Método para Desenvolvimento de Sistemas Orientados a Regras utilizando o Paradigma Orientado a Notificações

I. T. M. Mendonça, J. M. Simão, L. V. B. Wiecheteck, P. C. Stadzisz
Pós-Graduação em Engenharia Elétrica e Informática Industrial
Universidade Tecnológica Federal do Paraná
Curitiba, Paraná, Brasil

Resumo* — Sistemas de inteligência artificial são frequentemente desenvolvidos empregando o paradigma de programação declarativo, na forma de sistemas baseados em regras, notadamente. Mais recentemente, propôs-se um novo paradigma orientado a regras e notificações, denominado PON. Entretanto, o PON carece de métodos de Engenharia de Software que considerem suas particularidades no desenvolvido de software. Este trabalho apresenta uma proposta de método, chamado Desenvolvimento Orientado a Notificações (DON), criado para guiar o desenvolvimento de software seguindo o PON. O método DON é composto de oito atividades, nas quais o projeto do software é concebido e descrito por artefatos da UML e Redes de Petri. O método é avaliado por meio de um estudo de caso em que um jogo é modelado e os resultados demonstram que os artefatos criados são suficientes para descrever a estrutura e o comportamento do software para o PON.

Palavras-chave— Sistemas Baseados em Regras, Paradigma Orientado a Notificações, Projeto de Software, Modelagem de Software, Processo de Software, Desenvolvimento Orientado a Notificações.

I. INTRODUÇÃO

O desenvolvimento de sistemas de Inteligência Artificial e Inteligência Computacional, notadamente sistemas especialistas e *fuzzy*, utilizam, com frequência, Sistemas Orientados a Regras (SOR) [1]. Tradicionalmente em SOR são empregados Sistemas Baseados em Regras (SBR) que usam motores de inferência para associar fatos e regras e cadenciar a execução dos sistemas. Como alternativa, no contexto de SOR, o Paradigma Orientado a Notificações (PON) aparece como resposta, tanto para sistemas *crisp* quanto *fuzzy*, para superar algumas deficiências encontradas em SBRs e, mesmo, em outros paradigmas de programação atuais [2]. Entre estas deficiências, está, principalmente, a necessidade de motores de inferência para conduzir o fluxo lógico do software [3].

O PON é baseado em entidades pequenas, inteligentes e desacopladas que colaboram por meio de notificações precisas para realizar a inferência de *software*. Isto permite melhorar o desempenho do *software* e, potencialmente, torna mais fácil sua composição, tanto dos distribuídos quanto dos não-distribuídos [4]. No contexto da *Engenharia de Software* (ES) foi proposto um método para projetos de *software* que usam o PON em seu desenvolvimento. O método, chamado Desenvolvimento Orientado a Notificações (DON), abrange as fases de requisitos e projeto de *software* PON [5], [6].

II. PARADIGMA ORIENTADO A NOTIFICAÇÕES (PON)

As entidades que compõem o PON são ilustradas na Fig. 1, por meio de seu metamodelo. Diferente de outras abordagens nas quais é necessário um motor de inferência [3], no PON a inferência é realizada pela notificação pontual entre as entidades da aplicação. No PON as entidades possuem certa reatividade, permitindo que elas, ao detectar a modificação no seu estado, notifiquem outras entidades que têm interesse nessa informação [4].

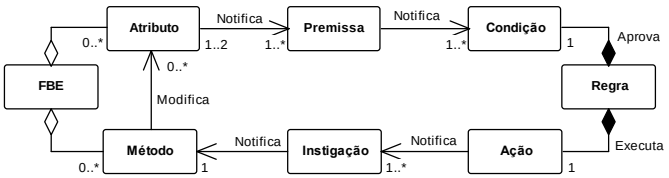

Fig. 1. Metamodelo do PON

O Elemento da Base de Fatos (*Fact Base Element* – FBE) armazena fatos do sistema por meio de *Atributos*. O FBE pode possuir *Métodos* que modificam esses *Atributos*. A *Premissa* é uma expressão causal relacionada a um *Atributo* que, quando notificada por ele, verifica se ele possui determinado valor (p.e. *AtributoStatus == "Ligado"*). A *Condição* possui uma ou mais *Premissas* associadas que, quando verdadeiras, aprovam uma *Regra*. A *Regra* realiza alguma ação quando é aprovada e, para isso, possui um conjunto de *Ações*. Cada *Ação* irá notificar um conjunto de *Instigações* que, por sua vez, irá instigar um ou mais *Métodos*, modificando um ou mais *Atributos*. Esta sequência constitui o mecanismo de inferência do PON. Desse modo, aplicações em PON não dependem de um elemento centralizador, pois não necessitam de um motor de inferência.

Essa nova lógica de concepção de sistemas ou *softwares* faz com que os métodos atuais de ES sejam insuficientes na concepção de aplicações no PON. A seguir é apresentado um método, baseado nos atuais processos de *software*, que os estende para adequá-los ao PON.

III. DESENVOLVIMENTO ORIENTADO A NOTIFICAÇÕES (DON)

O DON foi a primeira iniciativa para definir um método de ES que pudesse guiar o desenvolvimento de aplicações usando o PON. Desenvolvido no trabalho de mestrado de Wiecheteck [6], o DON se ocupa em: (i) estender diagramas da UML para representar os conceitos do PON, (ii) definir um método que

use essa extensão em uma sequência de passos para conduzir o desenvolvedor na criação de aplicações PON.

O primeiro passo foi criar um perfil UML, denominado Perfil PON, usando mecanismos de extensão da UML. O Perfil PON permite que a UML seja usada na representação dos principais conceitos do PON. O perfil fundamenta-se nos conceitos do paradigma e "principalmente" nos modelos de classes empregados no desenvolvimento do *framework* PON.

Um perfil UML permite que a UML seja particularizada para um domínio específico de aplicações, possibilitando determinar uma nova sintaxe e semântica aos elementos existentes na UML. Para isso, faz uso de estereótipos, valores etiquetados e restrições. Os detalhes do Perfil PON estão disponíveis em [5], [6].

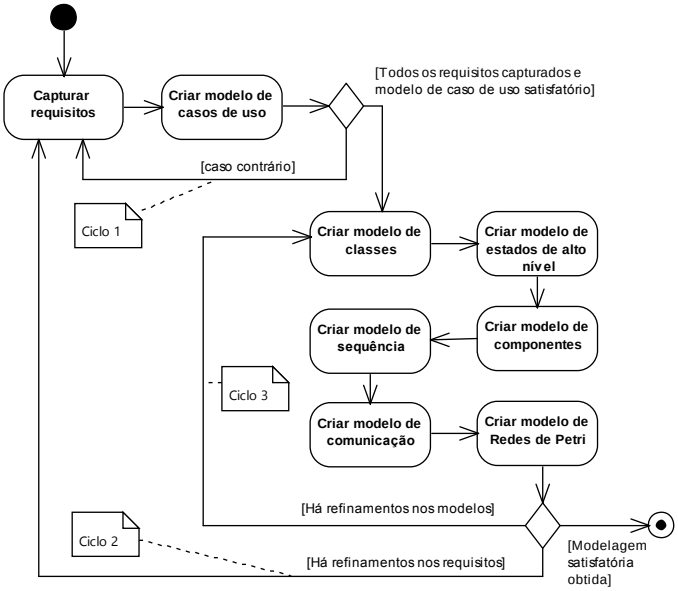

Fig. 2. Atividades e ciclos do DON

O DON abrange atividades das fases de Requisitos e Análise e Projeto para sistemas baseados no PON. Todavia, o método é aderente aos processos de *software* tradicionais e deve ser usado em conjunto com eles. Assim, é organizado em oito passos, cujos dois primeiros são usados na captura de requisitos do sistema e os outros seis são focados na criação de modelos para representar o sistema. Os modelos desenvolvidos no método são: modelo de classes, modelo de estados de alto nível, modelo de componentes, modelo de sequência, modelo de comunicação e modelo de Redes de Petri [6]. O diagrama de atividades da Fig. 2 ilustra os oito passos do método DON.

O método é dividido em três ciclos básicos. O primeiro compreende as atividades de levantamento de requisitos do *software* e é finalizado quando todos os requisitos foram capturados e tem-se um modelo de casos de uso satisfatório. O segundo ciclo cria as primeiras versões dos diagramas e retorna aos requisitos enquanto for necessário fazer refinamentos. Quando não houver mais refinamentos, o terceiro ciclo se concentra em finalizar os modelos até que os requisitos do *software* sejam atendidos. A codificação poderá ocorrer em ciclos intermediários, se o processo de *software* usado for iterativo e incremental, ou poderá iniciar ao final da aplicação do DON, caso o processo seja baseado no modelo de processo de *software* em cascata.

A seguir são apresentados os oito passos do DON aplicados na modelagem de um jogo hipotético em que o jogador controla um navio de guerra atirando contra um avião de guerra inimigo.

A. Capturar requisitos e Criar modelo de casos de uso

As duas primeiras atividades do DON não diferem de um processo usual de desenvolvimento de *software*. Nessas atividades, desenvolvidas inicialmente no ciclo 1, são levantados os requisitos para o *software*, criando uma lista de requisitos e o modelo de casos de uso seguindo o diagrama usual da UML. A Tabela I ilustra o artefato criado na atividade Capturar requisitos e a Fig. 3 ilustra o modelo de casos de uso. Adicionalmente, elabora-se, para o modelo de casos de uso, a descrição de cada caso de uso, bem como a qual requisito ele atende.

TABELA I. ARTEFATO CRIADO NO PASSO CAPTURAR REQUISITOS

#	**Requisitos não funcionais**
RNF1	O sistema deve oferecer um menu principal com duas opções: *Iniciar um novo jogo* e *Sair*.
RNF2	O sistema deve mostrar o navio de guerra na parte inferior da tela e o avião de guerra na parte superior da tela.
RNF3	O sistema deve limitar a movimentação do navio de guerra somente na parte visível do jogo.
RNF4	A tela do sistema deve ter o tamanho de 800x600 pixels.
RNF5	O sistema deve mostrar a quantidade de estamina do jogador e do avião de guerra.
#	**Requisitos funcionais**
RF1	O sistema deve mover o navio de guerra para a esquerda quando a seta para a esquerda for pressionada
RF2	O sistema deve mover o navio de guerra para a direita quando a seta para a direita for pressionada
RF3	O sistema deve fazer o navio de guerra atirar quando o botão de disparo for pressionado.
RF4	O sistema deve mover o inimigo (avião de guerra) para a esquerda e depois para a direita.
RF5	O sistema deve fazer o inimigo atirar a cada intervalo de dois segundos.
RF6	O sistema deve diminuir a estamina do inimigo quando ele for atingido pelo projétil do jogador.
RF7	O sistema deve diminuir a estamina do jogador quando ele for atingido pelo projétil do inimigo.
RF8	O sistema deve explodir o navio de guerra quando a estamina do jogador acabar e mostrar a mensagem "Você perdeu!", também deverá mostrar uma opção para voltar ao menu inicial.
RF9	O sistema deve explodir o avião de guerra quando a estamina do inimigo acabar e mostrar a mensagem "Você ganhou!", também deverá mostrar uma opção para voltar ao menu inicial.
RF10	O sistema deve permitir que o jogador pause o jogo e volte a jogar.
RF11	O sistema deve sair do jogo quando o botão de sair for pressionado.

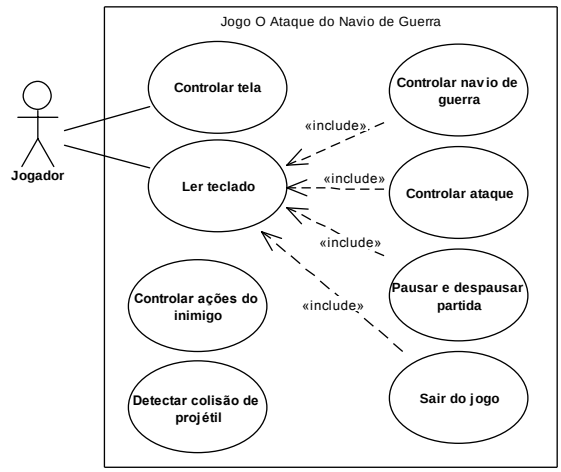

Fig. 3. Modelo de casos de uso

B. Criar modelo de classes

O modelo de classes começa a ser criado no ciclo 2 e é refinado nesse ciclo e no ciclo 3. O perfil PON começa a ser usado a partir desse modelo. O modelo de classes é o usual da UML, porém o uso de estereótipos associa as classes aos elementos do metamodelo do PON. Os atributos e métodos do PON compõem os FBEs e, assim, são definidos como atributos e métodos das classes FBE. A Fig. 4 ilustra o modelo de classes criado para o exemplo estudado.

A primeira classe criada nesse modelo representa a aplicação PON (Fig. 4, elemento estereotipado com *<<NOP_Application>>*). Essa classe é responsável por instanciar os objetos FBEs definidos pelas outras classes. As classes FBEs são identificadas pelo projetista a partir da análise dos Requisitos e Casos de uso (Fig. 4, elementos estereotipados com *<<NOP_FBE>>*) e incluem os Atributos e Métodos do FBE, conforme indicado pelos estereótipos de atributo e método *<<NOP_Attribute>>* e *<<NOP_MethodPointer>>*. O uso do Perfil PON permite identificar os elementos do PON com clareza.

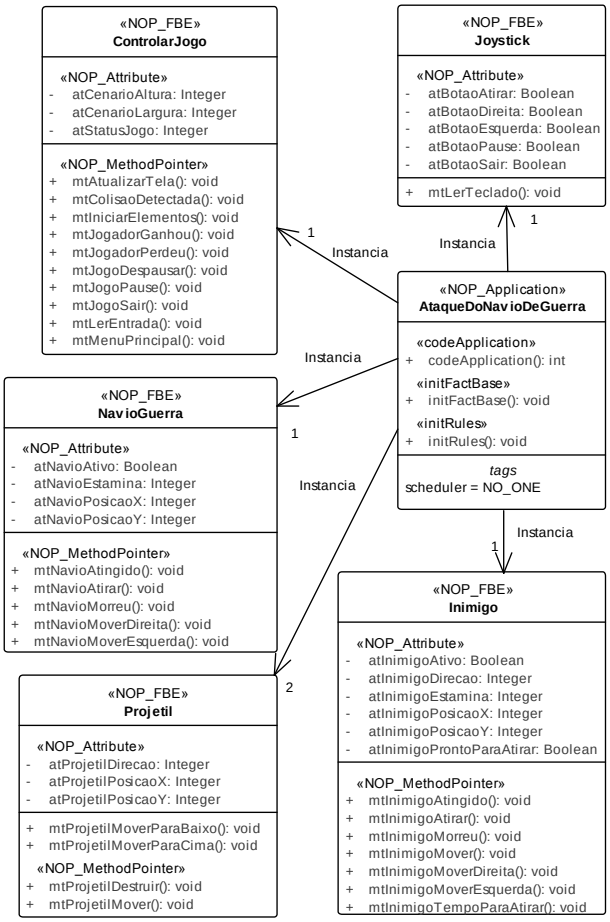

Fig. 4. Modelo de classes do jogo

O FBE *Joystick* foi identificado pela análise do caso de uso *Ler teclado*. Esse FBE é responsável por capturar as ações do jogador agindo como um controle real. Os FBEs *NavioGuerra*, *Inimigo* e *Projetil* foram identificados por serem elementos do jogo e estarem presentes tanto nos requisitos quanto nos casos de uso *Controlar Navio de Guerra*, *Controlar ataque*, *Controlar ações do inimigo* e *Detectar colisão de projétil*. O jogador controla o FBE *NavioGuerra* enquanto que os outros dois são autocontrolados. O FBE *ControlarJogo* foi identificado pela análise do casos de uso *Controlar tela* e *Pausar e despausar partida*. Este FBE atualiza os elementos na tela e é responsável por mecanismos adicionais de controle do jogo.

Os FBEs e seus atributos dão uma visão geral dos elementos do jogo. Os métodos mostram as ações que podem ser executadas pelos FBEs. Esses elementos são identificados à medida que as iterações vão ocorrendo. Por exemplo, o caso de uso *Pausar e despausar partida* sugere que o jogo pode estar pausado ou jogando. Então, o FBE *ControlarJogo* deverá ter um atributo para representar esses estados. Um elemento se movimentando na tela vai necessitar de atributos para indicar a sua posição e assim por diante.

C. Modelo de estados de alto nível

Neste passo do DON a lógica do sistema começa a ser desenhada. Em PON, o encadeamento lógico é definido pelas regras que compõem o sistema. O modelo deste passo irá fornecer uma visão geral da dinâmica do *software*, auxiliando na identificação das regras necessárias.

O diagrama de estados de alto nível criado neste passo poderá ser um diagrama de atividades ou um diagrama de máquina de estados, ambos da UML. O primeiro passo é criar um diagrama de estados para cada classe de FBE identificado. Cada diagrama irá mostrar a mudança de estados no âmbito de cada classe e essas mudanças podem ser vistas como regras do sistema. Por exemplo, quando o método *mtNavioAtingido* da classe *NavioGuerra* for executado, ele irá diminuir a estamina do jogador. Essa última frase corresponderá a uma regra do sistema.

A próxima etapa é agrupar os diagramas, resultando no modelo de estados de alto nível, que facilita a visualização da interação e relação entre os estados das classes. Por exemplo, mudar a posição do Navio de guerra depende das classes *Joystick*, *NavioGuerra* e *ControlarJogo*. A classe *Joystick* age capturando a entrada do jogador, a classe *NavioGuerra* age modificando a posição do Navio de guerra e a classe *ControlarJogo* age atualizando a tela para mostrar o Navio de guerra em sua nova posição. Além disso, o diagrama de estados de alto nível fornece uma visão geral do sistema.

A Fig. 5 ilustra o modelo de estados de alto nível após algumas iterações. Esse modelo pode ser melhorado com a inclusão de detalhes, porém este nível é apropriado para ilustrar alguns conceitos. Nesse modelo, após o passo de inicialização, o *software* pode iniciar uma nova partida do jogo ou ser fechado. Caso inicie uma nova partida, cinco atividades ocorrerão simultaneamente. Uma delas mostra os elementos do jogo na tela e leva o jogo ao estado de *Tela atualizada*. Esse estado é atualizado constantemente enquanto o botão de sair não for pressionado. Outra atividade executada é a leitura da entrada do jogador, que leva ao estado de *Entrada do jogador lida*. Após esse estado, o *software* irá executar a operação solicitada pelo usuário e retornar à leitura de entrada do jogador. As outras três atividades são similares à primeira.

O modelo criado neste passo mostra a sequência de estados dos casos de uso e, por ser unificado, as situações em que esses casos de uso se relacionam. O caso de uso *Controlar ataque*, por exemplo, pode ser visto seguindo os estados: *Aplicação inicializada*, *Menu principal exibido*, *Jogo iniciado*, *Entrada do jogador lida* e *Navio atirou*.

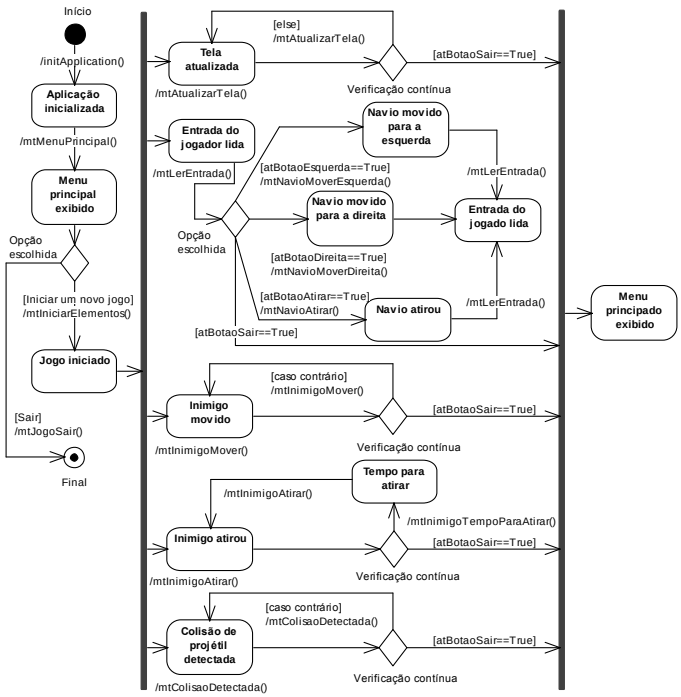

Fig. 5. Modelo de estados de alto nível

Este diagrama auxilia na descoberta de regras do sistema. Porém, como no processo de identificação de classes no Paradigma Orientado a Objetos, a descoberta de regras é uma atividade de síntese criativa. Então, são usados os elementos *<<NOP_FBE>>* do modelo de classes e o modelo de estados de alto nível como instrumento de ajuda.

Uma vez que uma regra seja identificada, deve-se responder algumas questões para ajudar no processo:

1) Qual o objetivo da Regra?
2) O que precisa acontecer para a Regra ser executada?
3) O que acontece se a Regra for executada?

O processo de análise dos modelos de classes e de estados de alto nível gerou 16 regras no exemplo considerado. A Tabela II ilustra algumas regras com as respostas para as questões acima.

TABELA II. REGRAS IDENTIFICADAS (PARCIAL)

#	Qual o objetivo da Regra?	O que precisa acontecer para a Regra ser executada?	O que acontece se a Regra for executada?
1	Mostrar os objetos em suas posições	O *status* do jogo deve ser JOGANDO.	Os objetos são posicionados na tela nas posições indicadas em seus atributos.
3	Fazer o avião de guerra atirar	O *status* do jogo deve ser JOGANDO, a estamina do inimigo deve ser maior do que zero, o último tiro deve ter ocorrido ha dois segundos e a posição do inimigo deverá estar na área visível do jogo.	Um projétil deve ser criado e direcionado para baixo.
11	Pausar jogo	O *status* do jogo deve ser JOGANDO e o botão pause deve ter sido pressionado.	O *status* do jogo deve ser modificado para PAUSADO.
12	Despausar jogo	O *status* do jogo deve ser PAUSADO e o botão pause deve ter sido pressionado.	O *status* do jogo deve ser modificado para JOGANDO.

D. Criar modelo de componentes

O diagrama de componentes é usado de forma inovadora no DON para modelar estaticamente as regras do *software*. As regras identificadas no passo anterior serão, neste passo, concluídas. Elas receberão seus nomes, premissas e instigações. Neste modelo faz-se uso do Perfil PON [5].

A criação do modelo de componentes é realizada em três etapas: (i) definir as regras, (ii) definir premissas e instigações e (iii) relacionar as regras aos FBEs.

1) Definir as regras

As 16 regras identificadas recebem nomes, conforme exibido na Tabela III.

TABELA III. NOMEANDO REGRAS

Regra	Nome
1	rlAtualizarPosicaoObjetos
2	rlInimigoMover
3	rlInimigoAtirar
4	rlInimigoModificarDirecaoDireita
5	rlInimigoModificarDirecaoEsquerda
6	rlDetectarColisaoContraInimigo
7	rlDetectarColisaoContraJogador
8	rlNavioMoverEsquerda
9	rlNavioMoverDireita
10	rlNavioAtirar
11	rlJogoPausar
12	rlJogoDespausar
13	rlJogoParar
14	rlJogadorGanha
15	rlJogadorPerde
16	rlIniciarNovoJogo

Para cada regra deverão ser criados três componentes estereotipados com *<<NOP_Rule>>*, *<<NOP_Action>>* e *<<NOP_Premise>>* (usando o Perfil PON). Além disso, um componente estereotipado com *<<NOP_Rule>>* deverá ser criado para abranger os outros três. Os três componentes internos são então relacionados usando os estereótipos *<<RuleNotifiesAction>>* e *<<ConditionNotifiesRule>>*. Esses estereótipos definidos no Perfil PON determinam a semântica e as restrições entre os elementos. Além disso, o Perfil PON automaticamente inclui a *tag logicalOperator* no elemento *<<NOP_Condition>>*, a qual será definida posteriormente.

2) Definir premissas e instigações

As premissas e instigações são identificadas pela análise dos atributos e métodos no modelo de classes e regras do modelo de estados de alto nível. A Tabela IV exibe o resultado dessa análise para as regras relacionadas na Tabela II.

TABELA IV. PREMISSAS E INSTIGAÇÕES DAS REGRAS (PARCIAL)

Regras / Premissas	Instigações
Regra 1 – rlAtualizarPosicaoObjetos	
ControlarJogo.atStatusJogo==Jogando	ControlarJogo.mtAtualizarTela()
Regra 3 – rlInimigoAtirar	
ControlarJogo.atStatusJogo==Jogando & Inimigo.atInimigoAtivo==True & Projetil.atProjetilStatus==Inativo	Inimigo.mtInimigoAtirar()
Regra 11 – rlJogoPausar	
ControlarJogo.atStatusJogo==Jogando & Joystick.atBotaoPause==True	ControlarJogo.mtJogoPause()
Regra 12 – rlJogoDespausar	
ControlarJogo.atStatusJogo== Paused & Joystick.atBotaoPause==True	ControlarJogo.mtJogoDespausar()

Após a identificação das premissas e instigações, esses elementos poderão ser representados nos diagramas de componentes de algumas formas. Em um nível maior de abstração, as premissas e instigações podem ser apresentadas como interfaces de componentes. Em um nível mais baixo de abstração, eles podem ser apresentados como componentes. Neste trabalho optou-se por usar níveis maiores de abstração.

Outra atividade desse passo é a definição do operador lógico para a regra. O operador "*CONJUNCTION*" foi usado

para a regra *rlNavioAtirar* para indicar que todas as premissas são obrigatórias. Outros operadores disponíveis são "*DISJUNCTION*", usado quando somente uma premissa é suficiente para ativar a regra, e "*SINGLE*" quando há somente uma premissa.

3) Relacionar as regras aos FBEs

Nesta última etapa da criação do modelo de componentes, cada regra é conectada aos FBEs responsáveis pelos atributos que notificam as premissas e cujas instigações afetam seus métodos. A Fig. 6 mostra o diagrama de componentes da regra *rlNavioAtirar* após a inclusão de relação com os respectivos FBEs.

Os atributos dos FBEs (p.e. *atStatusJogo* do FBE *ControlarJogo*) são representados nesse diagrama por interfaces fornecidas. Os métodos são apresentados como interfaces requisitadas pois, diferente dos atributos, esses elementos são requisitados por outros componentes.

Os diagramas de componentes de cada regra são agrupados em um único diagrama para formar o modelo de componentes do *software*. Esse modelo oferece uma visão global e mais abstrata das ligações entre as regras e os outros elementos do metamodelo do PON.

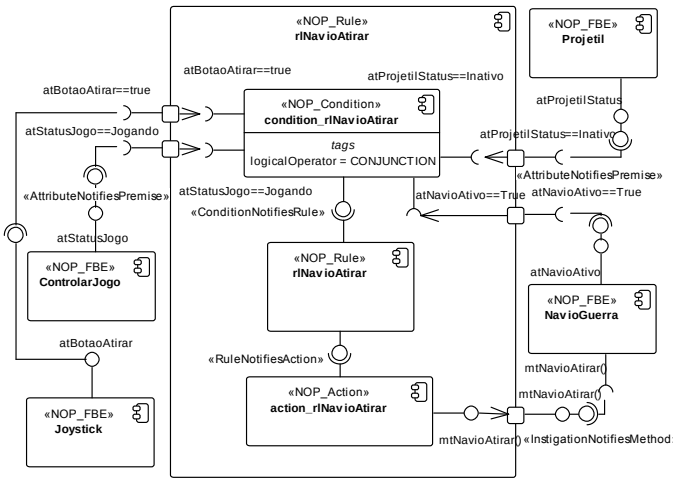

Fig. 6. Diagrama de componentes da regra rlNavioAtirar após etapa 3

E. Criar modelo de sequência e de comunicação

Os modelos criados nesse passo são usados para descrever as interações entre os elementos do metamodelo do PON. O nível de abstração dos diagramas nesses modelos pode variar de acordo com a importância do que se pretende modelar. Alguns dos diagramas de sequência podem envolver todos os elementos do *software*.

A Fig. 7 apresenta um diagrama de sequência para a regra *rlNavioAtirar*, que mostra a interação entre os FBEs *Joystick*, *NavioGuerra*, *Projetil* e *ControlarJogo* com essa regra. Esse diagrama poderia ter um detalhamento maior incluindo os objetos que correspondem às *Condições*, *Ações* e outros elementos do PON, porém para efeito de ilustração da comunicação entre os elementos, esse nível é adequado. O diagrama de comunicação correspondente a esse diagrama de sequência não será exibido neste artigo, pois apresenta informações similares ao diagrama de sequência, porém com foco na relação entre os objetos.

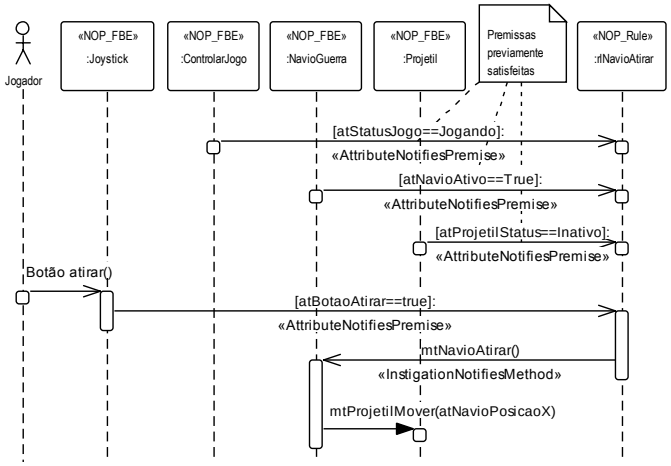

Fig 7. Diagrama de sequência de alto nível para a regra *rlNavioAtirar*

O diagrama de sequência exibido na Fig. 7 pode ser considerado um diagrama de alto nível de abstração. Nele, somente FBEs e regras são usados. Os outros elementos do metamodelo do PON ficam subentendidos. O diagrama mostra a regra *rlNavioAtirar* com três premissas previamente confirmadas por mensagens do diagrama. Essas premissas indicam que o *status* do jogo é Jogando, a estamina do jogador é maior do que zero e o *status* do projétil é inativo. Quando uma última premissa é confirmada pela ação do jogador que pressionou o botão de atirar, a regra é ativada e poderá ser executada. A execução da regra faz com que o método *mtNavioAtirar* seja executado, alterando o valor do *status* do projétil para ativo.

Os diagramas de sequência também podem ser utilizados para ilustrar uma sequência de regras sendo executadas, como por exemplo, a sequência de notificações da regra *rlNavioAtirar* e da regra *rlDetectarColisaoContraInimigo* quando a colisão é detectada.

Os diagramas criados nesse passo são importantes para entender e comparar a ordem de execução das regras, especialmente se mais de uma regra estiver no diagrama. O conjunto de diagramas compõe os modelos de sequência e comunicação do DON. Ademais, o uso do Perfil PON nesses modelos garante a execução correta da lógica do PON enquanto o processo de modelagem avança.

F. Criar modelo de Redes de Petri

O modelo de Redes de Petri é usado para demonstrar a dinâmica entre os elementos do PON. Na UML esta visão é usualmente criada com diagrama de estados para cada objeto. Porém, esta abordagem torna-se impraticável quando o número de estados do *software* é muito grande. Na prática, poucos projetos possuem toda a dinâmica modelada usando a UML.

As Redes de Petri (RdP) permitem a modelagem de concorrência, sincronização e compartilhamento de recursos em sistemas [7]. O DON sugere a criação de uma RdP para cada caso de uso. Na sequência, essas RdP são integradas em um único modelo de RdP.

O modelo de RdP é obtido a partir do mapeamento do modelo de componentes. O DON sugere duas alternativas de mapeamento, ambas de RdP Lugar/Transição, sendo uma detalhada e outra resumida. A seguir é apresentado o

mapeamento resumido, alternativa usada para modelar o *software* de exemplo deste trabalho.

1) Os componentes estereotipados *<<NOP_Rule>>* são Transições na RdP.
2) Os Atributos e seus estados pertencentes aos componentes *<<NOP_Premise>>* e *<<NOP_Instigation>>* são Lugares na RdP.
3) O componentes estereotipados *<<NOP_Premise>>* ou interfaces requeridas das regras são arcos chegando nas transições que representam os elementos *<<NOP_Rule>>*.
4) Os componentes estereotipados *<<NOP_Instigation>>* ou interface fornecidas são arcos que saem das transições que representam os elementos *<<NOP_Rule>>*.

Para ilustrar, foram mapeadas três regras. A primeira regra, *rlNavioAtirar*, refere-se ao caso de uso *Controlar ataque*. As outras, *rlJogoPausar* e *rlJogoDespausar*, pertencem ao caso de uso *Pausar e despausar partida*. A Fig. 8 mostra um passo intermediário em que a RdP dos casos de uso *Controlar ataque* e *Pausar e despausar partida* foram agrupadas.

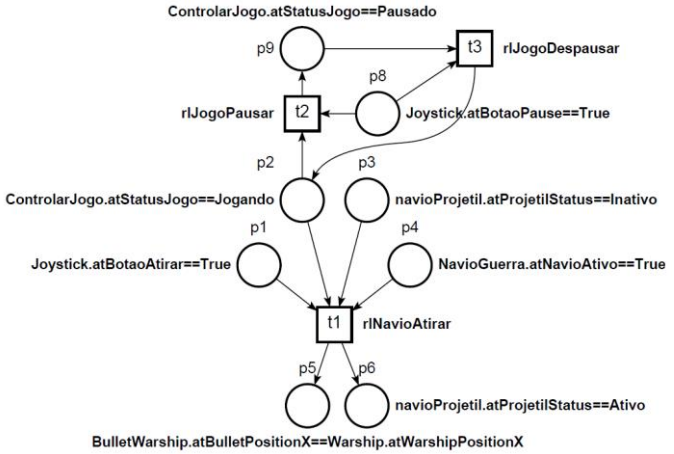

Fig. 8. RdP que agrupa os casos de uso *Controlar ataque* e *Pausar e despausar partida*

IV. Discussões e Conclusões

O Paradigma Orientado a Notificações surge como alternativa aos tradicionais usos de Sistemas Orientados a Regras que empregam mecanismos centralizados para realizar a inferência nas regras. O PON apresenta uma nova forma de conceber *software*, portanto necessita métodos específicos, sobretudo para garantir boas soluções de *software* usando o paradigma.

O Desenvolvimento Orientado a Notificações foi concebido para modelar *software* seguindo o PON. Os esforços se concentraram em fazer uso de técnicas atuais, estendendo-as para se adequarem ao PON. O método DON abrange atividades relacionadas aos requisitos e análise e projeto de *softwares* em PON, sendo aderente aos processos tradicionais de desenvolvimento de *software*.

O DON é iterativo, permitindo, assim, a melhoria dos modelos à medida que os ciclos ocorrem. Na utilização do DON para este trabalho, por exemplo, diversos atributos e métodos foram identificados somente durante a criação do modelo de componentes. Assim, os ciclos 2 e 3 do DON permitiram que os modelos anteriores fossem atualizados.

A existência de um perfil específico para o PON permitiu que a sintaxe e a semântica dos conceitos do PON fossem representados nos diagramas da UML. O modelo de Redes de Petri mostrou-se eficiente para simular o comportamento do *software* e permitir que ele seja comparado com os casos de uso e requisitos previamente estabelecidos.

O uso do método DON supera algumas limitações da modelagem de *softwares* e fornece uma abordagem abrangente, fácil de usar e eficiente para desenvolvimento de *softwares* baseados no PON. No entanto, existem algumas lacunas. A análise realizada com UML não beneficia a descoberta das regras no PON. Em parte, isso se deve ao fato de que a UML não foi desenvolvida para este paradigma. Assim, o processo de identificação de regras torna-se um intenso processo de síntese e requer muito esforço do desenvolvedor.

Pesquisas do grupo, em andamento, incluem o uso de métodos alternativos de modelagem e a criação de uma linguagem e ferramentas específicas que possam aderir aos conceitos de Sistemas Orientados a Regras e, mais especificamente, ao Paradigma Orientado a Notificações. Tanto o DON quanto as novas proposições se aplicam, naturalmente, no âmbito de sistemas de inteligência computacional e artificial.

References

[1] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Upper Saddle River, New Jersey: Prentice Hall International, 1995.

[2] L. C. V. Melo, J. M. Simão, and J. A. Fabro, "Adaptation of the Notification Oriented Paradigm (NOP) for the Development of Fuzzy Systems," *Mathw. Soft Comput. Mag.*, vol. 22, no. 1, pp. 40–64, 2015.

[3] G. J. Nalepa, S. Bobek, A. Ligęza, and K. Kaczor, "Algorithms for rule inference in modularized rule bases," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6826 LNCS, no. June 2015, pp. 305–312, 2011.

[4] J. M. Simão and P. C. Stadzisz, "Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues," *IEEE Trans. Syst. Man Cybern. Part A, Syst. Humans*, vol. 39, no. 1, pp. 238–250, 2009.

[5] L. V. B. Wiecheteck, P. C. Stadzisz, and J. M. Simão, "Um Perfil UML para o Paradigma Orientado a Notificações (PON)," in *Anais do III Congresso Internacional de Computacion Y Telecomunicaciones.*, 2011, pp. 1–16.

[6] L. V. B. Wiecheteck, "Método para projeto de software usando o Paradigma Orientado a Notificações – PON", Mestrado, Universidade Tecnológica Federal do Paraná, 2011.

[7] J. Cardoso and R. Valette, *Redes de Petri*. Florianópolis: Ed. da UFSC, 1997, p. 212.