mtAllegroClear()
«NOP_Rule»
rlAllegroClear
«NOP_FBE»
Allegro_Stage
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroClear
atGameStatus==PLAYING
atGameStatus==PAUSED
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
«NOP_FBE»
Stage


composite structure Estruturas Internas - Passo 3.17 rlAllegroBlit
«NOP_Rule»
Components::rlAllegroBlit
«NOP_Action»
action_rlAllegroBlit
mtAllegroBlit
«RuleNotifiesAction»
mtAllegroBlit()
«NOP_FBE»
Allegro_Stage
«NOP_Rule»
rlAllegroBlit
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroBlit
atGameStatus==PAUSED
atGameStatus==PLAYING
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
«NOP_FBE»
Stage

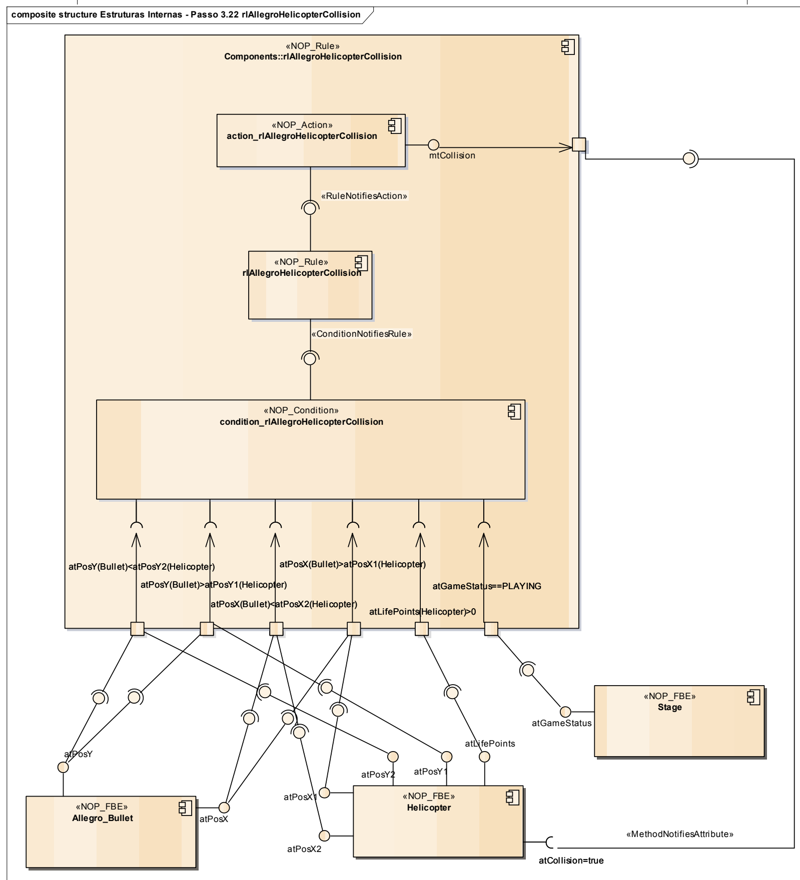

composite structure Estruturas Internas - Passo 3.18 rlAllegroKeyboard
«NOP_Rule»
Components::rlAllegroKeyboard
«NOP_Action»
action_rlAllegroKeyboard
mtKeyboardInterruption
mtKeyboardInterruption()
«RuleNotifiesAction»
«NOP_FBE»
Allegro_Stage
«NOP_Rule»
rlAllegroKeyboard
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroKeyboard
atGameStatus==PAUSED
atGameStatus==PLAYING
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
«NOP_FBE»
Stage


composite structure Estruturas Internas - Passo 3.19 rlAllegroDrawAirplane
«NOP_Rule»
Components::rlAllegroDrawAirplane
«NOP_Action»
action_rlAllegroDrawAirplane
mtDraw
(Airplane)
«InstigationNotifiesMethod»
mtDraw()
«RuleNotifiesAction»
«NOP_Rule»
rlAllegroDrawAirplane
«NOP_FBE»
Allegro_Character
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroDrawAirplane
«SubConditionNotifiesCondition»
«PremiseNotifiesCondition»
«NOP_SubCondition»
subcond1_rlAllegroDrawAirplane
«NOP_Premise»
cond1_rlAllegroDrawAirplane
atGameStatus==PLAYING
atGameStatus==PAUSED
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
atLifePoints
«NOP_FBE»
Stage
«NOP_FBE»
Airplane

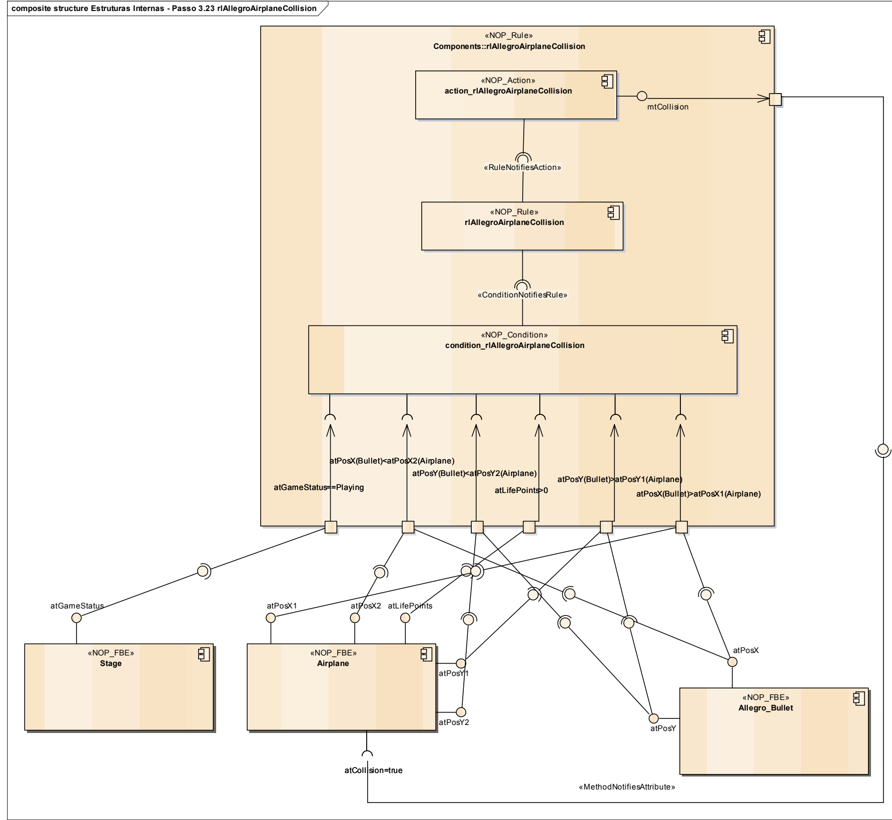

composite structure Estruturas Internas - Passo 3.20 rlAllegroDrawTheScenario
«NOP_Rule»
Components::rlAllegroDrawTheScenario
mtDraw
«InstigationNotifiesMethod»
«NOP_Action»
action_rlAllegroDrawTheScenario
«NOP_Rule»
rlAllegroDrawTheScenario
«NOP_FBE»
Allegro_Scenario
«NOP_Condition»
condition_rlAllegroDrawTheScenario
atGameStatus==PLAYING
atGameStatus==PAUSED
«AttributeNotifiesPremise»
atGameStatus
«NOP_FBE»
Stage

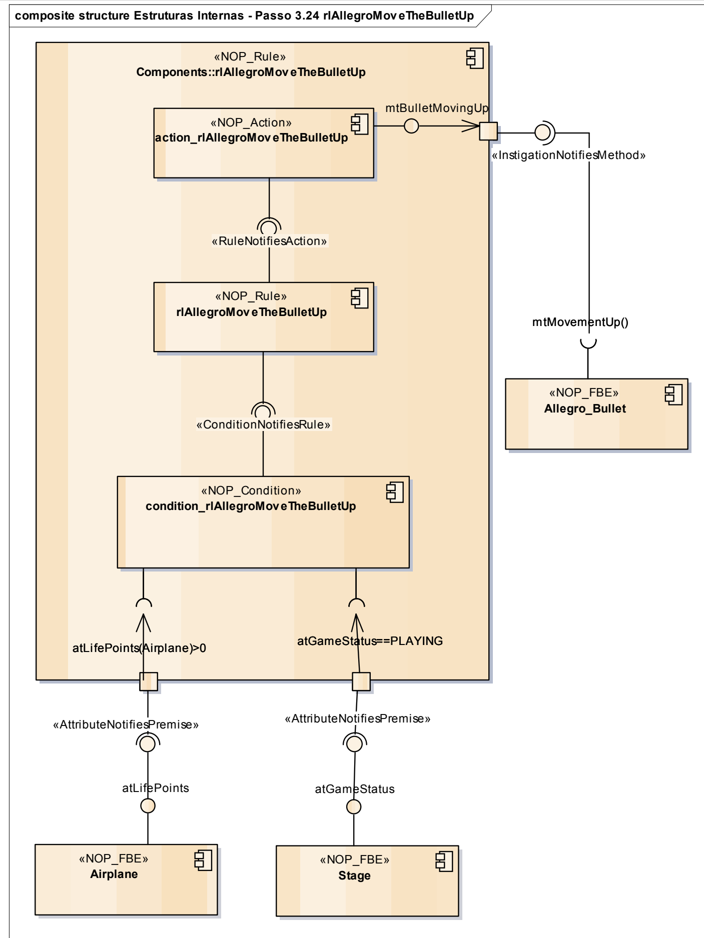


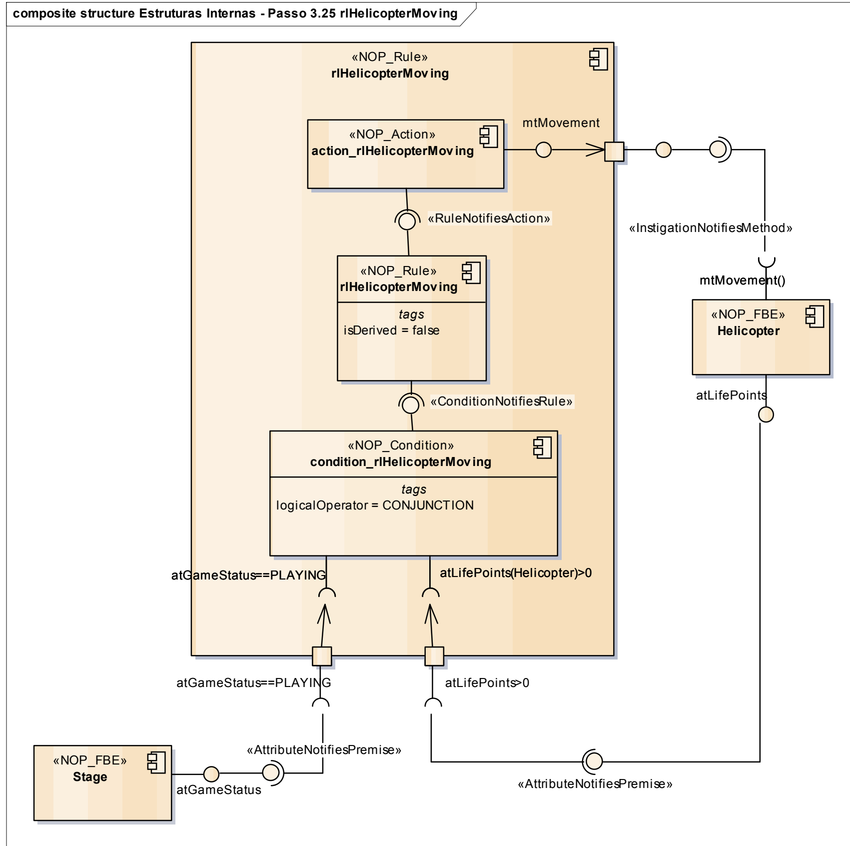
composite structure Estruturas Internas - Passo 3.21 rlAllegroDrawBullet
«NOP_Rule»
Components::rlAllegroDrawBullet
«NOP_Action»
action_rlAllegroDrawBullet
mtDraw
«InstigationNotifiesMethod»
mtDraw()
«NOP_FBE»
Allegro_Bullet
«RuleNotifiesAction»
«NOP_Rule»
rlAllegroDrawBullet
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroDrawBullet
«SubConditionNotifiesCondition»
«PremiseNotifiesCondition»
«NOP_SubCondition»
subcond1_rlAllegroDrawBullet
«NOP_Premise»
cond1_rlAllegroDrawBullet
atGameStatus==PLAYING
atGameStatus==PAUSED
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atGameStatus
atLifePoints
«NOP_FBE»
Stage
«NOP_FBE»
Airplane

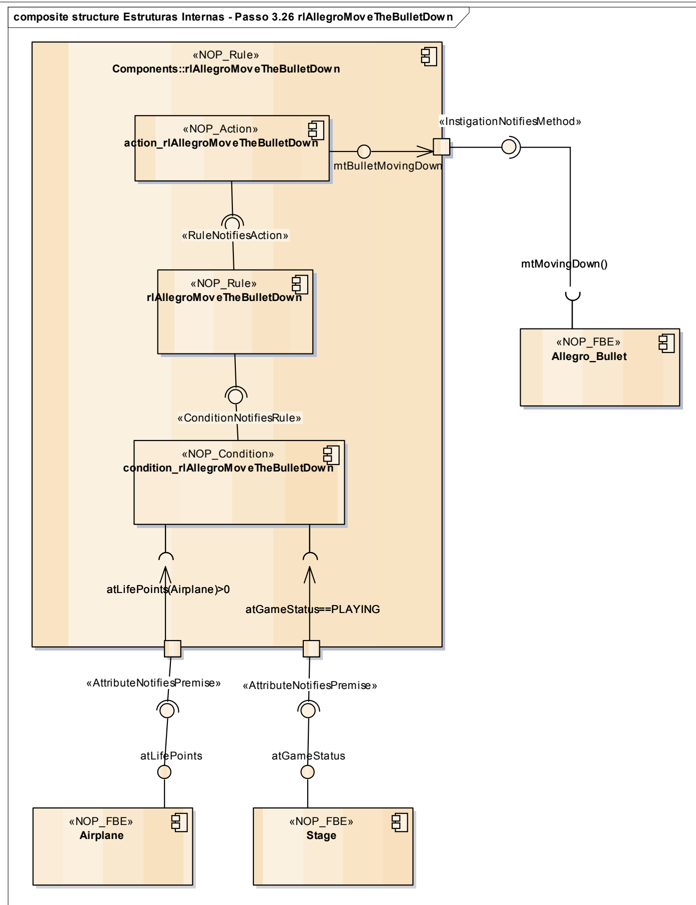


composite structure Estruturas Internas - Passo 3.22 rlAllegroHelicopterCollision
«NOP_Rule»
Components::rlAllegroHelicopterCollision
«NOP_Action»
action_rlAllegroHelicopterCollision
mtCollision
«RuleNotifiesAction»
«NOP_Rule»
rlAllegroHelicopterCollision
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroHelicopterCollision
atPosY(Bullet)<atPosY2(Helicopter)
atPosY(Bullet)>atPosY1(Helicopter)
atPosX(Bullet)<atPosX2(Helicopter)
atPosX(Bullet)>atPosX1(Helicopter)
atLifePoints(Helicopter)>0
atGameStatus==PLAYING
atGameStatus
«NOP_FBE»
Stage
atLifePoints
atPosY
atPosY2
atPosY1
atPosX1
«NOP_FBE»
Allegro_Bullet
atPosX
«NOP_FBE»
Helicopter
atPosX2
«MethodNotifiesAttribute»
atCollision=true


composite structure Estruturas Internas - Passo 3.23 rlAllegroAirplaneCollision
«NOP_Rule»
Components::rlAllegroAirplaneCollision
«NOP_Action»
action_rlAllegroAirplaneCollision
mtCollision
«RuleNotifiesAction»
«NOP_Rule»
rlAllegroAirplaneCollision
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroAirplaneCollision
atPosX(Bullet)<atPosX2(Airplane)
atPosY(Bullet)<atPosY2(Airplane)
atPosY(Bullet)>atPosY1(Airplane)
atGameStatus==Playing
atLifePoints>0
atPosX(Bullet)>atPosX1(Airplane)
atGameStatus
atPosX1
atPosX2
atLifePoints
«NOP_FBE»
Stage
«NOP_FBE»
Airplane
atPosY1
atPosY2
atPosX
«NOP_FBE»
Allegro_Bullet
atPosY
atCollision=true
«MethodNotifiesAttribute»


composite structure Estruturas Internas - Passo 3.24 rlAllegroMoveTheBulletUp
«NOP_Rule»
Components::rlAllegroMoveTheBulletUp
«NOP_Action»
action_rlAllegroMoveTheBulletUp
mtBulletMovingUp
«InstigationNotifiesMethod»
«RuleNotifiesAction»
«NOP_Rule»
rlAllegroMoveTheBulletUp
mtMovementUp()
«NOP_FBE»
Allegro_Bullet
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroMoveTheBulletUp
atLifePoints(Airplane)>0
atGameStatus==PLAYING
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atLifePoints
atGameStatus
«NOP_FBE»
Airplane
«NOP_FBE»
Stage


composite structure Estruturas Internas - Passo 3.25 rlHelicopterMoving
«NOP_Rule»
rlHelicopterMoving
«NOP_Action»
action_rlHelicopterMoving
mtMovement
«RuleNotifiesAction»
«InstigationNotifiesMethod»
«NOP_Rule»
rlHelicopterMoving
tags
isDerived = false
mtMovement()
«NOP_FBE»
Helicopter
atLifePoints
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlHelicopterMoving
tags
logicalOperator = CONJUNCTION
atGameStatus==PLAYING
atLifePoints(Helicopter)>0
atGameStatus==PLAYING
atLifePoints>0
«AttributeNotifiesPremise»
«NOP_FBE»
Stage
atGameStatus
«AttributeNotifiesPremise»


composite structure Estruturas Internas - Passo 3.26 rlAllegroMoveTheBulletDown
«NOP_Rule»
Components::rlAllegroMoveTheBulletDown
«NOP_Action»
action_rlAllegroMoveTheBulletDown
«InstigationNotifiesMethod»
mtBulletMovingDown
«RuleNotifiesAction»
mtMovingDown()
«NOP_Rule»
rlAllegroMoveTheBulletDown
«NOP_FBE»
Allegro_Bullet
«ConditionNotifiesRule»
«NOP_Condition»
condition_rlAllegroMoveTheBulletDown
atLifePoints(Airplane)>0
atGameStatus==PLAYING
«AttributeNotifiesPremise»
«AttributeNotifiesPremise»
atLifePoints
atGameStatus
«NOP_FBE»
Airplane
«NOP_FBE»
Stage

APÊNDICE D – Diagramas de sequência

sd UC-Controlar apresentação

«NOP_Condition» cond_GameStatusPlaying

«NOP_Condition» cond_GameStatusPausedORPlaying

«NOP_Rule» rlAllegroDrawHelicopter

«NOP_Rule» rlAllegroDrawTheScenario

«NOP_Rule» rlAllegroBlit

«NOP_Rule» rlAllegroClear

«NOP_Rule» rlAllegroDrawBullet

«NOP_Rule» rlProgressUpdating

«NOP_Rule» rlAllegroDrawAirplane

«NOP_Rule» rlScenarioMoving

«NOP_FBE» Allegro_Scenario

«NOP_FBE» Helicopter

«NOP_FBE» Allegro_Bullet

«NOP_FBE» Stage 1

«NOP_FBE» Airplane

mtDraw()

«InstigationNotifiesMethod»

desenhar (Allegro)

mtDraw()

«InstigationNotifiesMethod»

desenhar (Allegro)

mtBlit()

«InstigationNotifiesMethod»

desenhar (Allegro)

mtClear()

«InstigationNotifiesMethod»

desenhar (Allegro)

mtDraw()

«InstigationNotifiesMethod»

desenhar (Allegro)

mtProgressUpdating()

«InstigationNotifiesMethod»

mostrar mensagem (Allegro)

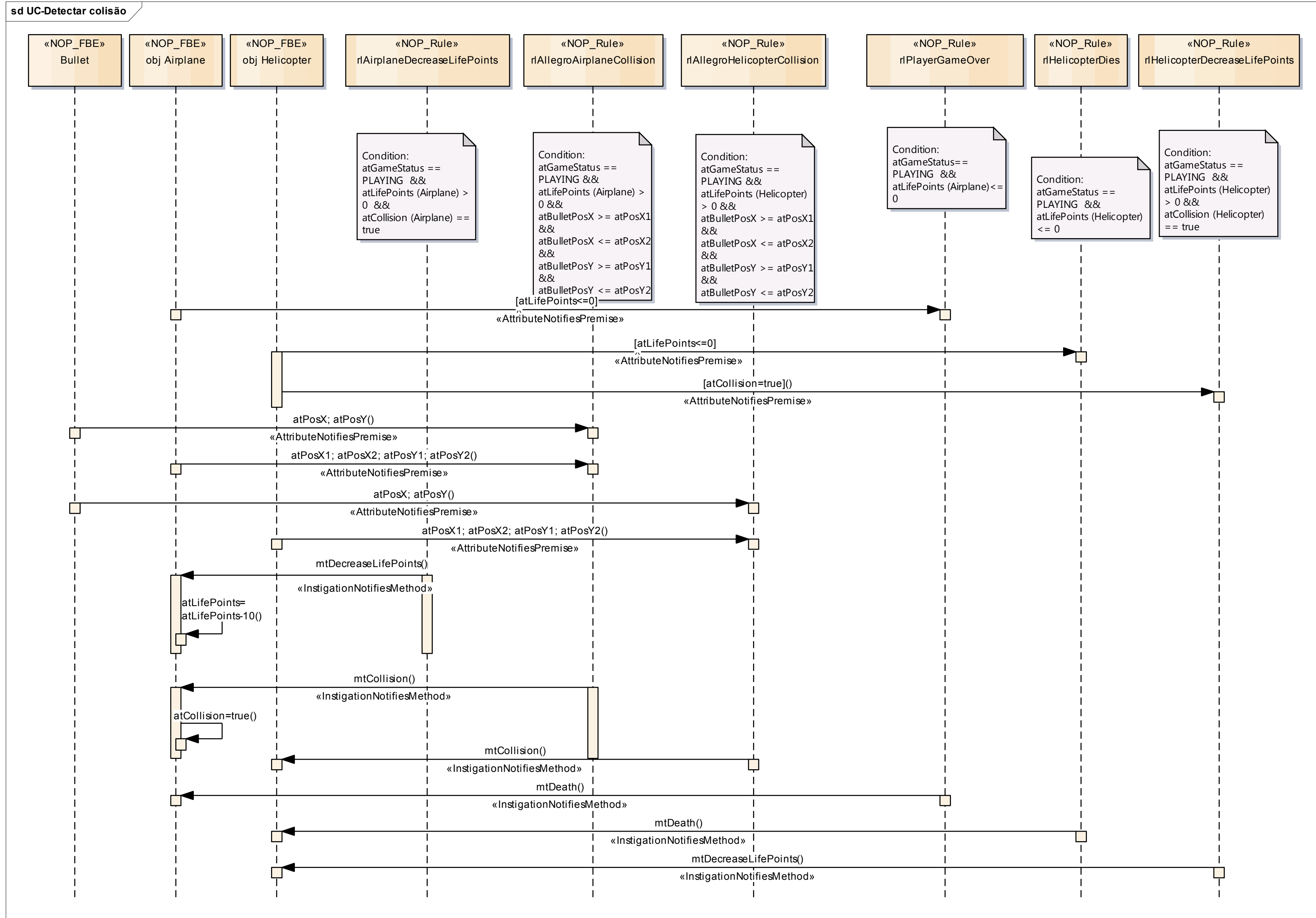
mtDraw()

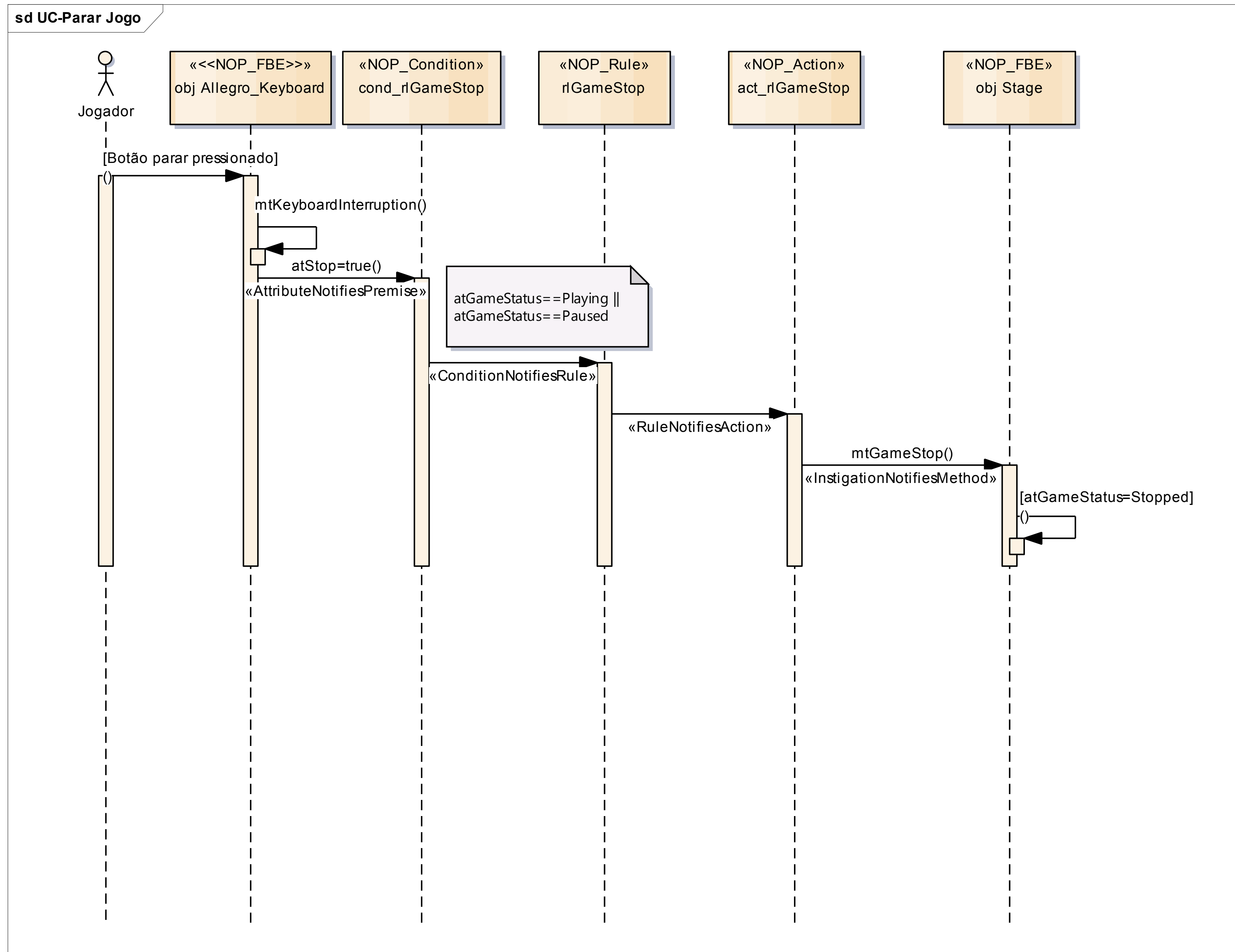
«InstigationNotifiesMethod»

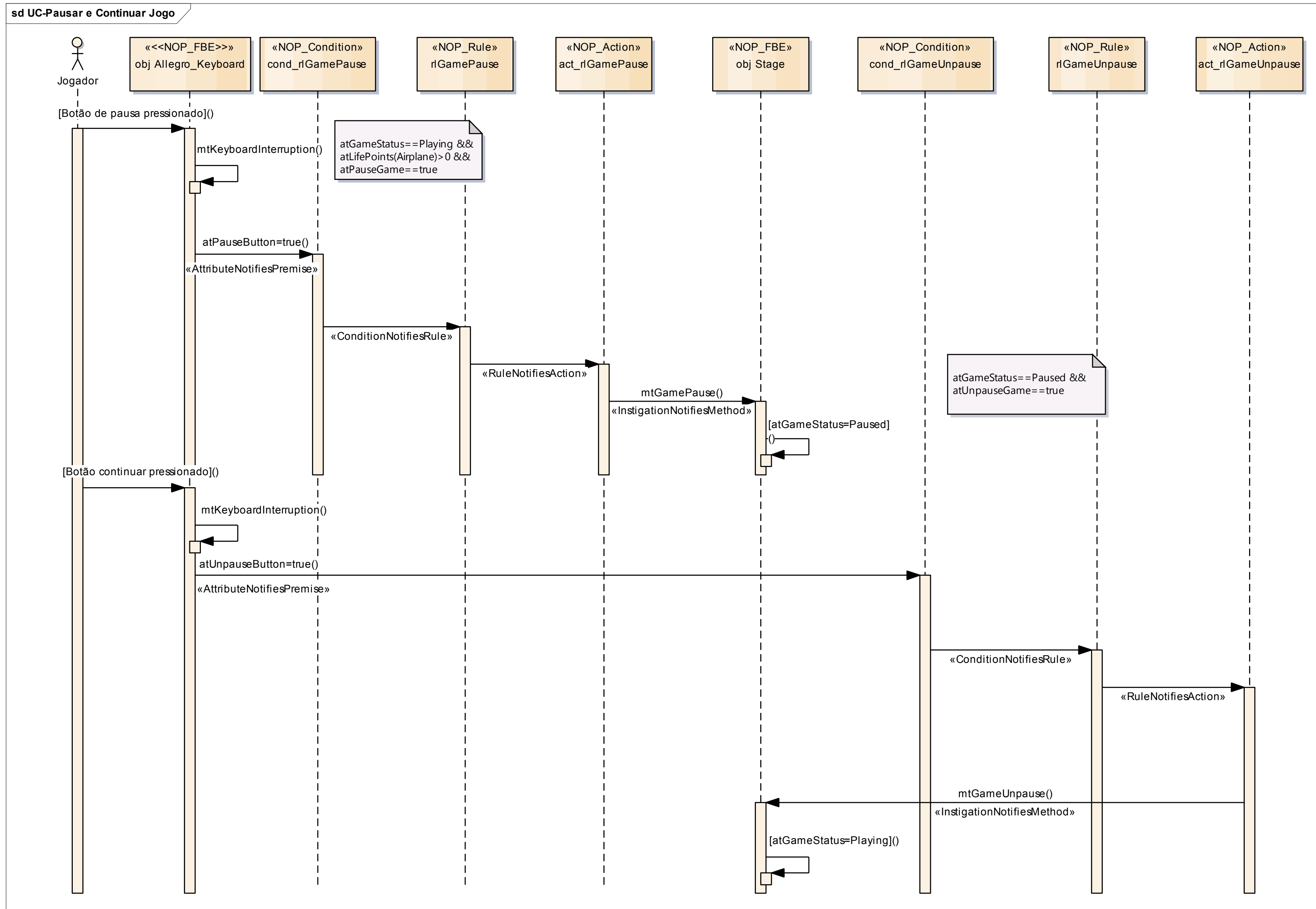
mtMovement()

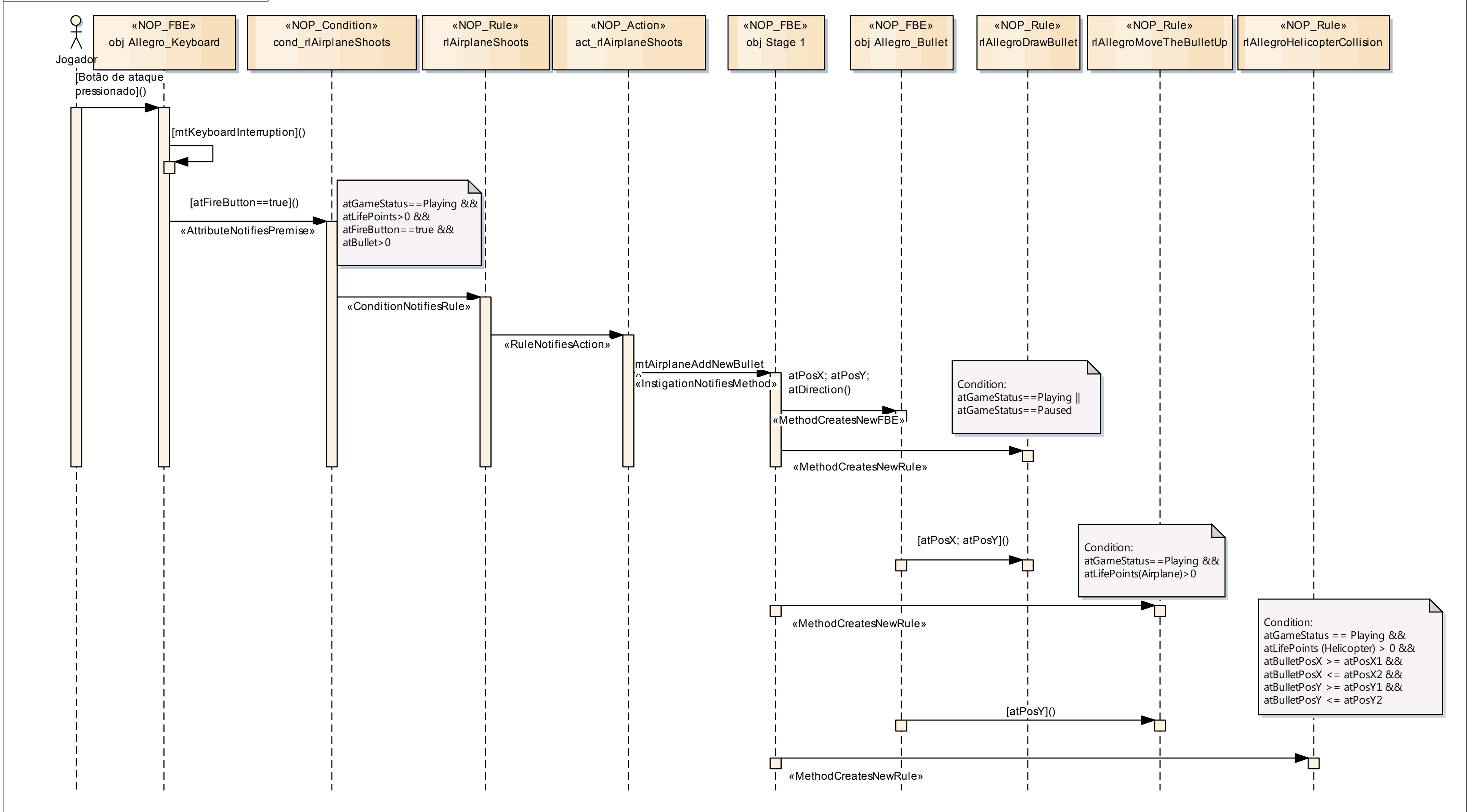
«InstigationNotifiesMethod»

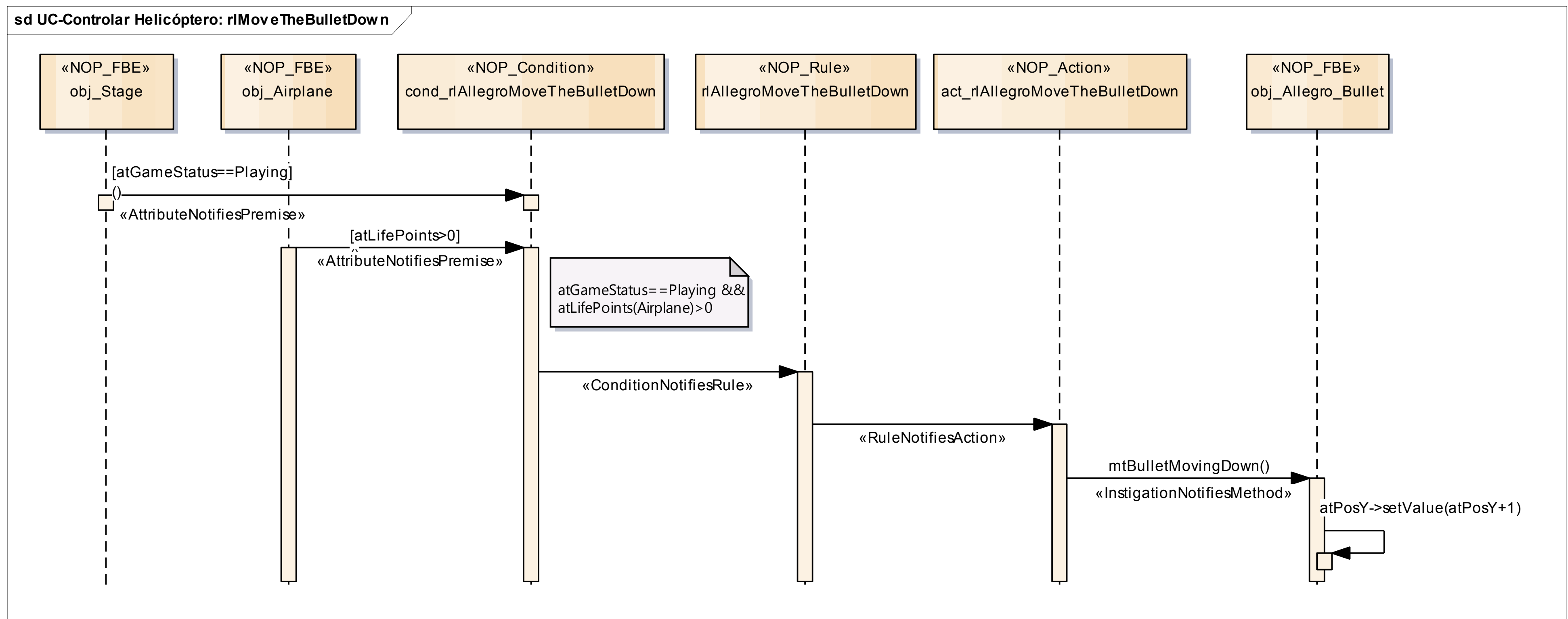
atPosY->setValue(atPosY-1)


sd UC-Detectar colisão
«NOP_FBE» Bullet
«NOP_FBE» obj Airplane
«NOP_FBE» obj Helicopter
«NOP_Rule» rlAirplaneDecreaseLifePoints
«NOP_Rule» rlAllegroAirplaneCollision
«NOP_Rule» rlAllegroHelicopterCollision
«NOP_Rule» rlPlayerGameOver
«NOP_Rule» rlHelicopterDies
«NOP_Rule» rlHelicopterDecreaseLifePoints
Condition: atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atCollision (Airplane) == true
Condition: atGameStatus == PLAYING && atLifePoints (Airplane) > 0 && atBulletPosX >= atPosX1 && atBulletPosX <= atPosX2 && atBulletPosY >= atPosY1 && atBulletPosY <= atPosY2
Condition: atGameStatus == PLAYING && atLifePoints (Helicopter) > 0 && atBulletPosX >= atPosX1 && atBulletPosX <= atPosX2 && atBulletPosY >= atPosY1 && atBulletPosY <= atPosY2
Condition: atGameStatus== PLAYING && atLifePoints (Airplane)<= 0
Condition: atGameStatus == PLAYING && atLifePoints (Helicopter) <= 0
Condition: atGameStatus == PLAYING && atLifePoints (Helicopter) > 0 && atCollision (Helicopter) == true
[atLifePoints<=0]
«AttributeNotifiesPremise»
[atLifePoints<=0]
«AttributeNotifiesPremise»
[atCollision=true]()
«AttributeNotifiesPremise»
atPosX; atPosY()
«AttributeNotifiesPremise»
atPosX1; atPosX2; atPosY1; atPosY2()
«AttributeNotifiesPremise»
atPosX; atPosY()
«AttributeNotifiesPremise»
atPosX1; atPosX2; atPosY1; atPosY2()
«AttributeNotifiesPremise»
mtDecreaseLifePoints()
«InstigationNotifiesMethod»
atLifePoints= atLifePoints-10()
mtCollision()
«InstigationNotifiesMethod»
atCollision=true()
mtCollision()
«InstigationNotifiesMethod»
mtDeath()
«InstigationNotifiesMethod»
mtDeath()
«InstigationNotifiesMethod»
mtDecreaseLifePoints()
«InstigationNotifiesMethod»

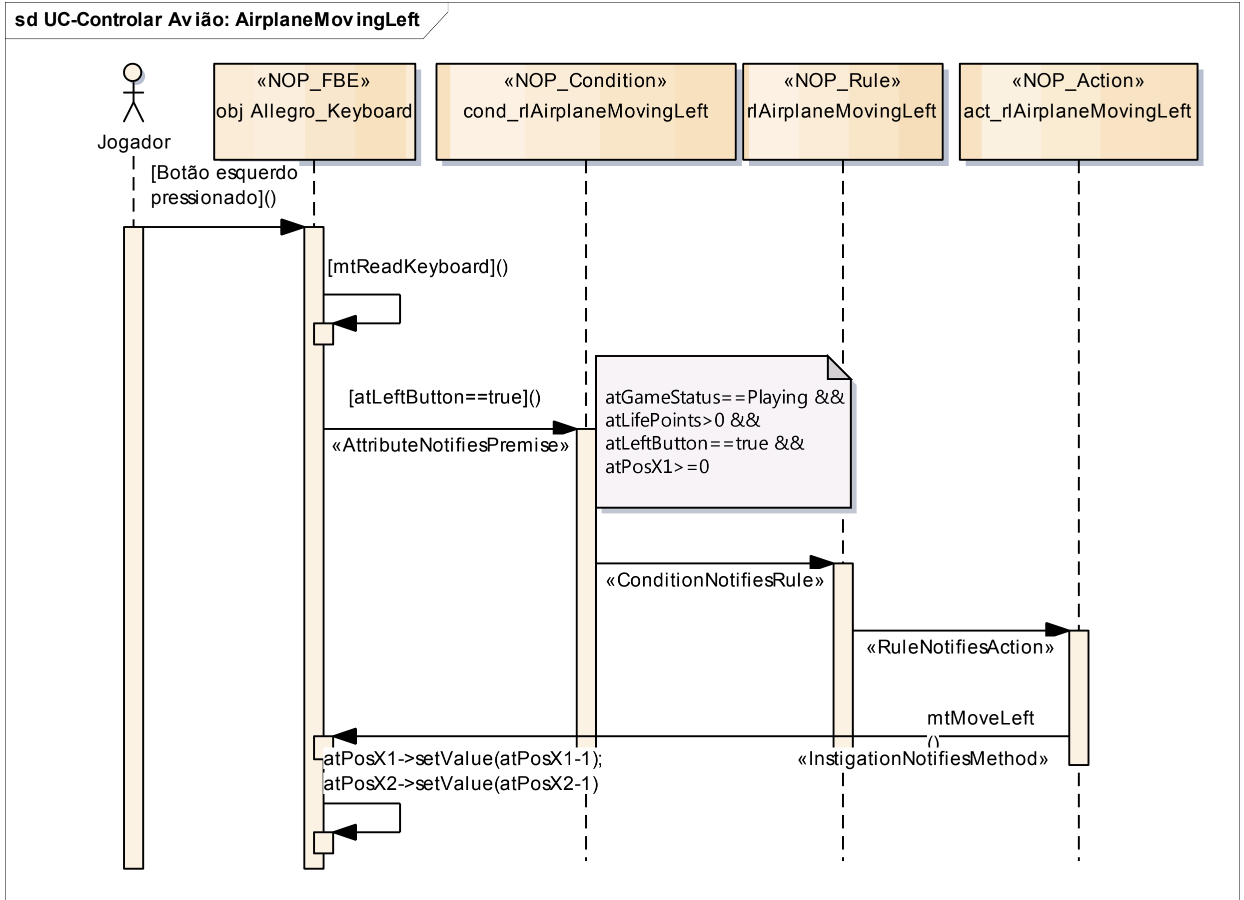

sd UC-Parar Jogo
Jogador
«<<NOP_FBE>>»
obj Allegro_Keyboard
«NOP_Condition»
cond_rlGameStop
«NOP_Rule»
rlGameStop
«NOP_Action»
act_rlGameStop
«NOP_FBE»
obj Stage
[Botão parar pressionado]
()
mtKeyboardInterruption()
atStop=true()
«AttributeNotifiesPremise»
atGameStatus==Playing ||
atGameStatus==Paused
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtGameStop()
«InstigationNotifiesMethod»
[atGameStatus=Stopped]
()

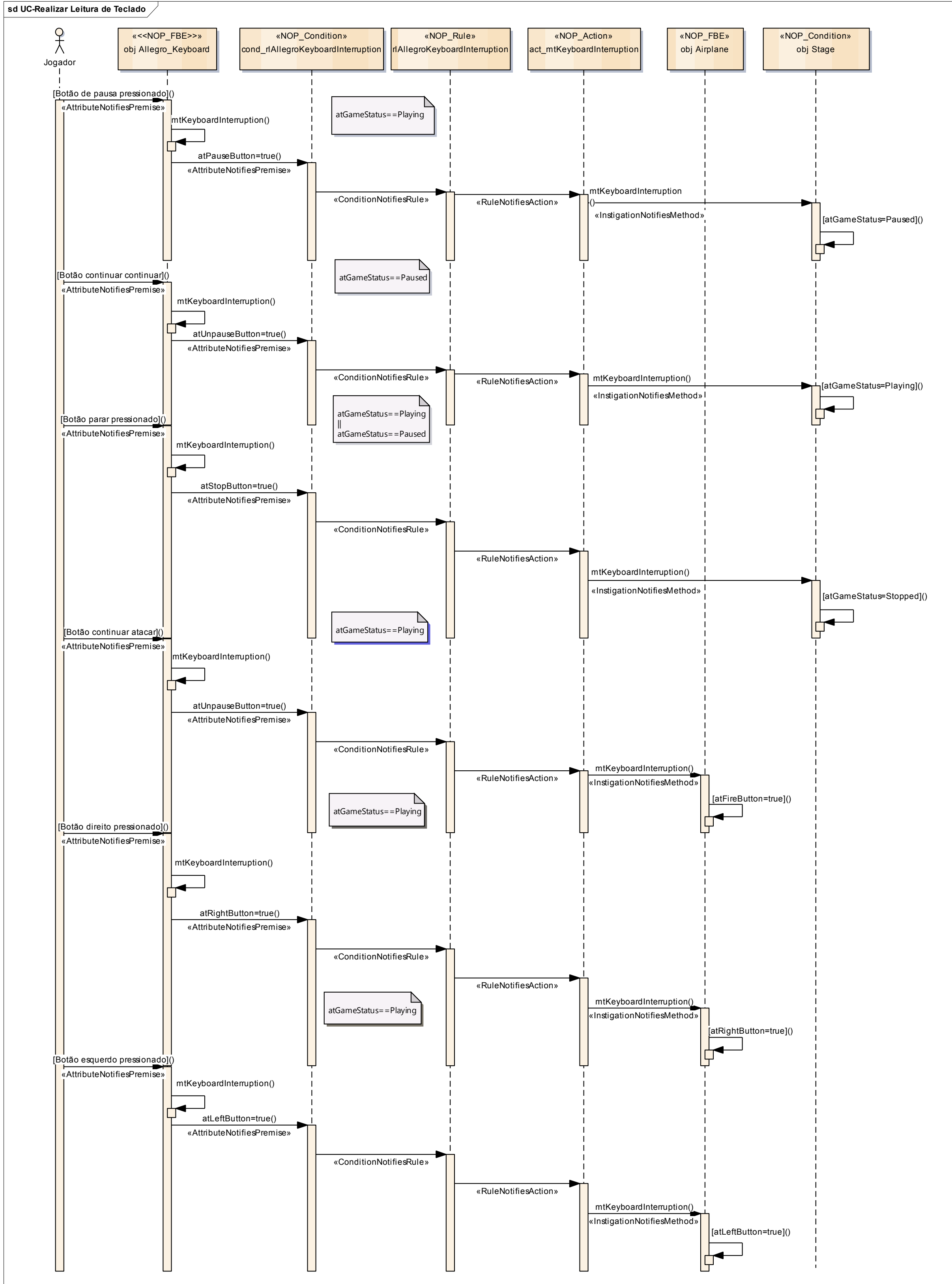

sd UC-Pausar e Continuar Jogo
Jogador
«<<NOP_FBE>>»
obj Allegro_Keyboard
«NOP_Condition»
cond_rlGamePause
«NOP_Rule»
rlGamePause
«NOP_Action»
act_rlGamePause
«NOP_FBE»
obj Stage
«NOP_Condition»
cond_rlGameUnpause
«NOP_Rule»
rlGameUnpause
«NOP_Action»
act_rlGameUnpause
[Botão de pausa pressionado]()
mtKeyboardInterruption()
atGameStatus==Playing &&
atLifePoints(Airplane)>0 &&
atPauseGame==true
atPauseButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtGamePause()
«InstigationNotifiesMethod»
[atGameStatus=Paused]
()
atGameStatus==Paused &&
atUnpauseGame==true
[Botão continuar pressionado]()
mtKeyboardInterruption()
atUnpauseButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtGameUnpause()
«InstigationNotifiesMethod»
[atGameStatus=Playing]()


sd UC-Controlar Avião: mtAirplaneAddNewBullet
Jogador
«NOP_FBE» obj Allegro_Keyboard
«NOP_Condition» cond_rlAirplaneShoots
«NOP_Rule» rlAirplaneShoots
«NOP_Action» act_rlAirplaneShoots
«NOP_FBE» obj Stage 1
«NOP_FBE» obj Allegro_Bullet
«NOP_Rule» rlAllegroDrawBullet
«NOP_Rule» rlAllegroMoveTheBulletUp
«NOP_Rule» rlAllegroHelicopterCollision
[Botão de ataque pressionado]()
[mtKeyboardInterruption]()
[atFireButton==true]()
«AttributeNotifiesPremise»
atGameStatus==Playing && atLifePoints>0 && atFireButton==true && atBullet>0
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtAirplaneAddNewBullet
«InstigationNotifiesMethod»
atPosX; atPosY; atDirection()
«MethodCreatesNewFBE»
Condition: atGameStatus==Playing || atGameStatus==Paused
«MethodCreatesNewRule»
[atPosX; atPosY]()
Condition: atGameStatus==Playing && atLifePoints(Airplane)>0
«MethodCreatesNewRule»
Condition: atGameStatus == Playing && atLifePoints (Helicopter) > 0 && atBulletPosX >= atPosX1 && atBulletPosX <= atPosX2 && atBulletPosY >= atPosY1 && atBulletPosY <= atPosY2
[atPosY]()
«MethodCreatesNewRule»

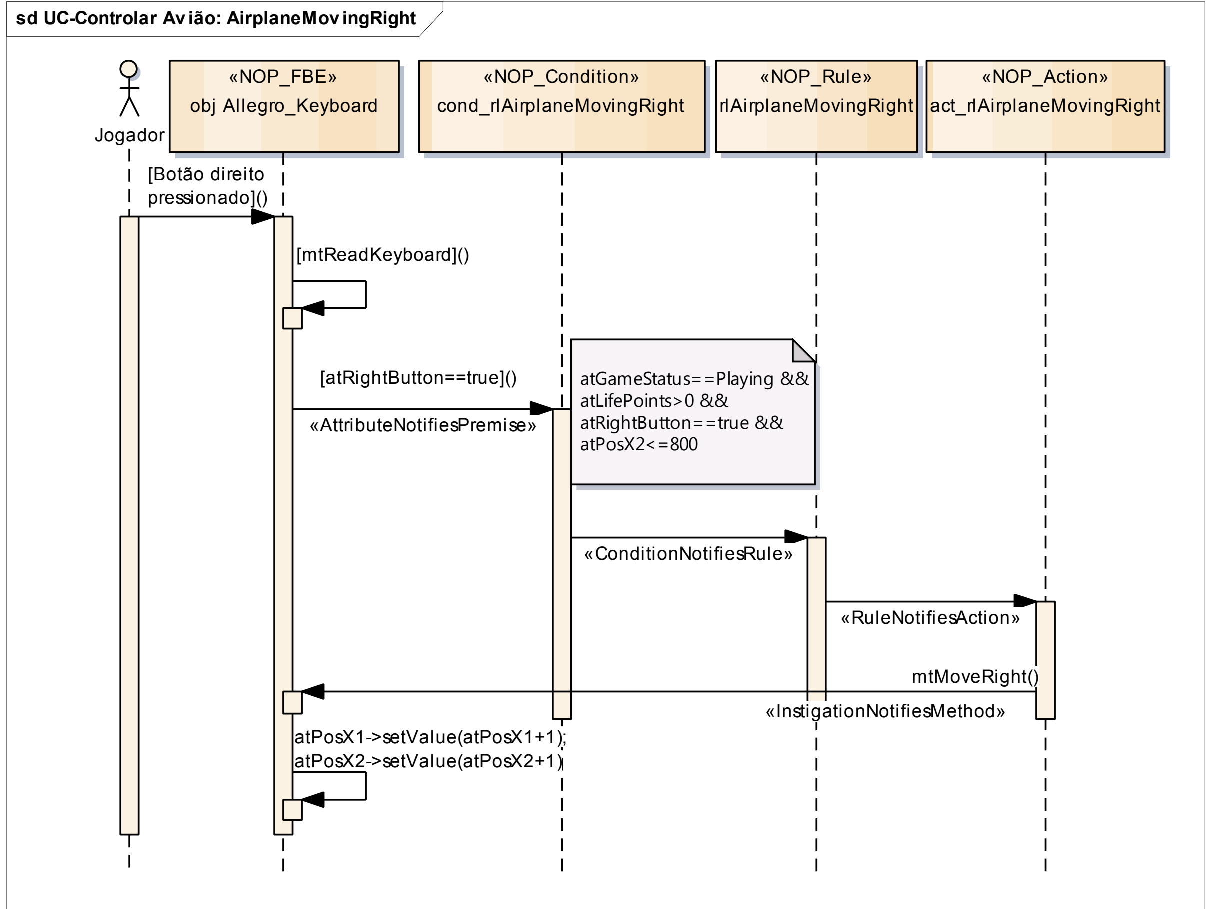


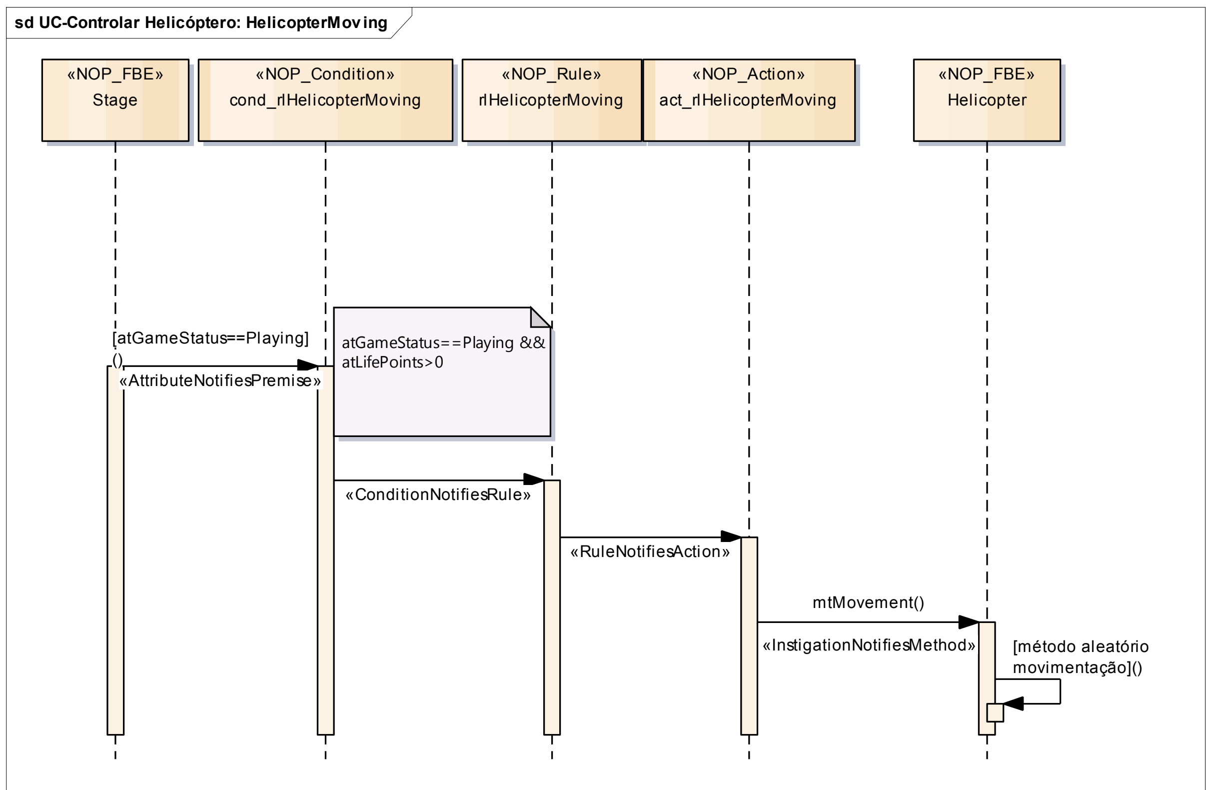
sd UC-Controlar Helicóptero: rlMoveTheBulletDown
«NOP_FBE»
obj_Stage
«NOP_FBE»
obj_Airplane
«NOP_Condition»
cond_rlAllegroMoveTheBulletDown
«NOP_Rule»
rlAllegroMoveTheBulletDown
«NOP_Action»
act_rlAllegroMoveTheBulletDown
«NOP_FBE»
obj_Allegro_Bullet
[atGameStatus==Playing]
()
«AttributeNotifiesPremise»
[atLifePoints>0]
«AttributeNotifiesPremise»
atGameStatus==Playing &&
atLifePoints(Airplane)>0
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtBulletMovingDown()
«InstigationNotifiesMethod»
atPosY->setValue(atPosY+1)


sd UC-Controlar Avião: AirplaneMovingLeft
Jogador
«NOP_FBE»
obj Allegro_Keyboard
«NOP_Condition»
cond_rlAirplaneMovingLeft
«NOP_Rule»
rlAirplaneMovingLeft
«NOP_Action»
act_rlAirplaneMovingLeft
[Botão esquerdo pressionado]()
[mtReadKeyboard]()
[atLeftButton==true]()
«AttributeNotifiesPremise»
atGameStatus==Playing &&
atLifePoints>0 &&
atLeftButton==true &&
atPosX1>=0
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtMoveLeft
()
«InstigationNotifiesMethod»
atPosX1->setValue(atPosX1-1);
atPosX2->setValue(atPosX2-1)



sd UC-Realizar Leitura de Teclado
Jogador
«<<NOP_FBE>>»
obj Allegro_Keyboard
«NOP_Condition»
cond_rlAllegroKeyboardInterruption
«NOP_Rule»
rlAllegroKeyboardInterruption
«NOP_Action»
act_mtKeyboardInterruption
«NOP_FBE»
obj Airplane
«NOP_Condition»
obj Stage
[Botão de pausa pressionado]()
«AttributeNotifiesPremise»
mtKeyboardInterruption()
atGameStatus==Playing
atPauseButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtKeyboardInterruption
()
«InstigationNotifiesMethod»
[atGameStatus=Paused]()
[Botão continuar continuar]()
«AttributeNotifiesPremise»
atGameStatus==Paused
mtKeyboardInterruption()
atUnpauseButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtKeyboardInterruption()
«InstigationNotifiesMethod»
[atGameStatus=Playing]()
atGameStatus==Playing
||
atGameStatus==Paused
[Botão parar pressionado]()
«AttributeNotifiesPremise»
mtKeyboardInterruption()
atStopButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtKeyboardInterruption()
«InstigationNotifiesMethod»
[atGameStatus=Stopped]()
atGameStatus==Playing
[Botão continuar atacar]()
«AttributeNotifiesPremise»
mtKeyboardInterruption()
atUnpauseButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtKeyboardInterruption()
«InstigationNotifiesMethod»
[atFireButton=true]()
atGameStatus==Playing
[Botão direito pressionado]()
«AttributeNotifiesPremise»
mtKeyboardInterruption()
atRightButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
atGameStatus==Playing
mtKeyboardInterruption()
«InstigationNotifiesMethod»
[atRightButton=true]()
[Botão esquerdo pressionado]()
«AttributeNotifiesPremise»
mtKeyboardInterruption()
atLeftButton=true()
«AttributeNotifiesPremise»
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtKeyboardInterruption()
«InstigationNotifiesMethod»
[atLeftButton=true]()


sd UC-Controlar Avião: AirplaneMovingRight
Jogador
«NOP_FBE»
obj Allegro_Keyboard
«NOP_Condition»
cond_rlAirplaneMovingRight
«NOP_Rule»
rlAirplaneMovingRight
«NOP_Action»
act_rlAirplaneMovingRight
[Botão direito pressionado]()
[mtReadKeyboard]()
[atRightButton==true]()
«AttributeNotifiesPremise»
atGameStatus==Playing &&
atLifePoints>0 &&
atRightButton==true &&
atPosX2<=800
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtMoveRight()
«InstigationNotifiesMethod»
atPosX1->setValue(atPosX1+1);
atPosX2->setValue(atPosX2+1)


sd UC-Controlar Helicóptero: HelicopterMoving
«NOP_FBE»
Stage
«NOP_Condition»
cond_rlHelicopterMoving
«NOP_Rule»
rlHelicopterMoving
«NOP_Action»
act_rlHelicopterMoving
«NOP_FBE»
Helicopter
[atGameStatus==Playing]
()
«AttributeNotifiesPremise»
atGameStatus==Playing &&
atLifePoints>0
«ConditionNotifiesRule»
«RuleNotifiesAction»
mtMovement()
«InstigationNotifiesMethod»
[método aleatório
movimentação]()

APÊNDICE E – Código fonte em framework PON Otimizado das Premisses e Rules do software desenvolvido

```
PREMISE(prAirplaneCollisionTrue,airplane->atCollision,true,Premise::EQUAL,
Premise::STANDARD, false);

PREMISE(prAirplaneFireButtonTrue, allegro_keyboard-> atFireButton, true,
Premise::EQUAL, Premise::STANDARD, false);

PREMISE(prAllegroLeftButtonFalse, allegro_keyboard->
atAirplaneLeftButton,false,Premise::EQUAL, Premise::STANDARD, false);

PREMISE(prAirplaneLeftButtonTrue, allegro_keyboard ->
atAirplaneLeftButton,true,Premise::EQUAL, Premise::STANDARD, false);

PREMISE(prAirplaneLimLeftScreen, airplane-> atPosX1,0,Premise::GREATERTHAN,
Premise::STANDARD, false);

PREMISE(prAirplaneLimRightScreen, airplane-> atPosX2,800,Premise::SMALLERTHAN,
Premise::STANDARD, false);

PREMISE(prAirplaneRightButtonFalse, allegro_keyboard ->
atRightButton,false,Premise::EQUAL, Premise::STANDARD, false);

PREMISE(prAirplaneRightButtonTrue, allegro_keyboard ->
atRightButton,true,Premise::EQUAL, Premise::STANDARD, false);

PREMISE(prAirplaneLifePointsGreaterZero,airplane->
atLifePoints,0,Premise::GREATERTHAN, Premise::STANDARD, false);

PREMISE(prAirplaneLifePointsZero,airplane->
atLifePoints,0,Premise::SMALLEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletXGreaterX1Airplane,bullet->atPosX,airplane->
atPosX1,Premise::GREATEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletXGreaterX1Helicopter,bullet->atPosX,helicopter->
atPosX1,Premise::GREATEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletXSmallerX2Airplane,bullet->atPosX,airplane->
atPosX2,Premise::SMALLEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletXSmallerX2Helicopter,bullet->atPosX,helicopter->
atPosX2,Premise::SMALLEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletYGreaterY1Airplane,bullet->atPosY,airplane->
atPosY1,Premise::GREATEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletYGreaterY1Helicopter,bullet->atPosY,helicopter->
atPosY1,Premise::GREATEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletYSmallerY2Airplane,bullet->atPosY,airplane->
atPosY2,Premise::SMALLEROREQUAL, Premise::STANDARD, false);

PREMISE(prBulletYSmallerY2Helicopter,bullet->atPosY,helicopter->
atPosY2,Premise::SMALLEROREQUAL, Premise::STANDARD, false);
PREMISE(prGameStatusPaused,atGameStatus,1,Premise::EQUAL, Premise::STANDARD,
false);
```

```
PREMISE(prGameStatusPlaying,atGameStatus,2,Premise::EQUAL, Premise::STANDARD,
false);

PREMISE(prGameStatusStopped,atGameStatus,0,Premise::EQUAL, Premise::STANDARD,
false);

PREMISE(prHelicopterCollisionTrue,helicopter-> atCollision,true,Premise::EQUAL,
Premise::STANDARD, false);

PREMISE(prHelicopterLifePointsGreaterZero,helicopter->
atLifePoints,0,Premise::GREATERTHAN, Premise::STANDARD, false);

PREMISE(prHelicopterLifePointsZero,helicopter->
atLifePoints,0,Premise::SMALLEROREQUAL, Premise::STANDARD, false);

PREMISE(prHelicopterTimeToShoot,helicopter->
atTimeToShoot,0,Premise::GREATERTHAN, Premise::STANDARD, false);

PREMISE(prPauseButtonTrue, allegro_keyboard ->
atPauseButton,true,Premise::EQUAL, Premise::STANDARD, false);

PREMISE(prStopButtonTrue, allegro_keyboard->atStopButton,true,Premise::EQUAL,
Premise::STANDARD, false);

PREMISE(prUnpauseButtonTrue, allegro_keyboard->
atUnpauseButton,true,Premise::EQUAL, Premise::STANDARD, false);
```

```
//Rule 1
rlScenarioMoving = new
RuleObject("rlScenarioMoving",SingletonScheduler::getInstance(),Condition::CONJU
NCTION);
rlScenarioMoving->addPremise(prGameStatusPlaying);
rlScenarioMoving->addPremise(prAirplaneLifePointsGreaterZero);
rlScenarioMoving->addMethod(cen1->mtMoviment);
rlScenarioMoving->end();
```

```
//Rule 2
rlProgressUpdating = new
RuleObject("rlProgressUpdating",SingletonScheduler::getInstance(),Condition::DIS
JUNCTION);
rlProgressUpdating->addPremise(prGameStatusPlaying);
rlProgressUpdating->addPremise(prGameStatusPaused);
rlProgressUpdating->addMethod(mtProgressUpdating);
rlProgressUpdating->end();
```

```
//Rule 3
rlAirplaneStayInPosition = new
RuleObject("rlAirplaneMoving",SingletonScheduler::getInstance(),Condition::CONJU
NCTION);
rlAirplaneStayInPosition->addPremise(prGameStatusPlaying);
rlAirplaneStayInPosition->addPremise(prAirplaneLifePointsGreaterZero);
rlAirplaneStayInPosition->addPremise(prAirplaneRightButtonFalse);
rlAirplaneStayInPosition->addPremise(prAirplaneLeftButtonFalse);
rlAirplaneStayInPosition->addMethod(airplane->mtCharacterMoviment);
rlAirplaneStayInPosition->end();
```

```
//Rule 4
rlAirplaneMovingLeft = new
RuleObject("rlAirplaneMovingLeft",SingletonScheduler::getInstance(),Condition::C
ONJUNCTION);
rlAirplaneMovingLeft->addPremise(prGameStatusPlaying);
rlAirplaneMovingLeft->addPremise(prAirplaneLifePointsGreaterZero);
rlAirplaneMovingLeft->addPremise(prAirplaneLeftButtonTrue);
rlAirplaneMovingLeft->addPremise(prAirplaneLimLeftScreen);
rlAirplaneMovingLeft->addMethod (airplane->mtAirplaneMoveLeft);
rlAirplaneMovingLeft->end();
```

```
//Rule 5
rlAirplaneMovingRight = new
RuleObject("rlAirplaneMovingRight",SingletonScheduler::getInstance(),Condition::
CONJUNCTION);
rlAirplaneMovingRight->addPremise(prGameStatusPlaying);
rlAirplaneMovingRight->addPremise(prAirplaneLifePointsGreaterZero);
rlAirplaneMovingRight->addPremise(prAirplaneRightButtonTrue);
rlAirplaneMovingRight->addPremise(prAirplaneLimRightScreen);
rlAirplaneMovingRight->addMethod (airplane->mtAirplaneMoveRight);
rlAirplaneMovingRight->end();
```

```
//Rule 6
rlAirplaneShoots = new
RuleObject("rlAirplaneShoots",SingletonScheduler::getInstance(),Condition::CONJU
NCTION);
rlAirplaneShoots->addPremise(prGameStatusPlaying);
rlAirplaneShoots->addPremise(prAirplaneLifePointsGreaterZero);
rlAirplaneShoots->addPremise(prAirplaneFireButtonTrue);
rlAirplaneShoots->addMethod(mtAirplaneAddNewBullet);
rlAirplaneShoots->end();
```

```
//Rule 7
rlHelicopterShoots = new
RuleObject("rlHelicopterShoots",SingletonScheduler::getInstance(),Condition::CON
JUNCTION);
rlHelicopterShoots->addPremise(prGameStatusPlaying);
rlHelicopterShoots->addPremise(prHelicopterLifePointsGreaterZero);
rlHelicopterShoots->addPremise(prHelicopterTimeToShoot);
rlHelicopterShoots->addMethod(mtHelicopterAddNewBullet);
rlHelicopterShoots->end();
```

```
//Rule 8
rlAllegroDrawHelicopter = new
RuleObject("rlAllegroDrawHelicopter",SingletonScheduler::getInstance(),Condition
::CONJUNCTION);
rlAllegroDrawHelicopter->addSubCondition(Condition::DISJUNCTION,false);
rlAllegroDrawHelicopter->addPremiseToSubCondition(prGameStatusPlaying);
rlAllegroDrawHelicopter->addPremiseToSubCondition(prGameStatusPaused);
rlAllegroDrawHelicopter->addSubCondition(Condition::CONJUNCTION,false);
rlAllegroDrawHelicopter-
>addPremiseToSubCondition(prHelicopterLifePointsGreaterZero);
rlAllegroDrawHelicopter->addMethod(helicopter->mtCharacterDraw);
rlAllegroDrawHelicopter->end();
```

```
//Rule 9
rlAirplaneDecreaseLifePoints = new
RuleObject("rlAirplaneDecreaseLifePoints",SingletonScheduler::getInstance(),Cond
ition::CONJUNCTION);
rlAirplaneDecreaseLifePoints->addPremise(prGameStatusPlaying);
rlAirplaneDecreaseLifePoints->addPremise(prAirplaneLifePointsGreaterZero);
rlAirplaneDecreaseLifePoints->addPremise(prAirplaneCollisionTrue);
rlAirplaneDecreaseLifePoints->addMethod(airplane->mtDecreaseLifePoints);
rlAirplaneDecreaseLifePoints->end();
```

```
//Rule 10
rlHelicopterDecreaseLifePoints = new
RuleObject("rlHelicopterDecreaseLifePoints",SingletonScheduler::getInstance(),Co
ndition::CONJUNCTION);
rlHelicopterDecreaseLifePoints->addPremise(prGameStatusPlaying);
rlHelicopterDecreaseLifePoints->addPremise(prAirplaneLifePointsGreaterZero);
rlHelicopterDecreaseLifePoints->addPremise(prHelicopterCollisionTrue);
rlHelicopterDecreaseLifePoints->addMethod(helicopter->mtDecreaseLifePoints);
rlHelicopterDecreaseLifePoints->end();
```

```
//Rule 11
rlHelicopterDies = new
RuleObject("rlHelicopterDies",SingletonScheduler::getInstance(),Condition::CONJU
NCTION);
rlHelicopterDies->addPremise(prGameStatusPlaying);
rlHelicopterDies->addPremise(prHelicopterLifePointsZero);
rlHelicopterDies->addMethod(helicopter->mtCharacterDeath);
rlHelicopterDies->end();
```

```
//Rule 12
rlPlayerGameOver = new
RuleObject("rlPlayerGameOver",SingletonScheduler::getInstance(),Condition::CONJU
NCTION);
rlPlayerGameOver->addPremise(prGameStatusPlaying);
rlPlayerGameOver->addPremise(prAirplaneLifePointsZero);
rlPlayerGameOver->addMethod(airplane->mtAirplaneDeath);
rlPlayerGameOver->end();
```

```
//Rule 13
rlGamePause = new
RuleObject("rlGamePause",SingletonScheduler::getInstance(),Condition::CONJUNCTIO
N);
rlGamePause->addPremise(prGameStatusPlaying);
rlGamePause->addPremise(prAirplaneLifePointsGreaterZero);
rlGamePause->addPremise(prPauseButtonTrue);
rlGamePause->addMethod(mtGamePause);
rlGamePause->end();
```

```
//Rule 14
rlGameUnpause = new
RuleObject("rlGameUnpause",SingletonScheduler::getInstance(),Condition::CONJUNCT
ION);
rlGameUnpause->addPremise(prGameStatusPaused);
rlGameUnpause->addPremise(prUnpauseButtonTrue);
rlGameUnpause->addMethod(mtGameUnpause);
rlGameUnpause->end();
```

```
//Rule 15
rlGameStop = new
RuleObject("rlGameStop",SingletonScheduler::getInstance(),Condition::CONJUNCTION
);
rlGameStop->addSubCondition(Condition::DISJUNCTION,false);
rlGameStop->addPremiseToSubCondition(prGameStatusPlaying);
rlGameStop->addPremiseToSubCondition(prGameStatusPaused);
rlGameStop->addSubCondition(Condition::CONJUNCTION,false);
rlGameStop->addPremiseToSubCondition(prStopButtonTrue);
rlGameStop->addMethod(mtGameStop);
rlGameStop->end();
```

```
//Rule 16
rlAllegroClear = new
RuleObject("rlAllegroClear",SingletonScheduler::getInstance(),Condition::DISJUNC
TION);
rlAllegroClear->addPremise(prGameStatusPaused);
rlAllegroClear->addPremise(prGameStatusPlaying);
rlAllegroClear->addMethod (mtAllegroClear);
rlAllegroClear->end();
```

```
//Rule 17
rlAllegroBlit = new
RuleObject("rlAllegroBlit",SingletonScheduler::getInstance(),Condition::DISJUNCT
ION);
rlAllegroBlit->addPremise(prGameStatusPaused);
rlAllegroBlit->addPremise(prGameStatusPlaying);
rlAllegroBlit->addMethod (mtAllegroBlit);
rlAllegroBlit->end();
```

```
//Rule 18
rlAllegroKeyboard = new
RuleObject("rlAllegroKeyboard",SingletonScheduler::getInstance(),Condition::DISJ
UNCTION);
rlAllegroKeyboard->addPremise(prGameStatusPaused);
rlAllegroKeyboard->addPremise(prGameStatusPlaying);
rlAllegroKeyboard->addMethod(mtKeyboardInterruption);
rlAllegroKeyboard->end();
```

```
//Rule 19
rlAllegroDrawAirplane = new
RuleObject("rlAllegroDrawAirplane",SingletonScheduler::getInstance(),Condition::
CONJUNCTION);
rlAllegroDrawAirplane->addSubCondition(Condition::DISJUNCTION,false);
rlAllegroDrawAirplane->addPremiseToSubCondition(prGameStatusPaused);
rlAllegroDrawAirplane->addPremiseToSubCondition(prGameStatusPlaying);
rlAllegroDrawAirplane->addSubCondition(Condition::CONJUNCTION,false);
rlAllegroDrawAirplane-
>addPremiseToSubCondition(prAirplaneLifePointsGreaterZero);
rlAllegroDrawAirplane->addMethod (airplane->mtCharacterDraw);
rlAllegroDrawAirplane->end();
```

```
//Rule 20
rlAllegroDrawTheScenario = new
RuleObject("rlScenarioDrawing",SingletonScheduler::getInstance(),Condition::DISJ
UNCTION);
rlAllegroDrawThescenario->addPremise(prGameStatusPaused);
rlAllegroDrawThescenario->addPremise(prGameStatusPlaying);
rlAllegroDrawThescenario->addMethod(cen1->mtDraw);
rlAllegroDrawThescenario->end();
```

```
//Rule 21 - Allegro draw Bullet
rlAllegroDrawBullet = new
RuleObject("rlAllegroDrawBullet",SingletonScheduler::getInstance(),Condition::DI
SJUNCTION);
rlAllegroDrawbullet->addPremiseToSubCondition(prGameStatusPlaying);
rlAllegroDrawbullet->addPremiseToSubCondition(prGameStatusPaused);
rlAllegroDrawbullet->addMethod(bullet->mtBulletDraw);
rlAllegroDrawbullet->end();
```

```
//Rule 22 - Helicopter Collision
rlAllegroHelicopterCollision = new
RuleObject("rlAllegroHelicopterCollision",SingletonScheduler::getInstance(),Cond
ition::CONJUNCTION);
rlAllegroHelicopterCollision->addPremise(prGameStatusPlaying);
rlAllegroHelicopterCollision->addPremise(prAirplaneLifePointsGreaterZero);
rlAllegroHelicopterCollision->addPremise(prBulletXGreaterX1Helicopter);
rlAllegroHelicopterCollision->addPremise(prBulletXSmallerX2Helicopter);
rlAllegroHelicopterCollision->addPremise(prBulletYGreaterY1Helicopter);
rlAllegroHelicopterCollision->addPremise(prBulletYSmallerY2Helicopter);
rlAllegroHelicopterCollision->addMethod(helicopter->mtCollision);
rlAllegroHelicopterCollision->end();
```

```
//Rule 23 - Airplane Collision
rlAllegroAirplaneCollision = new
RuleObject("rlAllegroAirplaneCollision",SingletonScheduler::getInstance(),Condit
ion::CONJUNCTION);
rlAllegroAirplaneCollision->addPremise(prGameStatusPlaying);
rlAllegroAirplaneCollision->addPremise(prHelicopterLifePointsGreaterZero);
rlAllegroAirplaneCollision->addPremise(prBulletXGreaterX1Airplane);
rlAllegroAirplaneCollision->addPremise(prBulletXSmallerX2Airplane);
rlAllegroAirplaneCollision->addPremise(prBulletYGreaterY1Airplane);
rlAllegroAirplaneCollision->addPremise(prBulletYSmallerY2Airplane);
rlAllegroAirplaneCollision->addMethod(airplane->mtCharacterColide);
rlAllegroAirplaneCollision->end();
```

```
//Rule 24 - Moving bullet
rlAllegroMoveTheBulletUp = new
RuleObject("rlAllegroMovingBullet",SingletonScheduler::getInstance(),Condition::
CONJUNCTION);
rlAllegroMoveTheBulletUp->addPremise(prGameStatusPlaying);
rlAllegroMoveTheBulletUp->addPremise(prAirplaneLifePointsGreaterZero);
rlAllegroMoveTheBulletUp->addMethod(bullet->mtBulletMovimentUp);
rlAllegroMoveTheBulletUp->end();
```

```
//Rule 25
rlHelicopterMoving = new
RuleObject("rlHelicopterMoving",SingletonScheduler::getInstance(),Condition::CON
JUNCTION);
rlHelicopterMoving->addPremiseToSubCondition(prGameStatusPaused);
rlHelicopterMoving->addPremiseToSubCondition(prGameStatusPlaying);
rlHelicopterMoving->
addPremiseToSubCondition(prHelicopterLifePointsGreaterZero);
rlHelicopterMoving->addMethod (helicopter->mtCharacterMoviment);
rlHelicopterMoving->end();
```

```
//Rule 26
rlAllegroMoveTheBulletDown = new
RuleObject("rlAllegroMovingBullet",SingletonScheduler::getInstance(),Condition::
CONJUNCTION);
rlAllegroMoveTheBulletDown->addPremise(prGameStatusPlaying);
rlAllegroMoveTheBulletDown->addPremise(prAirplaneLifePointsGreaterZero);
rlAllegroMoveTheBulletDown->addMethod(bullet->mtBulletMovimentDown);
rlAllegroMoveTheBulletDown->end();
```

APÊNDICE F – Problemas identificáveis no desenvolvimento em PON

Este apêndice apresenta conteúdo empírico baseado na experiência de desenvolvimento em PON do autor.

O desenvolvimento em PON requer atenção especial e alguns cuidados relacionados à declaração dos elementos e seus (possíveis) relacionamentos. Os tópicos a seguir irão apresentar os problemas potenciais que podem ocorrer durante o desenvolvimento em PON.

- Falta de conexão entre objetos

Como pode ser visto na Figura 44, a falta de conexões (ou usos) entre objetos pode ser um problema, pois o caso de uso poderá não ser realizado plenamente.

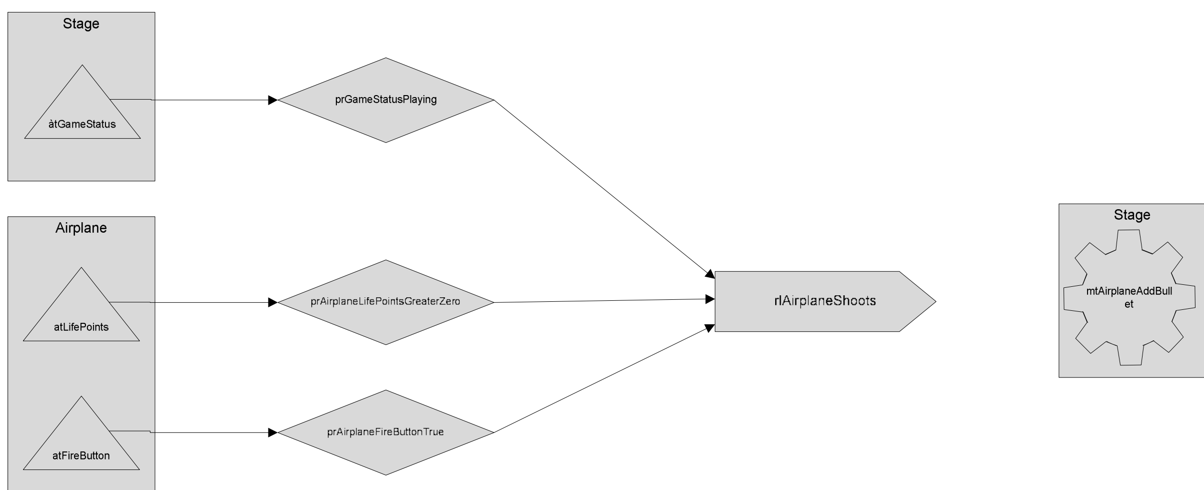


Figura 44 – Falta de conexões entre objetos

- Excesso de conexões entre objetos

O excesso de conexões entre objetos também pode ser um problema conforme ilustra a Figura 45. O projetista deve conhecer quantos são os elementos que estão dependentes de uma determinada entidade PON.

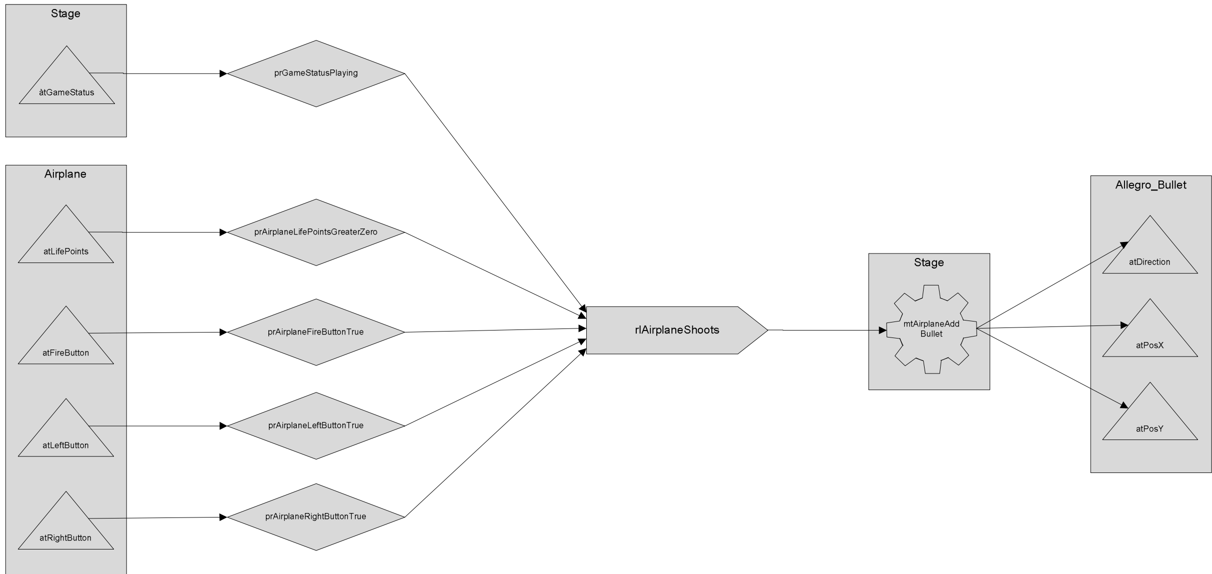


Figura 45 – Excesso de conexões entre objetos

- Ausência de *Attribute* em *FBE*

A ausência de *Attribute* em *FBE* é um problema facilmente identificável no diagrama de objetos PON (Figura 46). Sem determinado *Attribute* não é possível criar *Premise*.

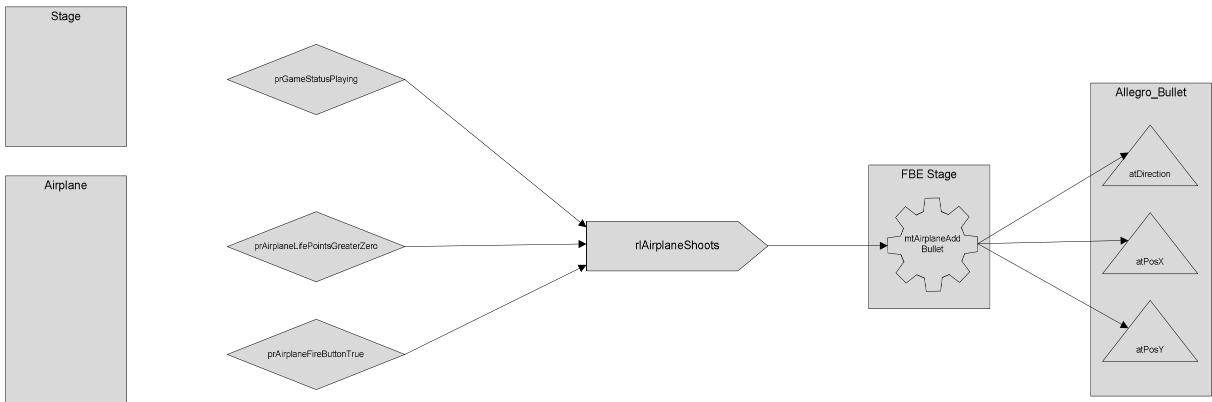


Figura 46 – Ausência de *Attribute* em *FBE*

- *Method* que não altera *Attribute*

Um *Method* que não altera *Attribute* normalmente deveria apenas mostrar alguma mensagem (Figura 47).

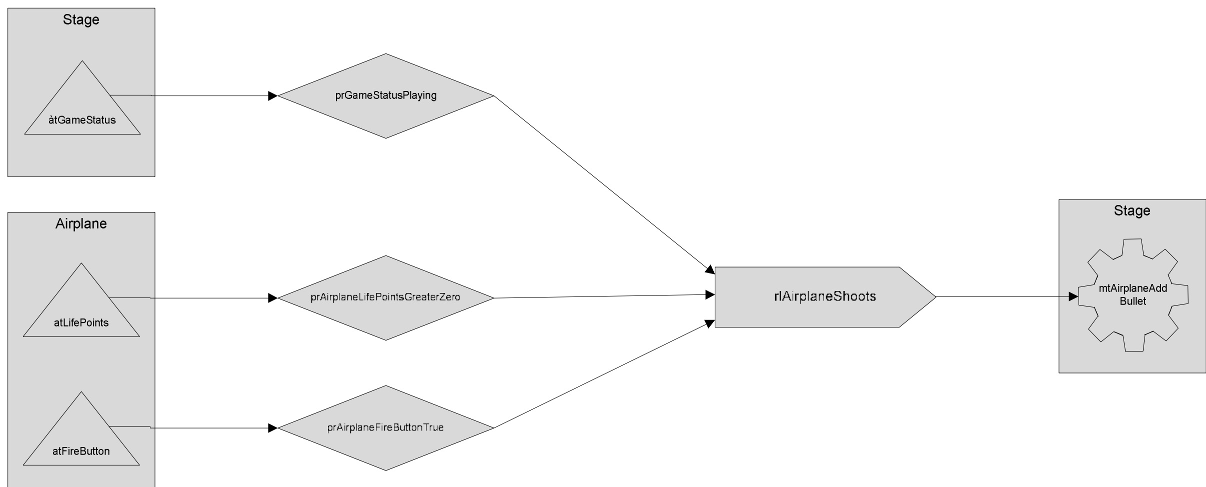


Figura 47 – *Method* que não altera *Attribute*

- *Attribute* que nunca é alterado

Um *Attribute* que nunca é alterado, normalmente, seria utilizado apenas para apresentar seu valor, no entanto, é necessário verificar se este *Attribute* realmente deveria existir (Figura 48).

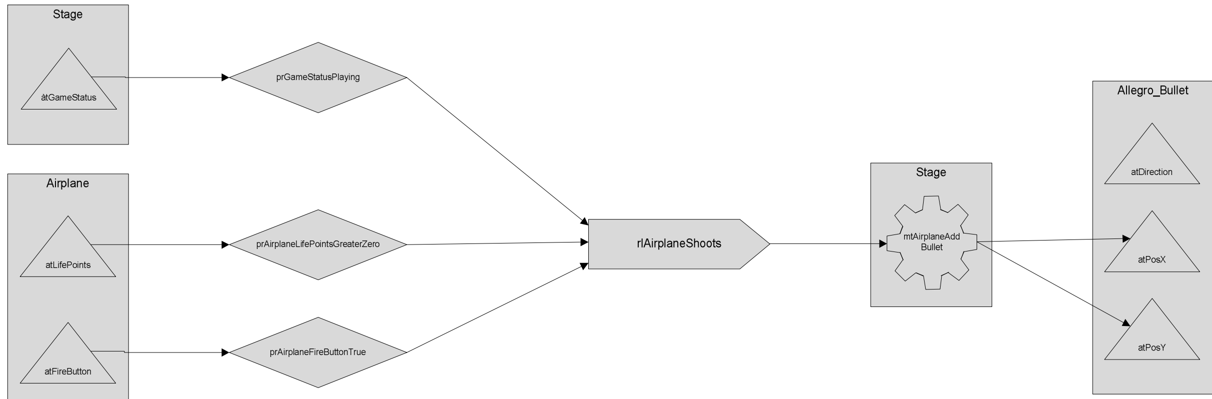


Figura 48 – *Attribute* que nunca é alterado

- *Premise* compartilhada entre *Rules*

A *Premise* compartilhada entre *Rules* incorre em potenciais problemas com relação à alcançabilidade (Figura 49).

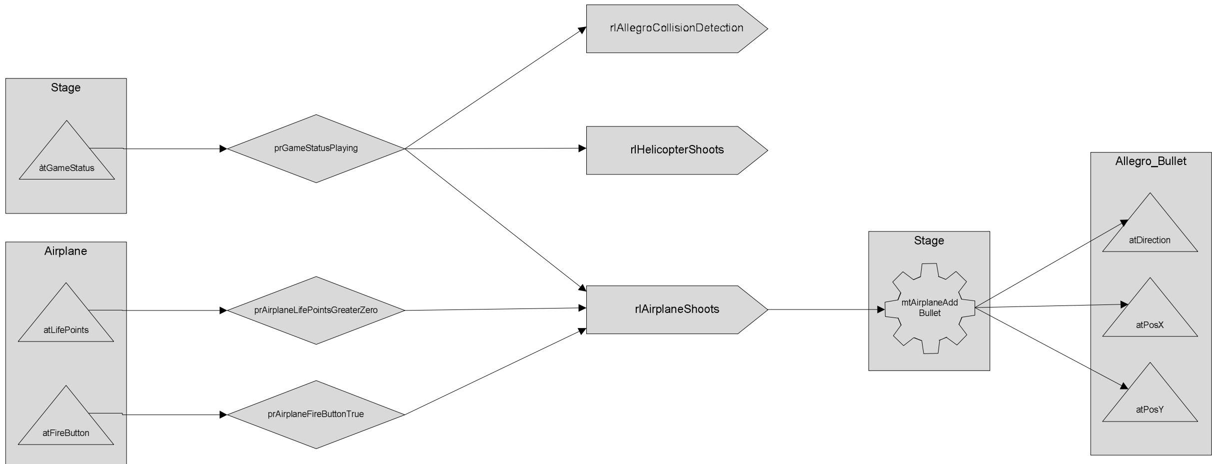


Figura 49 – *Premise* compartilhada entre *Rules*

- *Method* compartilhado entre *Rules*

O mesmo modo, *Method* compartilhado entre *Rules*, também pode incorrer em problemas de alcançabilidade e requerem atenção especial durante o desenvolvimento (Figura 50).

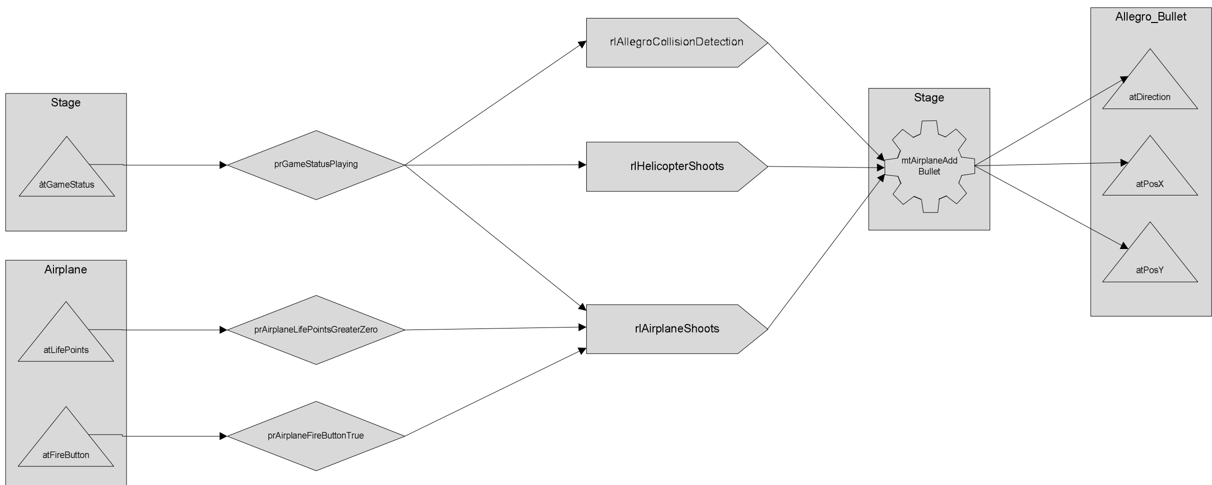


Figura 50 – *Method* compartilhado entre *Rules*

- *Master Rule* compartilhada

Quando uma *Rule* é compartilhada entre outras *Rules*, também fica evidente a alcançabilidade, sendo que a alteração do valor dela pode implicar em outros processamentos (Figura 51).

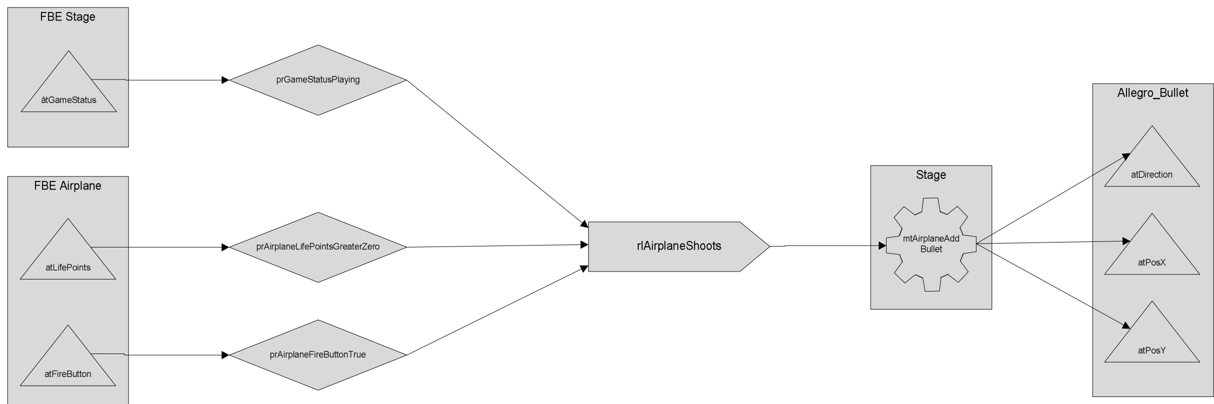


Figura 51 – *Master Rule* compartilhada

- *Attribute* compartilhado entre *Premisses*

Attribute compartilhado entre *Premisses* também requer atenção com relação à alcançabilidade sobre as *Premisses* relacionadas (Figura 52).

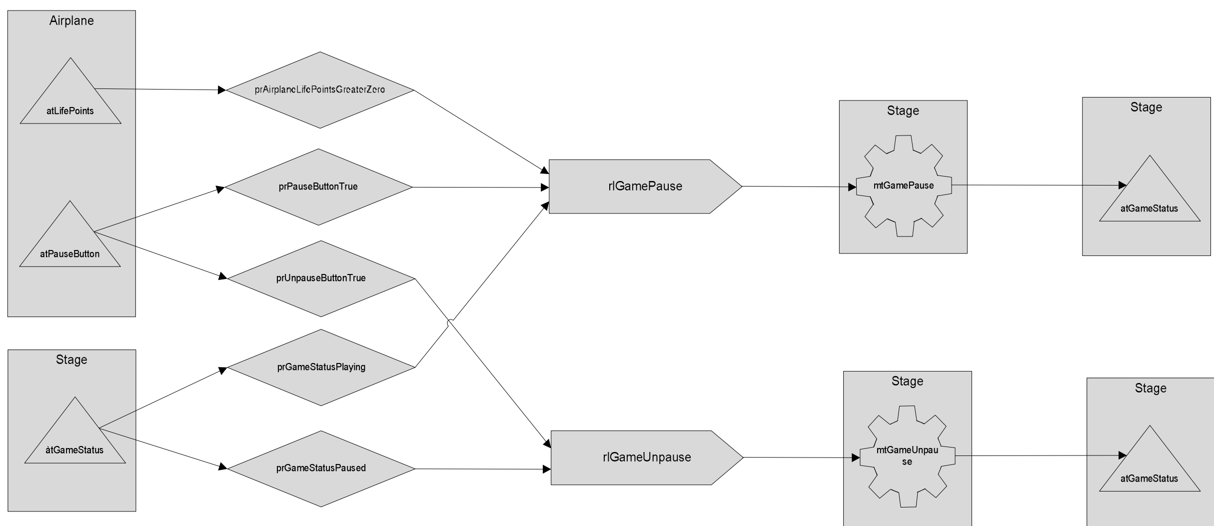


Figura 52 – *Attribute* compartilhado entre *Premisses*

- *Attribute* alterado por mais de um *Method*

Attribute alterado por mais de um *Method* também implica em problemas de alcançabilidade (Figura 53).

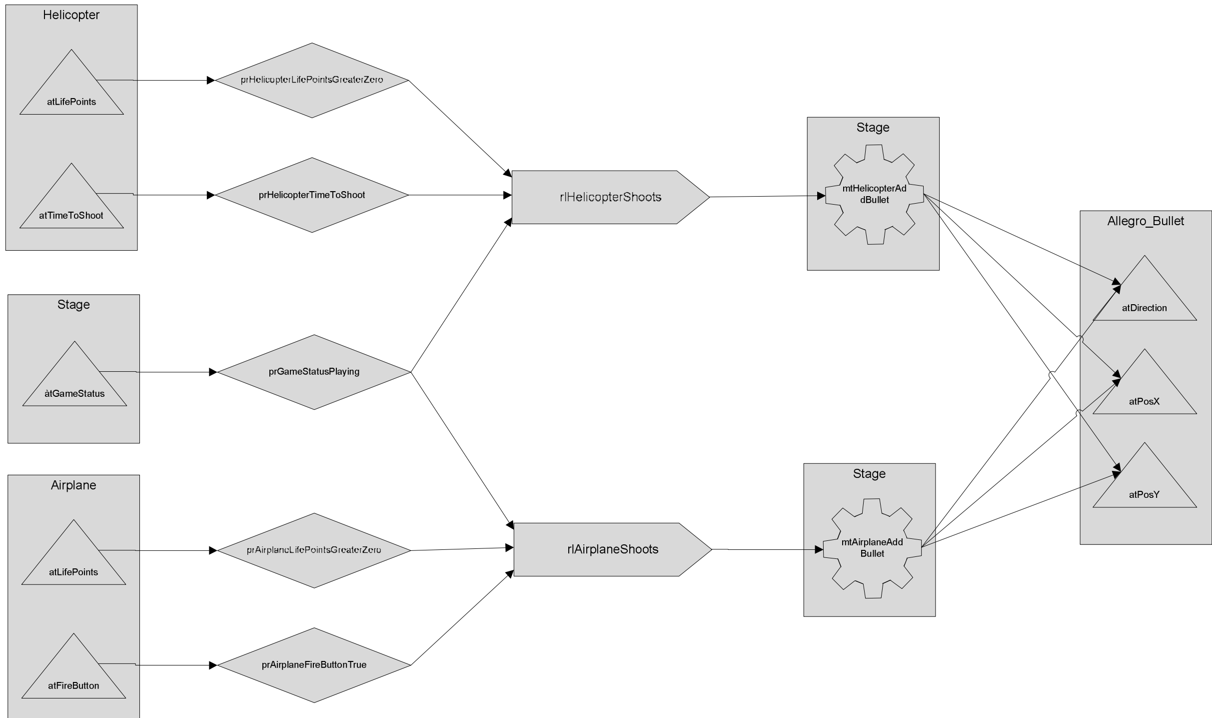
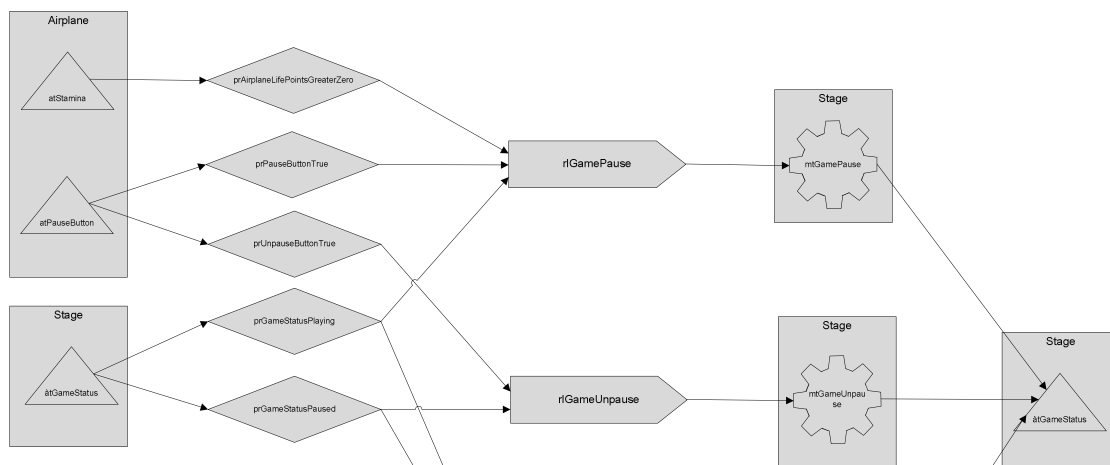


Figura 53 – *Attribute* alterado por mais de um *Method*

- Objetos que participam da realização de mais de um caso de uso

Certamente, a alcançabilidade mais impactante é a que altera inclusive *Attributes* de outros casos de uso (Figura 54). Por isso, requer a maior atenção, pois alterações mal projetadas podem prejudicar o sistema como um todo.

Caso de uso Pausar Jogo



Caso de uso Parar Jogo

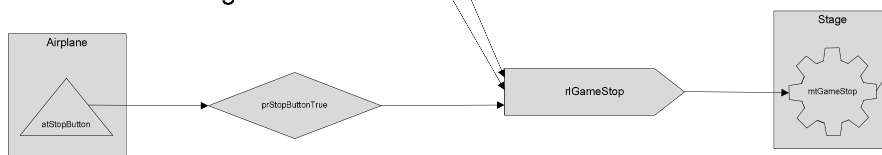


Figura 54 – Objetos que participam da implementação de vários casos de uso

Foi possível perceber que o desenvolvimento em PON exige muitos cuidados no tocante às conexões (e também possíveis conexões) entre objetos.

APÊNDICE G – Outras informações e técnica experimental para o teste de software em PON

Seção 1: ALCANÇABILIDADE NO PON

Este apêndice apresenta conteúdo experimental e ainda empírico a respeito de outros assuntos relacionados ao paradigma PON.

A alcançabilidade do ciclo de notificações do PON refere-se ao número de entidades que podem ser afetadas a partir da alteração do valor de um *Attribute*. Para demonstrar isto, considere as entidades que constituem o caso de uso “Controlar Avião” e a realização de cinco passos: 1) Investigar os *Attributes*; 2) Investigar quais *Premisses* avaliam o estado dos *Attributes* sobteste; 3) Investigar quais *Conditions* avaliam o estado das *Premisses* sobteste; 4) Investigar outras *Premisses* avaliadas pelas *Conditions* das *Rules* sob teste; e 5) Investigar todo o ciclo de notificações das entidades sob teste. A Figura 55 apresenta uma representação da técnica apresenta. Ressalta-se que esta técnica ainda é uma proposta para a obtenção de toda a rede de notificações pertinente a alteração de valor de um *Attribute*.

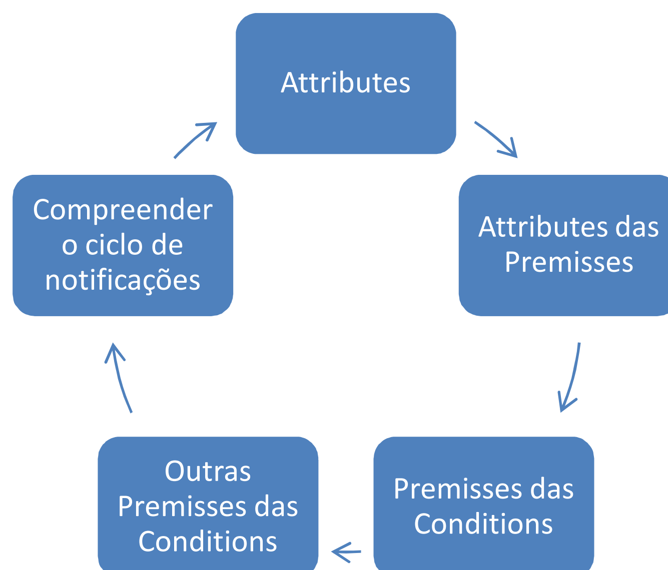


Figura 55 – Representação esquemática para obtenção da rede de alcançabilidade PON

Passo 1) Investigar os *Attributes*

Considere os *Attributes* que são alterados pelo *Method* mtMoveLeft do *FBE* Airplane apresentado no Algoritmo 6 (linha 3). Este *Method* executa duas linhas de instrução que decrementam em um o valor dos *Attributes* atPosX1 e atPosX2 (movem o avião ligeiramente à esquerda da posição em que se encontra). Esses *Attributes* podem fazer parte de uma ou mais *Premisses*, e por isso devem ser verificadas suas listas de *Premisses* inscritas para notificação.

```
1   Airplane::Airplane():Allegro_Character(){
2   ...
3   void Airplane::moveLeft(){
4    atPosX1->setValue(atPosX1->getValue()-1);
5    atPosX2->setValue(atPosX2->getValue()-1);
6   }
7   ...
8   };
```

Algoritmo 6 – Código fonte do *Method* moveLeft pertencente ao *FBE* Airplane

Passo 2) Investigar quais *Premisses* avaliam o estado dos *Attributes* sob teste

A investigação para identificar quais *Premisses* utilizam os *Attributes* sob teste leva em consideração a lista de inscrição de *Premisses* a serem notificadas pelos *Attributes*. As duas *Premisses* notificadas por esses *Attributes* são prAirplaneLimLeftScreen (Algoritmo 7) e prAirplaneLimRightScreen (Algoritmo 8).

***Premise* –** prAirplaneLimLeftScreen		
FBE	Airplane	
Attribute	Integer	atPosX1
Operador	*Greater or Equal*	
Valor	0	

Algoritmo 7 – Código fonte da *Premise* prAirplaneLimLeftScreen

***Premise* –** prAirplaneLimRightScreen		
FBE	Airplane	
Attribute	Integer	atPosX2
Operador	*Smaller or Equal*	
Valor	800	

Algoritmo 8 – Código fonte da *Premise* prAirplaneLimRightScreen

A *Premise* prAirplaneLimLeftScreen requer que o valor de atPosX1 seja superior a zero. Por sua vez, a *Premise* prAirplaneLimRightScreen requer que o valor de atPosX2 seja inferior a 800. Essas *Premisses* podem notificar uma ou mais *Conditions*.

Passo 3) Investigar quais *Conditions* utilizam as *Premisses* sob teste

A continuação da investigação para conhecer o alcance do ciclo de notificações da aplicação PON considera a lista de inscrição de *Conditions* a serem notificadas pelas *Premisses* avaliadas. É possível perceber que as *Rules* rlAirplaneMovingLeft (Algoritmo 9) e rlAirplaneMovingRight (Algoritmo 10), respectivamente, avaliam, em suas *Conditions,* as *Premisses* prAirplaneLimLeftScreen e prAirplaneLimRightScreen e, ainda, essas *Conditions* possuem outras *Premisses* (prGameStatusPlaying, prAirplaneLifePointsGreaterZero, prAirplaneLeftButtonTrue e prAirplaneRightButtonTrue) e, portanto, também requerem investigação.

<table>
<tr><td colspan="3">Rule – rlAirplaneMovingLeft</td></tr>
<tr><td rowspan="4">Condition –
Operador Conjunção</td><td rowspan="4">Condition –
Operador conjunção</td><td>prGameStatusPlaying</td></tr>
<tr><td>prAirplaneLifePointsGreaterZero</td></tr>
<tr><td>prAirplaneLeftButtonTrue</td></tr>
<tr><td>prAirplaneLimLeftScreen</td></tr>
<tr><td>Method</td><td colspan="2">Airplane mtMoveLeft</td></tr>
</table>

Algoritmo 9 – Código fonte da *Rule* rlAirplaneMovingLeft

A *Condition* da *Rule* rlAirplaneMovingLeft requer uma conjunção de quatro *Premisses* (prAirplaneLimLeftScreen, prGameStatusPlaying, prAirplaneLifePointsGreaterZero e prAirplaneLeftButtonTrue) para executar o *Method* mtMoveLeft do *FBE* Airplane. A *Condition* da *Rule* rlAirplaneMovingRight requer uma conjunção de quatro *Premisses* (prAirplaneLimRightScreen, prGameStatusPlaying, prAirplaneLifePointsGreaterZero e prAirplaneRightButtonTrue) para executar o *Method* mtMoveRight do *FBE* Airplane. Quando esta etapa for concluída, deve-se conhecer as outras *Premisses* que as *Conditions* destas *Rules* utilizam.

Rule – rlAirplaneMovingRight		
***Condition –* Operador Conjunção**	*Condition –* Operador conjunção	prGameStatusPlaying
		prAirplaneLifePointsGreaterZero
		prAirplaneRightButtonTrue
		prAirplaneLimRightScreen
Method	Airplane mtMoveRight	

Algoritmo 10 – Código fonte da *Rule* rlAirplaneMovingLeft

Passo 4) Investigar outras *Premisses* das *Conditions* avaliadas nas *Rules* sob teste

As duas *Rules* sob teste, rlAirplaneMovingLeft e rlAirplaneMovingRight, compartilham duas *Premisses*: prGameStatusPlaying (Algoritmo 11) e prAirplaneLifePointsGreaterZero (Algoritmo 12).

Premise – prGameStatusPlaying		
FBE	Stage	
Attribute	Integer	atGameStatus
Operador	*Equal*	
Valor	2	

Algoritmo 11 – Código fonte da *Premise* prGameStatusPlaying

Premise – prAirplaneLifePointsGreaterZero		
FBE	Airplane	
Attribute	Integer	atLifePoints
Operador	*Greater Than*	
Valor	0	

Algoritmo 12 – Código fonte da *Premise* prAirplaneLifePointsGreaterZero

A *Premise* prAirplaneLeftButtonTrue (Algoritmo 13) avalia o *Attribute* atLeftButton (requer que o botão esquerdo esteja com estado verdadeiro). Por sua vez, a *Premise* prAirplaneRightButtonTrue (Algoritmo 14) avalia o *Attribute* atRightButton (requer que o botão direito esteja com estado verdadeiro). Por fim, essas duas *Premisses* não são utilizadas por outras *Conditions* no sistema.

Premise – prAirplaneLeftButtonTrue		
FBE	Airplane	
Attribute	Boolean	atLeftButton
Operador	*Equal*	
Valor	True	

Algoritmo 13 – Código fonte da *Premise* prAirplaneLeftButtonTrue

Premise – prAirplaneRightButtonTrue		
FBE	Airplane	
Attribute	Boolean	atRightButton
Operador	*Equal*	
Valor	True	

Algoritmo 14 – Código fonte da *Premise* prAirplaneRightButtonTrue

Passo 5) Investigar toda a rede de notificações dos elementos sob teste

A última consiste em compreender sobre todo o ciclo de notificações que pode ocorrer com esses elementos envolvidos. A Tabela 135 apresenta as *Premisses* inscritas na lista de notificação dos *Attributes* investigados. A Tabela 136 apresenta as *Conditions* inscritas na lista de notificação das *Premisses* investigadas.

Tabela 135 – *Attributes* e suas lista de inscrições de *Premisses*

FBE	***Attribute***	***Premise***
Airplane	atLeftButton	prAirplaneRightButtonTrue
Airplane	atRightButton	prAirplaneLeftButtonTrue
Stage	atGameStatus	prGameStatusPlaying
Airplane	atLifePoints	prAirplaneLifePointsGreaterZero
Airplane	atPosX1	prAirplaneLimLeftScreen
Airplane	atPosX2	prAirplaneLimRightScreen

Tabela 136 – *Premisses* e suas listas de inscrições de *Conditions*

Premise	***Conditions* de *Rules***
prGameStatusPlaying	rlAirplaneMovingLeft rlAirplaneMoving rlAirplaneMovingRight
prAirplaneLimLeftScreen	rlAirplaneMovingLeft
prAirplaneLifePointsGreaterZero	rlAirplaneMovingLeft rlAirplaneMoving rlAirplaneMovingRight
prAirplaneLeftButtonTrue	rlAirplaneMovingLeft
prAirplaneLeftButtonFalse	rlAirplaneMoving
prAirplaneRightButtonTrue	rlAirplaneMovingRight
prAirplaneRightButtonFalse	rlAirplaneMoving
prAirplaneLimRightScreen	rlAirplaneMovingRight

A Tabela 137 apresenta a lista de *Methods* executados pelas *Rules* investigadas. A Tabela 138 apresenta os *Attributes* que tem seus valores modificados pelas execuções dos *Methods* e as listas de *Premisses* inscritas para serem notificadas por esses *Attributes*.

Tabela 137 - *Rules* e suas listas de execução de *Methods*

Rule	*Method*
rlAirplaneMovingRight	mtMoveRight
rlAirplaneMovingLeft	mtMoveLeft

Tabela 138 – *Attributes* alterados pelos *Methods* e o impacto nas *Premisses*

Method	*Attribute*	Valor do *Attribute*	*Premise*
mtMoveLeft	atPosX1	atPosX1= atPosX1-1	prAirplaneLimLeftScreen
	atPosX2	atPosX2= atPosX2-1	prAirplaneLimRightScreen
mtMoveRight	atPosX1	atPosX1= atPosX1+1	prAirplaneLimLeftScreen
	atPosX2	atPosX2= atPosX2+1	prAirplaneLimRightScreen

Seção 2) DENOMINAÇÕES COMUNS DO TESTE ESTRUTURAL ADAPTADAS PARA O PON

- Bloco de instrução

Um bloco de instrução são as entidades PON que possuem estado, escopo limitado e podem influenciar o comportamento de outras entidades. *Attribute*, *Premise*, *Condition*, *Rule* e *Method* são blocos de instrução do PON.

- Caminho

Consiste em acompanhar todo o ciclo de notificações do PON a partir da alteração do valor de um ou mais *Attributes.* Por exemplo, a alteração do valor de um *Attribute* que pode provocar a aprovação de uma *Premise*. Esta *Premise* pode provocar a aprovação de uma *Condition* que provoca a aprovação de sua *Rule* e consequentemente a execução de um *Method*, que não altera ou cria outros elementos.

- Uso computacional

O uso computacional é relacionado à atribuição de valor a um *Attribute*. Este uso é essencial para iniciar todo o ciclo de notificações. Por exemplo, o Algoritmo 15 apresenta um exemplo de atribuição de valor 2 (Playing) ao *Attribute* atGameStatusPlaying. Tipicamente, esta atividade é realizada por um *Method*.

```
atGameStatusPlaying ← 2
```

Algoritmo 15 – Exemplo de uso computacional em código PON

- Uso predicativo

O uso predicativo é caracterizado pela existência de uma expressão condicional de uma entidade PON avaliando uma ou mais entidades. Em PON, a declaração de uma *Premise*, *Condition* ou *Rule* representa uso predicativo. Por exemplo, a Tabela 139 apresenta a estrutura da *Premise* prGameStatusPlaying que avalia o *Attribute* atGameStatus. O valor deste *Attribute* deve ser igual a 2

(PLAYING) para que a *Premise* seja aprovada. Isso caracteriza uso predicativo de apenas um *Attribute*.

Tabela 139 - Exemplo de uso predicativo em *Premise*

Premise	*Attribute*	Operador	Valor
prGameStatusPlaying	**atGameStatus**	**=**	**2 (Playing)**

Para a declaração de *Rule* são requeridas definições de *Premise (s)*, *Condition(s)* e *Method(s)* a ela relacionada. Por exemplo, na Tabela 140, a *Rule* rlAirplaneShoots contém apenas uma *Condition* que requer uma conjunção de quatro *Premisses*: prGameStatusPlaying, prAirplaneLifePointsGreaterZero, prAirplaneHasBullet e prAirplaneFireButtonTrue. Assim, o uso predicativo fica caracterizado sobre a *Condition* que avalia o estado destas três *Premisses* e também sobre a *Rule* que avalia o estado da *Condition*.

Tabela 140 - Exemplo de uso predicativo em *Rule*

Rule	*Condition* – **Conjunction**	*Method*
rlAirplaneShoots	**prGameStatusPlaying** **prAirplaneLifePointsGreaterZero** **prAirplaneFireButtonTrue** **prAirplaneHasBullet**	mtAirplaneAddNewBullet

Seção 3) DETECÇÃO DE ENTIDADES REDUNDANTES OU DUPLICADAS

As entidades *Premise*, *Condition* e *Rule* avaliam os estados de outras entidades PON e reagem de acordo com esses estados. Quando uma entidade avalia a instância e o estado dos mesmos elementos que são avaliados por outra entidade e ambas as entidades não avaliam outros elementos fica caracterizada a redundância de entidades. Entidades redundantes no PON podem causar inconsistências na aplicação, além de consumir processamento desnecessário.

Alguns conceitos da teoria dos conjuntos (POTTER, 2004), adaptados ao PON, apresentam uma forma intuitiva e útil de agrupar objetos que pode ser utilizada para identificação de entidades redundantes. Para isso, cada círculo (grupo) pode representar uma entidade e seus elementos que a compõe. A intersecção de duas entidades permite visualizar os elementos que são utilizados por mais de uma entidade.

Um exemplo de redundância de entidades é apresentado na Figura 56. É possível notar que a *Premise* A e a *Premise* B compartilham os mesmos estados dos mesmos *Attributes* avaliados, caracterizando redundância. Por sua vez, a *Premise* C avalia dois *Attributes* sendo que apenas um é compartilhado com a *Premise* D, que não caracteriza redundância de *Premisses*. Considere que os três elementos que compõem uma *Premise* (Referência, Operador e Valor), conforme visto na Figura 3 (Seção 2.2.1), foram simplificados recebendo nomenclatura reduzida (v.g. *Attribute* X) para ser apresentado na exemplificação.

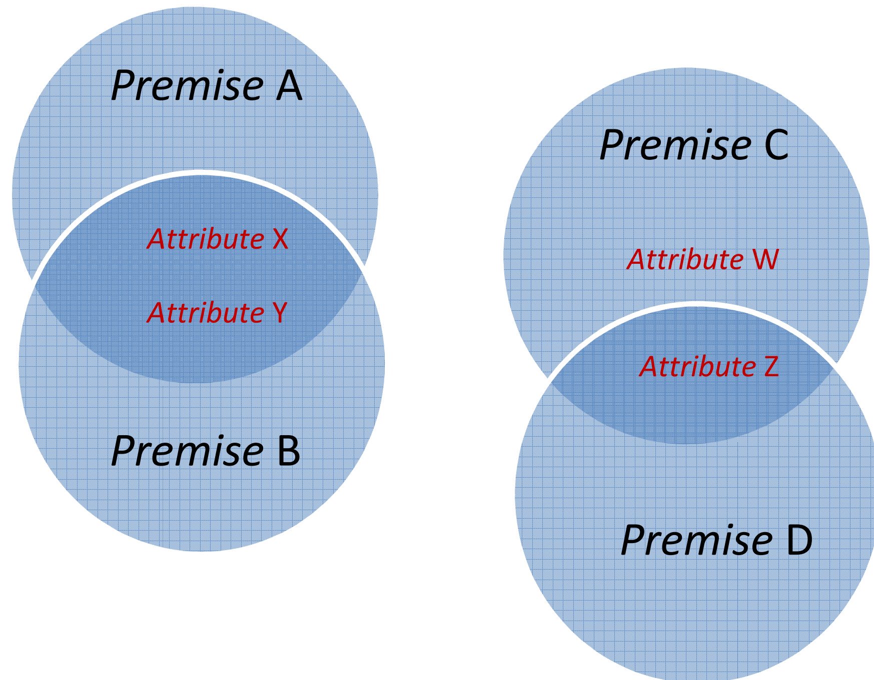


Figura 56 – Interseção para visualização de *Premisses* redundantes.
***Premise* A e *Premise* B são redundantes**

De modo análogo à *Premise*, a detecção de *Conditions* e *Rules* redundantes também pode ser feita por meio da interseção com outras entidades. *Condition*, por meio de operadores lógicos, avalia o estado de uma ou mais *Premisses* ou mais *Attributes* por meio de operadores lógicos. *Rule* avalia o estado de sua *Condition*. Por sua vez, *Attribute* e *Method*, a priori, não oferecem condições para que seja verificada sua duplicação.