UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

REALIZANDO PADRÕES DE WORKFLOW EM SISTEMAS BASEADOS EM COMPONENTES

TRABALHO DE GRADUAÇÃO

ALUNA: **FLÁVIA LEITE SOARES** (fls@cin.ufpe.br)
ORIENTADOR: **AUGUSTO CEZAR ALVES SAMPAIO** (acas@cin.ufpe.br)
CO – ORIENTADOR: **RODRIGO TEIXEIRA RAMOS** (rtr@cin.ufpe.br)

JANEIRO DE 2008

Assinaturas

Este Trabalho de Graduação é resultado dos esforços da aluna Flávia Leite Soares, sob a orientação do professor Augusto Cezar Alves Sampaio. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Flávia Leite Soares

_____

Augusto Cezar Alves Sampaio

Resumo

Workflow pode ser definido como uma seqüência de passos necessários para atingir a automação de processos de negócio, segundo um conjunto de regras pré-definidas. A tecnologia de *workflow* tem se mostrado bastante útil no gerenciamento de processos com alto grau de qualidade, maior eficiência e maior controle sobre suas operações.

Este trabalho propõe a especificação em UML-RT do conjunto de *Workflow Patterns* proposto por Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski e Alistair Barros para suportar o desenvolvimento de aplicações que implementam processos de negócios.

Este catálogo de padrões provê um conjunto de componentes reusáveis para acelerar o processo de desenvolvimento de aplicações de negócios e têm como objetivo prover uma maneira simples e consistente de traduzir prioridades de negócios e requisitos em soluções técnicas utilizando os componentes propostos.

Palavras-chave:
Padrões de Projeto, *Workflow Patterns*, UML-RT, Componentes.

Agradecimentos

Primeiramente quero agradecer a Deus pelo dom da vida, pela minha família, pela saúde, pelo Seu amor incondicional. Obrigada por ter conduzido minha vida com alegrias, tristezas, vitórias e dificuldades, mas em todo momento sempre ao meu lado, dando forças para não desistir dos meus sonhos.

Meu agradecimento especial à minha família que eu amo muito! A meu pai e minha mãe sempre amigos e conselheiros, vocês são os principais responsáveis pelo que sou. Sei que muitas vezes deixaram de realizar seus próprios desejos para que eu tivesse uma boa educação. Vocês me inspiram todos os dias a continuar seguindo em frente. Eu tenho muito orgulho de ter vocês como pais.

A meus irmãos Fábio e Nanda pelo amor, amizade, carinho, compreensão e brincadeiras em todos os momentos da minha vida e principalmente nestes anos em que moramos juntos estudando aqui em Recife.

A Thiago, uma pessoa muito especial que Deus colocou na minha vida, desde o início do curso, pelo seu apoio, paciência, amizade e principalmente pelo seu amor. Sem você eu não teria chegado até aqui!

A todos os meus amigos que estiveram juntos nessa caminhada. Ao quarteto fantástico: Pimentel, Camarotti, Bruno e eu, pelos momentos divertidos, seja fazendo projetos no CIn ou nos churrascos da sala. Aos amigos que fizeram o movimento "Volta Flavinha" para que eu não desistisse do curso para fazer Medicina: Sid, João Paulo, Rômulo, Figa, Márcio e Dio. E todos os engenheiros da nossa turma Black Box!

Aos que não estiveram tão perto, mas continuam sempre presentes: Aninha, Marcela, Diego, Priscila. A meus primos Diego e Tiago. A todos os amigos da Samsung que tiveram paciência comigo durante o TG e com quem aprendo todos os dias.

Aos professores do CIn pelo seu trabalho, dedicação e lições que nos fazem crescer pessoal e profissionalmente. Às pessoas que conheci no CIn, que certamente deixaram contribuições importantes para a minha formação.

Quero agradecer, enfim, a Rodrigo Teixeira que me deu todo o apoio necessário para que eu fizesse este trabalho, sempre disposto a tirar dúvidas e contribuir com o meu aprendizado.

Ao meu orientador Augusto Sampaio pela sua atenção, sempre me cobrando quando necessário e contribuindo com dicas valiosas para o meu trabalho.

Muito obrigada!

Sumário

Índice de Figuras

1. Introdução

Os processos de trabalho nas organizações têm sido encarados como essenciais para manter a sua competitividade no século XXI. As empresas buscam a reformulação constante de seus processos através de iniciativas de reengenharia a fim de garantir eficiência e qualidade total ao seu negócio.

Ao focarem no trabalho em equipe, na criação e reuso de um conhecimento comum na organização e na melhoria contínua dos processos de produção, há uma demanda das organizações pelo gerenciamento eficaz e eficiente dos seus processos de negócio.

Neste contexto, a tecnologia de *workflow* tem se mostrado bastante útil no gerenciamento de processos com alto grau de qualidade e personalização às necessidades dos clientes. A utilização desta tecnologia está intimamente ligada à busca de maior eficiência, à racionalização de custos e ao maior controle sobre suas operações.

Workflow é definido como uma seqüência de passos para atingir a automação de processos de negócio, segundo um conjunto de regras pré-definidas. Para suportar o desenvolvimento de aplicações que implementam processos de negócio há um conjunto de padrões de *Workflow* bastante conhecido, proposto por Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski e Alistair Barros no artigo *Workflow Patterns* [2]. Esta coleção de padrões foca em um aspecto específico do desenvolvimento de aplicações orientadas a processos: a descrição de dependências de fluxo de controle entre as atividades de um processo.

Workflows são descritos em linguagens de alto nível, normalmente utilizadas em fases iniciais do processo de desenvolvimento de software. Posteriormente, lógicas e padrões representados nestas linguagens são realizados em uma linguagem de mais baixo nível, como linguagens de programação [6]. Uma solução intermediária é o mapeamento de *workflow* em uma linguagem de modelagem, ainda com grande poder de abstração.

Um padrão pode ser definido como “uma idéia que foi útil em um contexto prático e provavelmente será útil em outros” [1] ou como “a abstração de uma forma concreta que se mantém recorrente em contextos específicos não arbitrários” [3].

Técnicas de padronização se estabeleceram como uma ferramenta importante em projetos de arquitetura devido a Christopher Alexander, um arquiteto da construção civil, que descreveu esta abordagem no seu livro *The Timeless Way of Building*, publicado em 1979[21]. Este livro provê uma introdução às idéias por trás do uso de padrões e explica as características e benefícios do uso de padrões na arquitetura [5].

A idéia de Alexander inspirou alguns desenvolvedores de software a embutir padrões de projeto no processo de desenvolvimento de software no início dos anos 80 e desde então muitos artigos e livros na área de Engenharia de Software foram publicados. Um dos catálogos de padrões de software mais conhecido é o *Design Patterns: Elements of Reusable Object-Oriented Software* [3] que descreve soluções simples e elegantes, de acordo com diferentes propósitos e escopos, para problemas específicos do projeto de engenharia de software.

Estamos propondo neste trabalho padrões de projeto que se referem especificamente a problemas recorrentes e soluções verificadas relacionadas ao

desenvolvimento de aplicações de *workflow* em particular, e de forma mais abrangente a aplicações orientadas a processos.

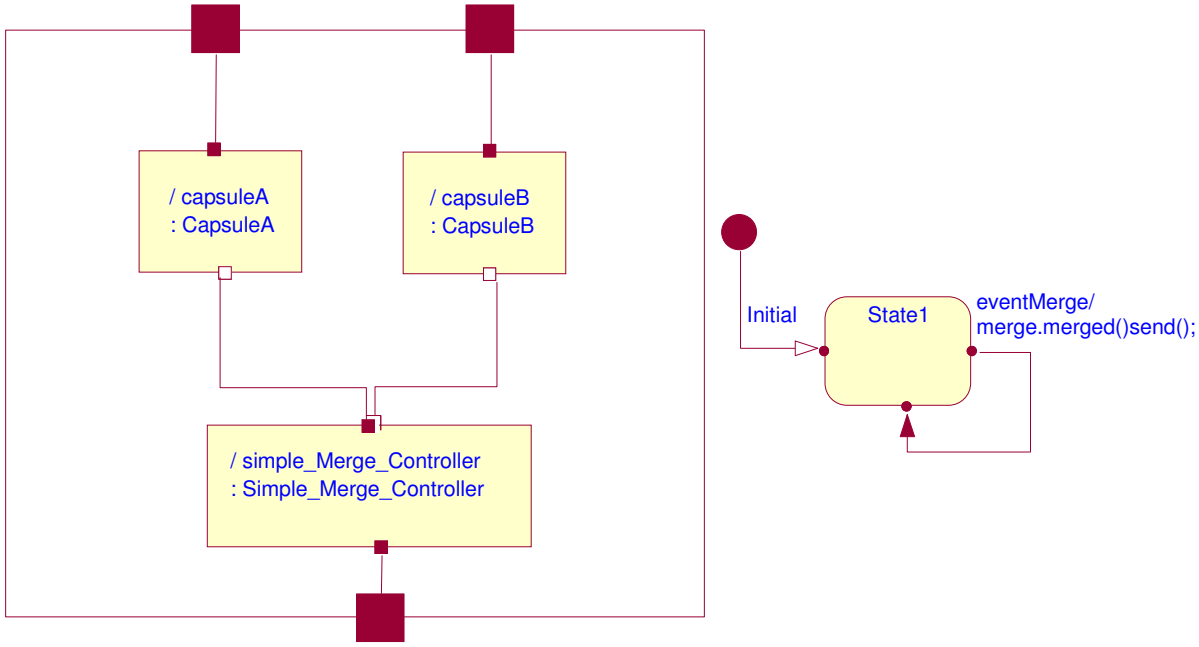
Um fluxo de *workflow* pode ser descrito usando técnicas de diagramação formais ou informais, mostrando fluxos direcionados entre passos de processamento. Este trabalho propõe a apresentação destes padrões utilizando a linguagem de modelagem UML-RT[31], uma linguagem de descrição arquitetural para sistemas baseados em componentes reativos, a fim de prover um conjunto de componentes reusáveis para o desenvolvimento de aplicações orientadas a processos.

Com UML-RT representamos os fluxos de *workflow* de forma compreensível à equipe técnica que irá implementar os processos, facilitando o processo de desenvolvimento, uma vez que o comportamento dos componentes é descrito numa linguagem próxima às linguagens de programação e por meio dos componentes criados o comportamento recorrente do padrão pode ser reutilizado diversas vezes.

Devido à complexidade destes padrões, os componentes apresentados em nosso trabalho são representados em três visões arquiteturais: diagramas de classes, estrutura e estados. O comportamento destes componentes é extraído de informações descritas implicitamente no catálogo de proposto pela *Workflow Patterns Initiative* [4] e a partir da definição dos padrões em Redes de Petri. Para ilustrar o fluxo do padrão apresentado utilizaremos diagramas de atividades de UML a partir das descrições de cada padrão encontradas em [20] onde é utilizada esta notação para representá-los.

Os padrões modelados foram estruturados de forma a simplificar a sua apresentação, por isso adotamos alguns critérios para sua construção: cada padrão é constituído por um componente formado pela composição de um componente principal – o controlador, que implementa de fato o comportamento do padrão através da sua máquina de estados – e componentes auxiliares - que representam as atividades relacionadas com o processo descrito e que serão especificadas no momento da instanciação do padrão no modelo do processo. Desta maneira o primeiro componente representa a estrutura do padrão e os componentes seguintes (controlador e auxiliares) representam os papéis encontrados no padrão. Seu comportamento e estrutura mostram então o relacionamento entres estes papéis no padrão.

A seguir há um exemplo de como os padrões estão estruturados: um diagrama de estrutura apresentando os elementos que o constituem e suas ligações e a máquina de estados do controlador que implementa o seu comportamento, com seus eventos e transições.



Estes diagramas representam o padrão *Simple Merge* que faz a junção de ramos distintos (A e B) sem sincronizá-los. Com esta construção, as tarefas em comum necessitam ser descritas apenas uma vez. Seu controlador, o *Simple_Merge_Controller*, envia um sinal *merged(void)* avisando que o *merge* foi concluído através da porta *merge*.

Todos os padrões são apresentados conforme a representação acima incluindo o diagrama de classes que participam do padrão apresentado. Cada cápsula representa um componente que poderá ser reusado e ao serem combinadas, como no diagrama acima, formam um padrão específico. Os padrões também podem ser combinados de forma a construir processos que possuem diversos tipos de fluxo de controle.

No total, modelamos treze padrões de *Control Flow* apresentados em [2], classificados em *Basic Control Flow Patterns, Advanced Branching and Synchronizazation Patterns, State-based Patterns, Cancellation and Force Completion Patterns* descrevendo seus objetivos, estrutura, elementos de colaboração, implementação e conseqüências.

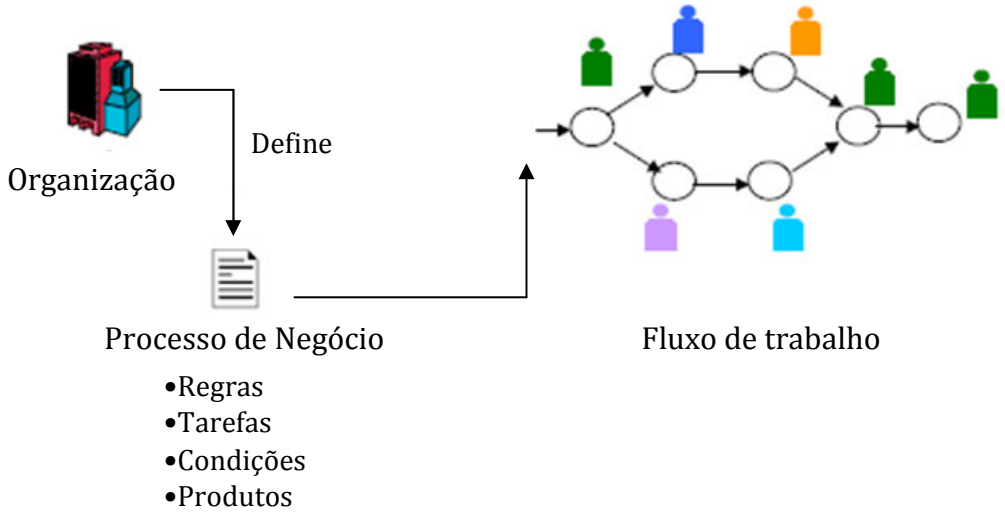
O restante da monografia será organizado como a seguir:

- No Capítulo 2 são introduzidos os conceitos de *Workflow*, Gerenciamento de *Workflow* e Gerenciamento de Processos de Negócios e como estes conceitos influenciam no desenvolvimento de aplicações orientadas a processos, além da apresentação do conjunto de *Workflow Patterns*.
- Em seguida, no Capítulo 3, são abordadas as linguagens de descrição de *Workflow* existentes e a motivação para a modelagem destes padrões em UML-RT com uma breve apresentação dos conceitos relacionados a esta linguagem de modelagem.
- Após o conteúdo teórico inicial, são apresentados no Capítulo 4 os padrões de *Control Flow* selecionados e implementados utilizando UML-RT com seus diagramas de classes, estrutura e estados.

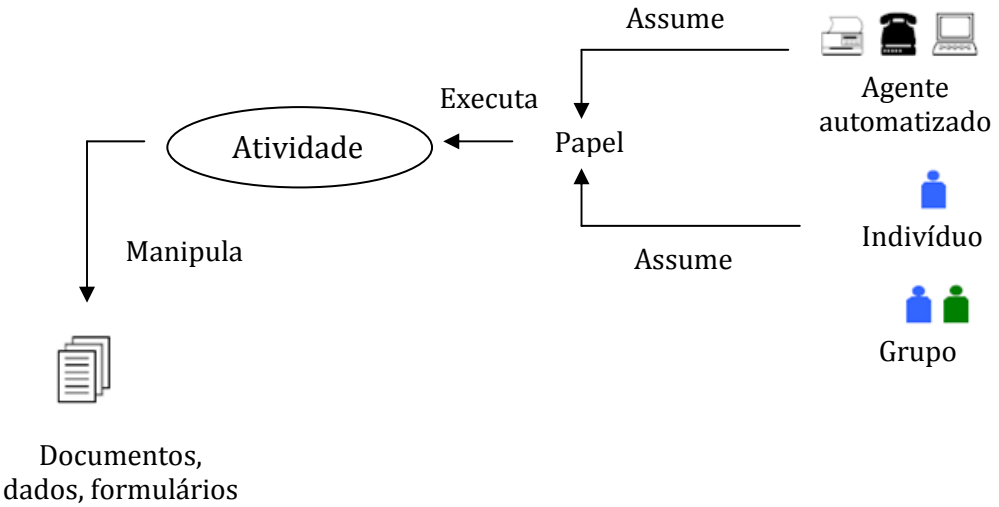
- Exemplos práticos da utilização destes padrões são apresentados no Capítulo 5 através dos processos *Travel Request* e *Submit Form*.
- Por fim, é feita a conclusão juntamente com a análise de trabalhos relacionados e as possibilidades de trabalhos futuros a partir deste Trabalho de Graduação no Capítulo 6.

2. Workflow

Processo de negócio é um procedimento no qual documentos, informações ou tarefas são passados entre participantes de acordo com um conjunto de regras definidas a serem alcançadas ou realizadas para o objetivo do negócio. Um *workflow* é a representação do processo de negócio que especifica as atividades individuais, a ordem e as condições sob as quais as atividades devem ser executadas, as ferramentas a serem usadas em cada atividade, etc.



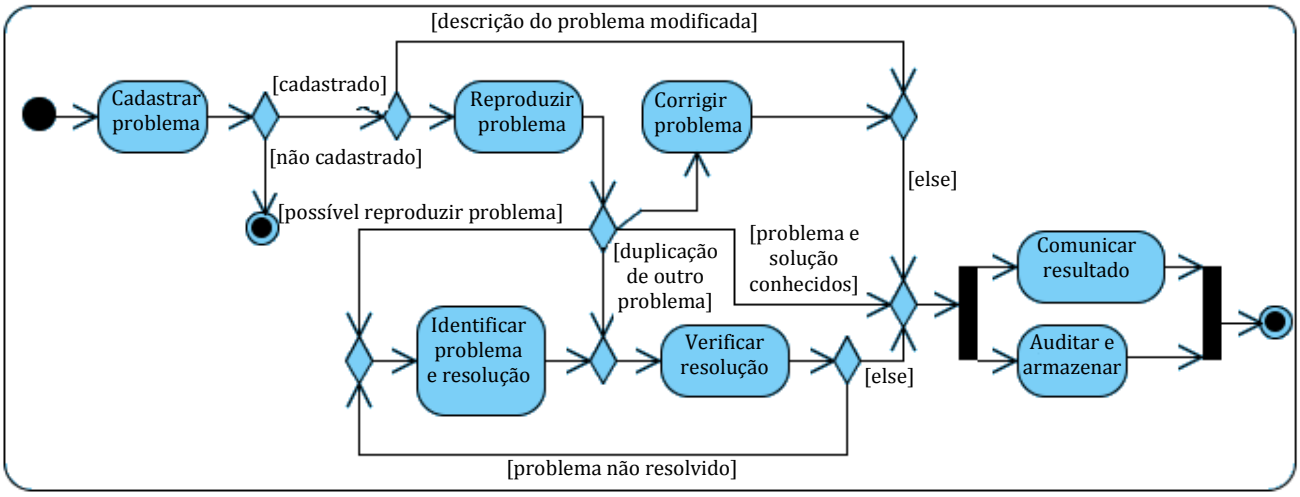
A modelagem de um *workflow* é a representação do processo em uma formalização computacional e processável. Este modelo deve possuir as atividades, suas condições de início e finalização, regras para sua execução, responsáveis por cada atividade, documentos utilizados e aplicações necessárias.



A seguir um exemplo de um *workflow* para um processo de acompanhamento de "*bugs*" descrito em UML através de um diagrama de atividades. Este processo apóia equipes de garantia de qualidade ou de suporte a clientes.

Um "*bug*" ou problema é identificado e deve ser cadastrado; o cadastro deve ser checado; a partir de uma instância do problema sua causa é identificada; uma

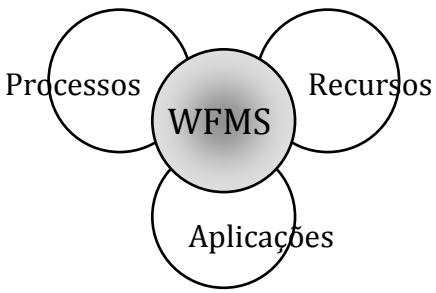
resolução é identificada, que deve ser comunicada de volta ao grupo que estava com o problema. Este processo contém um loop potencial. Se a resolução não for bem sucedida, então o problema volta ao passo de resolução. Retrabalho é um tanto comum em processos, porque a verificação da corretude do trabalho pode ser possível apenas alguns passos adiante.



2.1 *Gerenciamento de Workflow*

Gerenciamento de *Workflow* (*Workflow Management* – WFM) diz respeito ao gerenciamento do fluxo de trabalho a ser feito no tempo certo pela pessoa adequada. Um sistema de gerenciamento de *workflow* (*Workflow Management System* - WFMS) é usado para definição, criação e gerência da execução de fluxos de trabalho através do uso de software, capaz de interpretar a definição de processos, interagir com seus participantes e, quando necessário, invocar ferramentas e aplicações [14]. Este sistema define, gerencia e executa *workflows* através de um software cuja ordem de execução é direcionada por uma representação computacional da lógica do *workflow*.

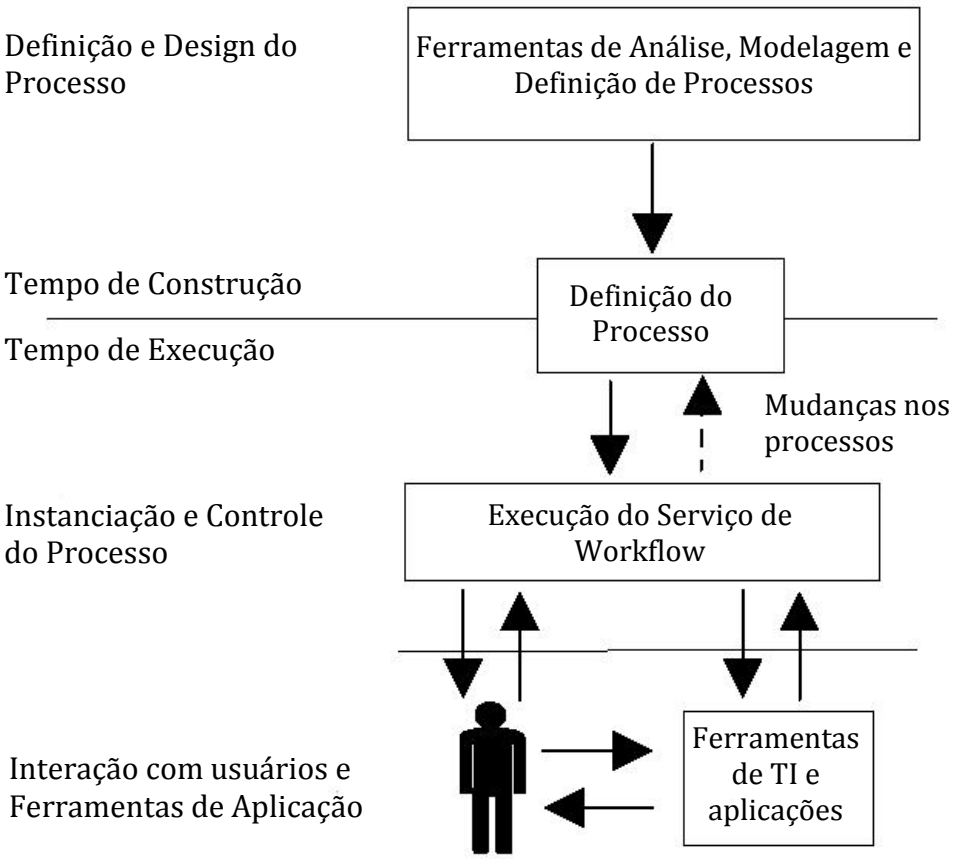
Um sistema de *workflow* (*Workflow System* - WFS) é um sistema baseado em WFMS que suporta um conjunto específico de processos de negócio pela execução de definições computáveis de processos. A idéia básica consiste na separação de processos, recursos e aplicações e foco na logística de processos de trabalho, não no conteúdo individual das tarefas.



As três áreas funcionais para o suporte ao Gerenciamento de *Workflow* são:

- Funções em tempo de construção: concernentes à definição e à modelagem do processo de *workflow* e suas atividades;

- Funções de controle em tempo de execução: relacionadas ao gerenciamento de processos de *workflow* em um ambiente operacional e seqüenciamento das atividades a serem manipuladas como parte de cada processo;
- As interações em tempo de execução com usuários e ferramentas de aplicações de TI para processamento dos passos das atividades.



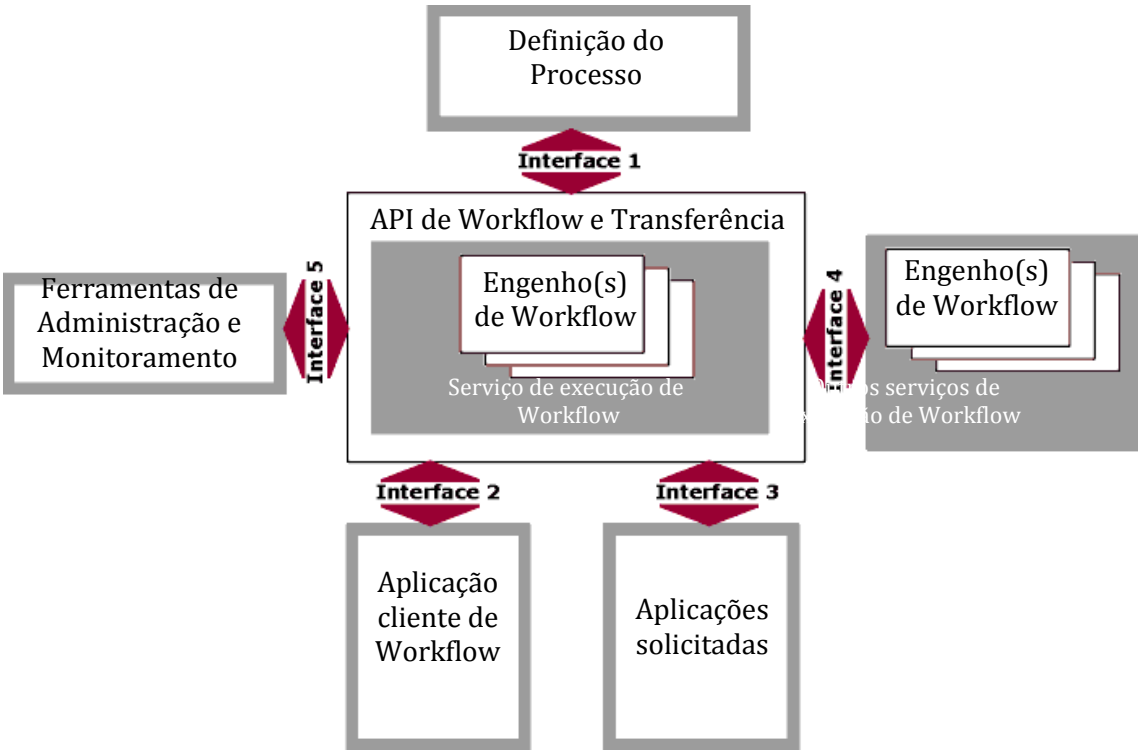
O conceito que estende o de Gerenciamento de *Workflow* é o de Gerenciamento de Processos de Negócio (*Business Process Management* – BPM) que é de fato a junção da tecnologia de processos cobrindo três categorias de processos: 1. Pessoa para pessoa; 2. Sistemas para sistemas; 3. Sistemas para pessoas – todas a partir de uma perspectiva centrada em processos.

Business Process Management é uma área de conhecimento e pesquisa da interseção entre gerenciamento e tecnologia da informação, compreendendo métodos, técnicas e ferramentas para projetar, implementar, controlar e analisar processos de negócio envolvendo pessoas, organizações aplicações, documentos e outras fontes de informação. [7]

2.2 *Modelo de Referência para Gerenciamento de Workflow*

A necessidade crescente de flexibilidade e interoperabilidade entre sistemas de *workflow*, sua capacidade de operar em diversas plataformas e a existência de processos de negócio complexos que envolvem diferentes departamentos ou empresas trouxeram a necessidade de padronizar estes serviços de suporte a *workflow*.

Um modelo de referência para sistemas de gerenciamento de *Workflow* foi definido pelo *Workflow Management Coalition* (WfMC). Este comitê especifica um framework para sistemas de *workflow*, identificando suas características, funções e interfaces[8].



Modelo de Referência e Glossário: especificam um framework para sistemas de *workflow*, identificando suas características, funções e interfaces. Desenvolvimento de terminologia padrão para sistemas de *Workflow*.

Interface para Ferramentas de Definição de Processos (1): definição de uma interface padrão entre a definição do processo e ferramentas de modelagem e os engenhos de *workflow*.

Interface para a aplicação cliente de *Workflow* (2): definição de APIs para aplicações cliente que requisitam serviços do engenho de *workflow* para controlar a progressão dos processos, atividades e itens de trabalho.

Interface para aplicações solicitadas (3): uma definição de interface padrão de APIs que permitem o engenho de *workflow* invocar uma variedade de aplicações através de um agente de software comum.

Interface de Interoperabilidade de *Workflow*s (4): Definição de modelos de interoperabilidade entre *workflow*s e os padrões correspondentes para suportar o funcionamento interligado.

Interface de Ferramentas de Administração e Monitoramento (5): definição de funções de monitoramento e controle.

Conformidade: para desenvolver a política do Comitê sobre a conformidade do produto com as especificações e acordar uma abordagem para certificação do fornecedor.

***Workflow* Application Programming Interface (WAPI)**: interface ao redor do serviço de implantação de *workflow*; suporta funções de gerenciamento de *workflow* através das cinco áreas funcionais.

Serviços de Execução de *Workflow*: um serviço de software que consiste de um ou mais engenhos para criar, gerenciar e executar instâncias de *workflow*. As aplicações se comunicam com esse serviço via WAPI.

Engenho de *Workflow*: um serviço de execução de *workflow* consiste de múltiplos engenhos de *workflow*; é o ambiente de execução da instância do *workflow* e é responsável pelo controle do ambiente em tempo de execução de um serviço de implantação.

O serviço de execução de *workflow* pode ser considerado uma máquina de estados: instâncias de processos ou atividades mudam de estado em resposta a eventos externos (por exemplo, finalização de uma atividade) e decisões de controle específicas tomadas por um engenho de *workflow* (por exemplo, navegação para um próximo passo de uma atividade dentro de um processo).

Ferramentas para definição de processos são usadas para criar, descrever e documentar processos de negócios. Estas ferramentas são baseadas em uma linguagem de definição de processo, um modelo de relacionamento entre objetos e um conjunto de comandos para transferir informação entre usuários participantes.

Uma descrição de um processo é uma estrutura descrevendo as tarefas ou atividades a serem executadas e sua ordem de execução. A descrição de um processo pode ser provida por uma Linguagem de Descrição de *Workflow*.

Uma Linguagem de Definição de Processos de *Workflow* (*Workflow Process Definition Language* - *WPDL*) permite a definição de ações a serem tomadas em cada estado possível, pré e pós-condições dos estados, transições entre estados, definição da seqüência de tarefas ou estados, definição de estados automatizados e de estados que requerem entrada do usuário.

A qualidade da solução de automação de quaisquer processos de negócio é determinada pela sua implementação, mas ao mesmo tempo, é restrita às limitações da tecnologia subjacente. Um dos fatores que determinam a qualidade de um produto de *Workflow* é o poder de expressão da modelagem das construções dos processos. Este poder de expressão ou qualidade da modelagem tem um profundo impacto na flexibilidade, desempenho e manutenibilidade de uma implementação de *Workflow*.

Um *Workflow* pode ser descrito usando técnicas de diagramação formais ou informais, mostrando fluxos direcionados entre passos de processamento. Existem inúmeros formalismos para descrição de *workflow* como, por exemplo, Redes de Petri, álgebra de processos, diagramas de atividades de UML, máquina de estados finita, *Business Process Modeling Notation* (BPMN) entre outras. Dentre as linguagens não formais usadas pela indústria ou pela comunidade científica estão a

Business *Process Modeling Language* (BPML) e a *Yet Another Workflow Language* (YAWL), ambas com suporte às tecnologias XML.

Todas estas linguagens representam os fluxos de *Workflow* de forma padronizada e compreensível a todos os *stakeholders* do negócio. Estes *stakeholders* incluem os analistas do negócio que criam e refinam os processos, os desenvolvedores técnicos responsáveis por implementar os processos e os gerentes de negócio que monitoram e gerenciam os processos.

A *Unified Modelling Language* – UML é utilizada como uma WPDL, pois define diferentes tipos de diagramas para sistemas orientados a objetos, auxilia na especificação, visualização e documentação de modelos e oferece notações gráficas para modelos de *workflow*.

2.3 Workflow Patterns

Um grupo de pesquisa da *Eindhoven University of Technology* juntamente com a *Queensland University of Technology* fazem parte de uma iniciativa que provê uma base conceitual para a tecnologia de processos. A pesquisa analisa várias perspectivas (*control flow*, *data*, *resource* e *exception handling*) que são suportadas por uma linguagem de *workflow* ou uma linguagem de modelagem de processos de negócio descrevendo os padrões relevantes para sistemas de informação para definição de processos.

A iniciativa *Workflow Patterns* define quatro perspectivas de padrões:

- ***Control Flow***: captura os aspectos relacionados a dependências de fluxo de controle entre várias tarefas (por exemplo, paralelismo, escolha, sincronização, etc.). Originalmente vinte padrões foram propostos para esta perspectiva, mas na última iteração ela cresceu para quarenta e três padrões.
- ***Data Flow***: captura as várias maneiras pelas quais dados são representados e utilizados em *workflows*: lida com a passagem de informação, escopo de variáveis, etc.
- ***Resource***: lida com recursos para alocação de tarefas, delegação, etc.
- ***Exception Handling***: lida com as várias causas de exceções e várias ações que precisam ser tomadas como resultado da ocorrência de exceções.

Estes padrões podem ser utilizados em instâncias de processos dentro de um engenho de *workflow* para execução dos fluxos de controle. As instâncias de processos ou atividades mudam de estado em resposta a eventos externos e o engenho toma decisões de controle específicas como navegar para um próximo passo de atividade do processo.

Estes recursos reusáveis são úteis para acelerar o processo de desenvolvimento de aplicações de negócios e têm como objetivo prover uma maneira simples e consistente de traduzir prioridades de negócios e requisitos em soluções técnicas e facilitar o reuso de capital intelectual tais como arquiteturas de referência, frameworks e outros recursos.

Há vários formatos usados na literatura para descrever padrões, no entanto, há uma concordância acerca do que um padrão deve conter. Alexander estabeleceu que um padrão deve ser descrito em cinco partes:

- Nome: uma descrição da solução.
- Exemplo: com figuras, diagramas ou descrições ilustrando um protótipo de aplicação.
- Contexto: descrição das situações sob as quais o padrão se aplica.
- Problema: descrição das forças e restrições envolvidas e como elas interagem.
- Solução: relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão, citando variações e formas de ajustar a solução segundo as circunstâncias.

Neste trabalho cada padrão será apresentado com o seu nome, uma descrição sobre o que o padrão faz (seu objetivo), uma motivação para aplicação do padrão, a solução proposta identificando sua estrutura estática com diagramas de classes e diagramas de estrutura, e seu comportamento dinâmico utilizando diagramas de estado. A descrição da solução contém as linhas diretivas para a implementação do padrão e identificam o contexto para a sua aplicação, ou seja, as situações sob as quais o padrão é aplicável.

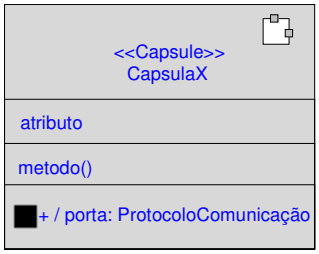
3. UML – RT

A automação dos processos de negócio requer uma implementação de qualidade a qual é determinada pela qualidade da modelagem das construções dos processos. A linguagem utilizada para sua modelagem deve ter um poder de expressão tal que garanta à solução flexibilidade, desempenho e manutenibilidade. As linguagens que representam os fluxos de *workflow* o fazem de forma padronizada e compreensível a todos os *stakeholders* do negócio, desde gerentes até a equipe técnica, sendo estas, portanto, linguagens de alto nível e que não consideram questões de implementação em sua representação.

Este trabalho propõe a utilização da linguagem UML-RT [11] para modelar os padrões de *workflow* e criar um catálogo de componentes reusáveis a fim de facilitar o processo de desenvolvimento, uma vez que esta linguagem está mais próxima da equipe desenvolvedora, e prover um direcionamento acerca de como os componentes devem ser implementados.

UML – RT é uma extensão da *Unified Modeling Language* (UML) usada para modelagem de sistemas embarcados ou de tempo real que incorpora os conceitos de ROOM (*Real-Time Object-Oriented Modeling*) [31], notação visual orientada a objetos para modelagem de sistemas de tempo real, à UML padrão. Para modelagem dos aspectos estruturais do sistema são utilizados três elementos básicos: Cápsulas, Portas e Conectores. [10]

Cápsulas são classes ativas, que possuem seu próprio thread de controle lógico, e cujo fluxo de controle é sensível apenas à troca de mensagens com outras cápsulas [9]. Elas interagem com outras cápsulas através de um ou mais objetos de fronteira baseados em sinais chamados Portas. Cápsulas podem conter outras cápsulas com suas responsabilidades correspondentes.



Cápsulas possuem métodos e atributos, como classes ordinárias, mas são visíveis apenas ao contexto interno da cápsula. A representação visual de uma cápsula é semelhante à de classes ordinárias, mas contém o estereótipo «Capsule» e um compartimento adicional para portas. [9]

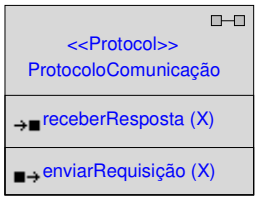
Uma Porta é a única interface de comunicação das cápsulas com o ambiente externo ou suas subcápsulas. Fortalecem o encapsulamento e o reuso de cápsulas e permitem que sejam usadas apenas em função das suas interfaces. As cápsulas não precisam conhecer as características do ambiente externo ou de suas subcápsulas.

Cada porta deve realizar algum Protocolo de comunicação o qual define o tipo e a seqüência válida de mensagens que podem ser comunicadas pelas portas. O termo evento significa a recepção de uma mensagem pela porta.

Existem dois tipos de portas: Portas *Relay* e Portas *End*. As Portas *Relay* são conectadas às subcápsulas para simplesmente repassar os sinais, enquanto que as Portas *End* são conectadas à Máquina de Estados da cápsula para disparar possíveis transições entre os estados desta máquina.

Para que duas cápsulas se comuniquem é necessário existir uma conexão estrutural entre suas portas através dos Conectores. Estes são canais de informação que conectam apenas duas portas compatíveis com o mesmo Protocolo.

O Protocolo define as regras de comunicação entre portas de cápsulas. São classes abstratas puramente comportamentais, diferente de cápsulas, que são elementos estruturais. Em um protocolo é possível definir mais de uma interface, chamada papel, que será implementada pela porta. Cada papel de um protocolo define os sinais de entrada e de saída que podem ser recebidos ou enviados por uma porta. Máquinas de estado de protocolo são usadas para indicar às cápsulas as regras de comunicação específicas de cada porta, mas não podem pré-determinar o comportamento interno das cápsulas.



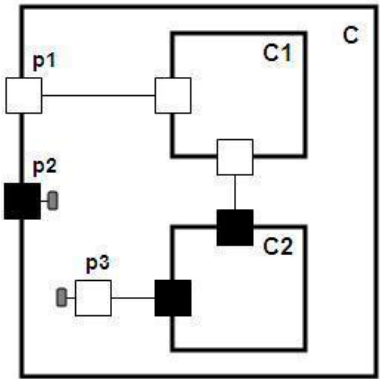
Protocolos e Máquinas de Estados Finitos são úteis para capturar aspectos comportamentais do sistema. Protocolos definem um acordo contratual entre os participantes que se comunicam. As definições do fluxo de sinais permitido, de entrada ou de saída, entre duas cápsulas que interagem são definidas nas responsabilidades do Protocolo que identificam o que é recebido ou enviado por uma porta. Estas são chamadas protocolo base e conjugado, respectivamente.

Uma Máquina de Estados Finita que está associada a uma cápsula especifica a seqüência de estados pela qual um objeto passa à medida que reage às mensagens que chegam às Portas *End*. Uma máquina de estados de uma cápsula processa as mensagens que ela recebe através das transições de estados disparadas pelas mensagens.

A estrutura interna de uma cápsula pode ser composta de subcápsulas e portas de comunicação, que podem ser públicas ou protegidas. Portas públicas compõem a interface de comunicação com o meio externo e portas protegidas podem conectar subcápsulas ou comunicar eventos internos [9]. O diagrama de estrutura de uma cápsula é um diagrama de colaboração especializado que apresenta as suas subcápsulas e portas associadas.

Para duas cápsulas se comunicarem elas devem estar conectadas através de suas portas. No diagrama de estrutura esta ligação é representada por um conector simples, isto é, uma linha sem orientação. Duas portas só podem ser ligadas se possuírem protocolos complementares, que possuem sinais de entrada e saída invertidos um em relação ao outro, ou simétricos, que possuem o mesmo conjunto de sinais em tipo de dado e orientação. Em protocolos binários, portas *base* e *conjugada* são complementares entre si. Portas *conjugadas* são representadas por caixas brancas e portas *base* por caixas pretas.

A figura seguinte mostra a cápsula **C** contendo duas subcápsulas **C1** e **C2**. A porta *relay* e pública conjugada **p1** comunica mensagens do ambiente externo diretamente para a cápsula **C1** e vice-versa. A porta *end*, *base* e pública **p2** transmite mensagens do ambiente externo para a máquina de estados da cápsula **C**. A porta *end*, conjugada e protegida **p3** permite que a máquina de estados de **C** troque mensagens com a cápsula **C2** e não se comunica com o meio externo.

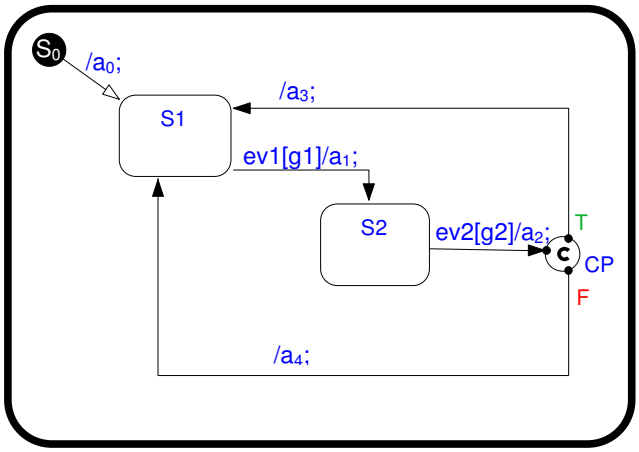


Além do comportamento passivo definido pelos seus métodos, as cápsulas possuem um comportamento dinâmico definido pela sua máquina de estados. Este comportamento é reativo, pois depende de eventos externos percebidos pela máquina de estados, isto é, apenas eventos que ocorrem em portas *end*.

O círculo preto S_0 representa o estado inicial da máquina de estados com transição automática para S_1 quando a cápsula é criada. Transições iniciais são geralmente usadas para configuração interna e podem ter uma ação associada.

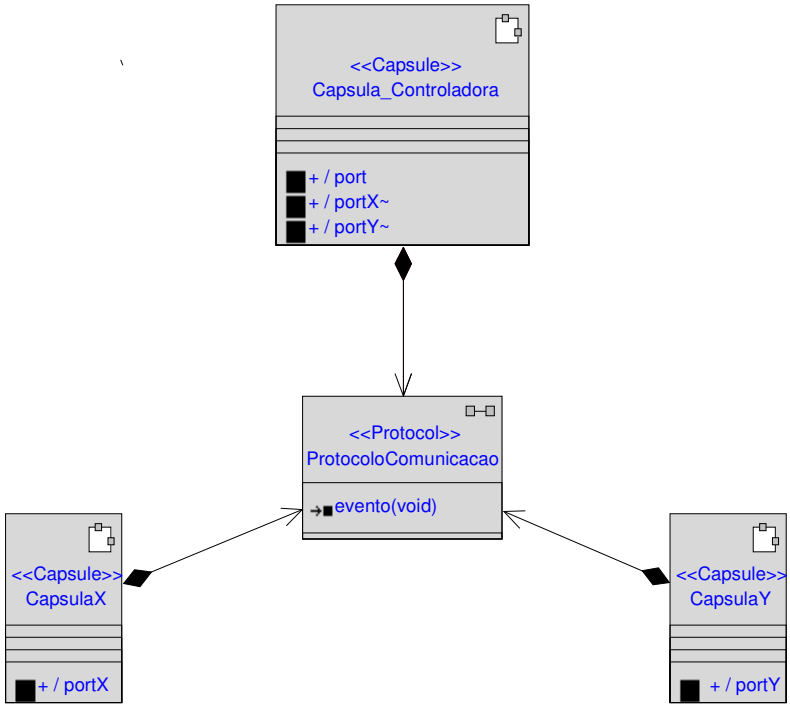
Conceitualmente, a guarda de uma transição é avaliada antes da ocorrência do evento que dispara aquela transição, mesmo que este evento não venha a ocorrer. Se a expressão lógica da guarda for falsa, então a ocorrência do evento de disparo está bloqueada[9]. Ao realizar uma transição a máquina de estados executa a ação associada a esta transição.

Já as expressões lógicas de pontos de escolha são avaliadas quando o este ponto é alcançado. O círculo com a letra 'C' é um pseudo-estado que executa uma expressão lógica (condição de escolha) para decidir qual transição de saída executar. Se a expressão lógica for verdadeira, a transição do ponto de junção T é disparada, executando a ação a_3, caso contrário, a transição do ponto de junção F é disparada e a ação a_4 é executada.



Em UML-RT, máquinas de estado não possuem estados finais, porque cápsulas são consideradas elementos ativos, que nunca acabam seu processamento, a não ser que o ambiente externo decida destruí-la[9].

Além dos diagramas de estrutura e de estados, utilizaremos o diagrama de classes para representar a estrutura e as relações das cápsulas e protocolos modelados para a implementação do padrão.



No diagrama mostrado temos a *Capsula_Controladora* que é composta pelas subcápsulas *CapsulaX* e *CapsulaY*. Estas cápsulas se comunicam através do *ProtocoloComunicação* que possui o sinal de entrada *evento(void)* que não transmite dados, pois não possui argumento, apenas indica a ocorrência de um evento. As cápsulas ficam cientes deste evento através de suas portas *end* públicas *port*, *portX* e *portY*.

Como UML-RT é apenas uma especificação, é necessária uma ferramenta CASE para especificação e implementação de modelos. No momento só há uma ferramenta que implementa UML-RT: a *Rational Rose Real Time* da IBM [12], que foi usada neste trabalho para realização dos padrões implementados.

Os padrões serão apresentados neste trabalho através de 1. Seu Diagrama de Classes; 2. Diagramas de Estrutura das cápsulas participantes; 3. Diagramas de Estados das cápsulas de controle.

Embora a versão 2.0 de UML [15] tenha incorporado vários conceitos de UML-RT e de outras notações (que a tornariam inclusive mais completa que UML-RT), seus elementos e diagramas ainda são ambíguos e pouco divulgados [9]. De fato, os elementos relevantes à representação dos padrões abordados neste trabalho podem ser descritos satisfatoriamente com UML-RT, inclusive com ferramentas de apoio sólidas [12], o que ainda não é possível para UML 2.0. Outra vantagem de UML-RT são os estereótipos para cápsulas e protocolos, e as noções

claras e intuitivas de componentes reativos e independentes, também pouco claros em UML 2.0[9].

4. Realizando Control Flow Patterns em UML-RT

A iniciativa *Workflow Patterns* foi estabelecida com o objetivo de delinear e descrever os requisitos fundamentais que surgem durante a modelagem de processos de negócio em uma base recorrente. O primeiro resultado deste projeto de pesquisa foi um conjunto de vinte padrões descrevendo a perspectiva *Control Flow* dos sistemas de *workflow*. A partir deste primeiro resultado, estes padrões têm sido amplamente usados por profissionais, vendedores e acadêmicos para seleção, projeto e desenvolvimento de sistemas de *workflow*. [2].

A perspectiva dos Padrões *Control Flow* descreve as **atividades** e suas **ordens** de execução através de diferentes construtores, que permitem o fluxo de controle de execução. Atividades elementares são unidades atômicas de trabalho, e compostas com outras modularizam uma ordem de execução de um conjunto de atividades.

Os quarenta e três padrões de *Control Flow* estão divididos em:

- *Basic Control Flow Patterns*
- *Advanced Branching and Synchronizazation Patterns*
- *Multiple Instance Patterns*
- *State-based Patterns*
- *Cancellation and Force Completion Patterns*
- *Iteration Patterns*
- *Termination Patterns*
- *Trigger Patterns*

Este trabalho apresenta treze padrões de *Control Flow* propostos pela iniciativa que correspondem aos grupos *Basic Control Flow Patterns, Advanced Branching and Synchronizazation Patterns, State-based Patterns* e *Cancellation and Force Completion Patterns* descrevendo seus objetivos, estrutura, elementos de colaboração, implementação e conseqüências.

O conjunto de padrões está descrito no seguinte formato:

- Nome e descrição do padrão.
- Diagrama de Atividades para ilustrar o fluxo do workflow (este diagrama é uma forma de ilustrar, de forma gráfica, como o padrão se comporta) que foram extraídos de [28] e [29]. A nomenclatura de cada atividade do diagrama não corresponde diretamente aos nomes das cápsulas implementadas no padrão.
- Diagrama de Classes para mostrar a colaboração dos elementos do padrão e uma descrição das responsabilidades de cada componente.
- Diagrama de Estrutura para mostrar a estrutura estática do padrão. As cápsulas de controle são chamadas de controladores e cada cápsula que representa uma atividade de workflow é denominada genericamente pelas letras A, B, C e D.
- Diagrama de Estados das cápsulas principais responsáveis pelo comportamento de cada padrão. Estes diagramas definem como estas cápsulas devem se comportar ao serem reusadas.

A apresentação dos componentes será feita de modo que cada padrão é uma cápsula e esta contém um controlador ou mais (se o padrão utilizar conceitos de

outros padrões existentes) e cápsulas auxiliares para representar as atividades relacionadas com o processo descrito.

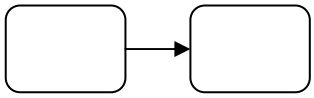
As cápsulas compostas não possuem máquinas de estado e por isso não possuem portas *end*. Por sua vez, as cápsulas simples não possuem subcápsulas e possuem máquinas de estados. O comportamento das cápsulas compostas é diretamente representado pela composição das cápsulas internas a elas, e das cápsulas simples pela sua máquina de estados.

4.1 Basic Control Flow Patterns

Esta classe de padrões captura os aspectos essenciais do processo de controle e seguem as definições propostas na especificação do *Workflow Management Coalition* (WfMC) [13].

4.1.1 Sequence

Uma atividade em um processo de workflow é habilitada após a finalização de uma atividade precedente no mesmo processo. [4]



O padrão *Sequence* é o bloco de construção fundamental de processos. É usado para construir uma série de tarefas consecutivas que executam uma após a outra.

Exemplo

A tarefa *verificar conta* executa após a captura das informações sobre o cartão de crédito.

Diagrama de Classes

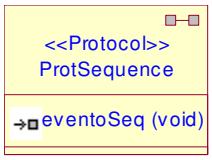


Figura 1. Diagrama de Classes do padrão *Sequence*

- *ProtSequence:* Este protocolo possui um sinal de entrada *eventoSeq(void)* que indica a finalização de uma atividade para outra cápsula através das portas que o implementam. A definição deste protocolo com o sinal *eventoSeq(void)* é uma maneira de padronizar as interfaces de cápsulas que se comunicam nos processos (no estudo de caso veremos como esta conexão é útil para ligar cápsulas que implementam atividades independentes de um padrão).

Duas tarefas fazem parte do padrão *Sequence* se há uma fronteira de controle entre estas tarefas que não possui guardas ou condições associadas a ela. Uma restrição associada ao contexto deste padrão é que uma instância do padrão *Sequence* não pode ser iniciada até que a execução da *thread* de controle precedente tenha completado.

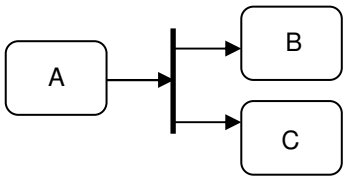
Assim, este comportamento é especificado pelo elemento de conexão simples de UML-RT que deve ser usado para conectar duas cápsulas através de suas portas que implementam este protocolo. O evento *eventoSeq(void)* será

transmitido através do conector indicando que uma atividade precedente concluiu sua execução e será recebido pela cápsula adjacente.

Figura 2. Conector simples de UML-RT para o padrão *Sequence*

4.1.2 Parallel Split

A divergência de um *branch* em dois ou mais *branches* paralelos que executam concorrentemente. [4]



O padrão *Parallel Split* ou *AND-split* permite que uma única thread de execução seja dividida em dois ou mais *branches* que podem executar tarefas concorrentemente. Estes *branches* podem ou não ser ressincronizados futuramente.

Exemplo

Quando um *alarme de invasão* é recebido, são disparadas as tarefas *enviar patrulha* e *informar polícia* imediatamente.

Diagrama de Classes

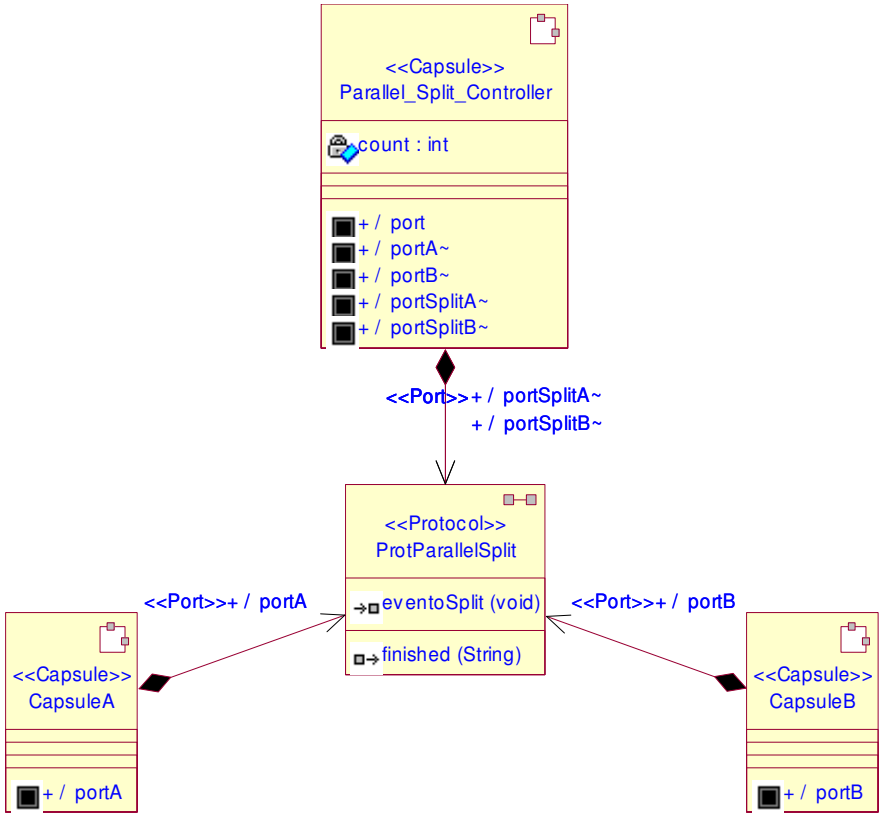


Figura 3. Diagrama de Classes do padrão *Parallel Split*

- *Parallel_Split_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. Ela executa sua primeira

atividade e em seguida ativa as duas cápsulas que irão executar em paralelo. Esta cápsula possui como atributo o inteiro *count* para indicar quantas das cápsulas que foram disparadas completaram sua execução e o controlador possa voltar ao seu estado inicial.

- *CapsuleA*: cápsula que representa a thread que irá executar em paralelo com a *CapsuleB*.
- *CapsuleB*: cápsula que representa a thread que irá executar em paralelo com a *CapsuleA*.
- *ProtParallelSplit:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventoSplit(void)* o qual representa o evento para indicar que as duas cápsulas *CapsuleA* e *CapsuleB* podem proceder concorrentemente; e do sinal de saída *finished(String)* usado para informar sobre a finalização de cada *branch* ativado.

Diagrama de Estrutura

O diagrama abaixo representa a estrutura da cápsula *Parallel_Split*. Esta cápsula contém as cápsulas *Parallel_Split_Controller, CapsuleA e CapsuleB.* Ela se comunica com o meio externo através da porta *port,* de onde recebe o evento de ativação do componente, que obedece ao protocolo *ProtSequence* e das portas *portA* e *portB* que obedecem ao protocolo *protStructDiscriminator* para indicar o resultado da execução das cápsulas *CapsuleA* e *CapsuleB*, respectivamente, à estrutura *Structured_Discriminator_Controller* compondo o padrão *Structured Discriminator*, que será apresentado mais adiante.

A cápsula *Parallel_Split_Controller* se comunica com as cápsulas *CapsuleA* e *CapsuleB* através das portas *portSplitA* e *portSplitB* que obedecem ao protocolo *ProtParallelSplit*.

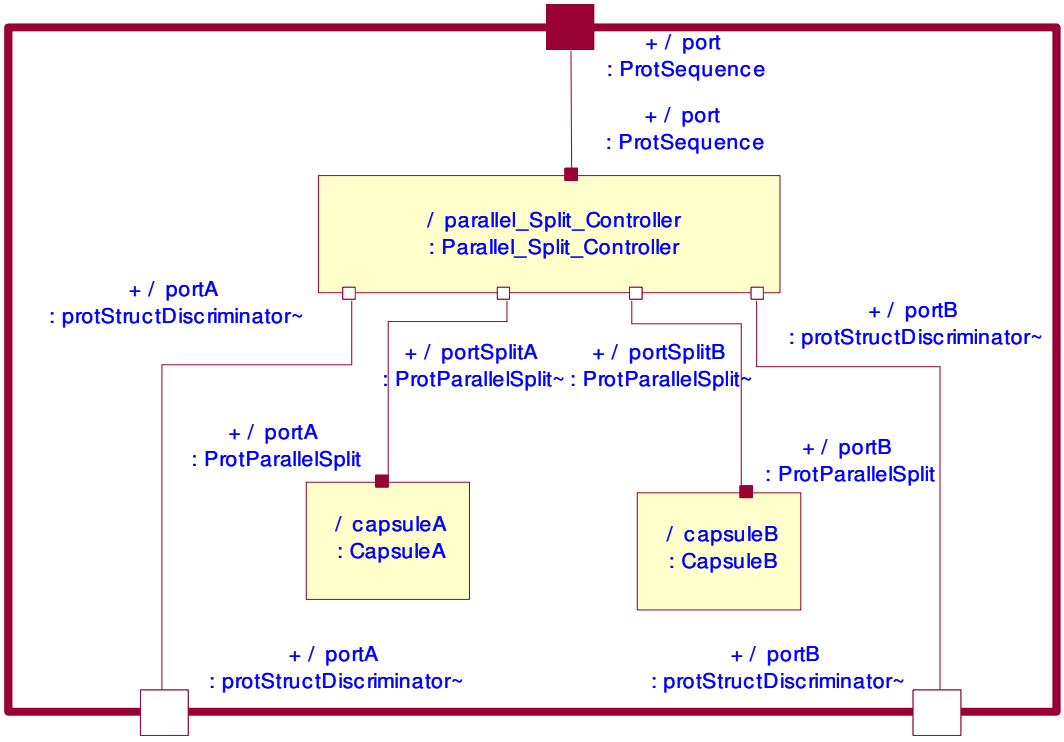


Figura 4. Diagrama de Estrutura da cápsula *Parallel_Split*

Diagrama de Estados

A máquina de estados abaixo define o comportamento do padrão *Parallel Split*, no qual após a execução de uma tarefa, duas *threads* distintas de execução são iniciadas e podem proceder concorrentemente.

No *State1* o contador *count* é inicializado com o valor inteiro '0' e o componente está pronto para receber o *eventoSeq(void)* através da porta *port*. Ao ser notificada do evento, a transição para o *State2* é realizada com o disparo do *eventoSplit(void)* através das portas *portSplitA* e *portSplitB* para que as cápsulas *CapsuleA* e *CapsuleB* iniciem suas tarefas paralelamente.

No *State2* o *Parallel_Split_Controller* aguarda a finalização dos *branches* que foram disparados através da notificação do sinal *finished(String)* recebido de cada cápsula que foi ativada. Na ocorrência deste evento, o controlador informa ao ambiente qual *branch* concluiu a execução e incrementa a variável *count* até que este valor seja igual ao número de *branches* ativados, voltando assim, ao estado inicial.

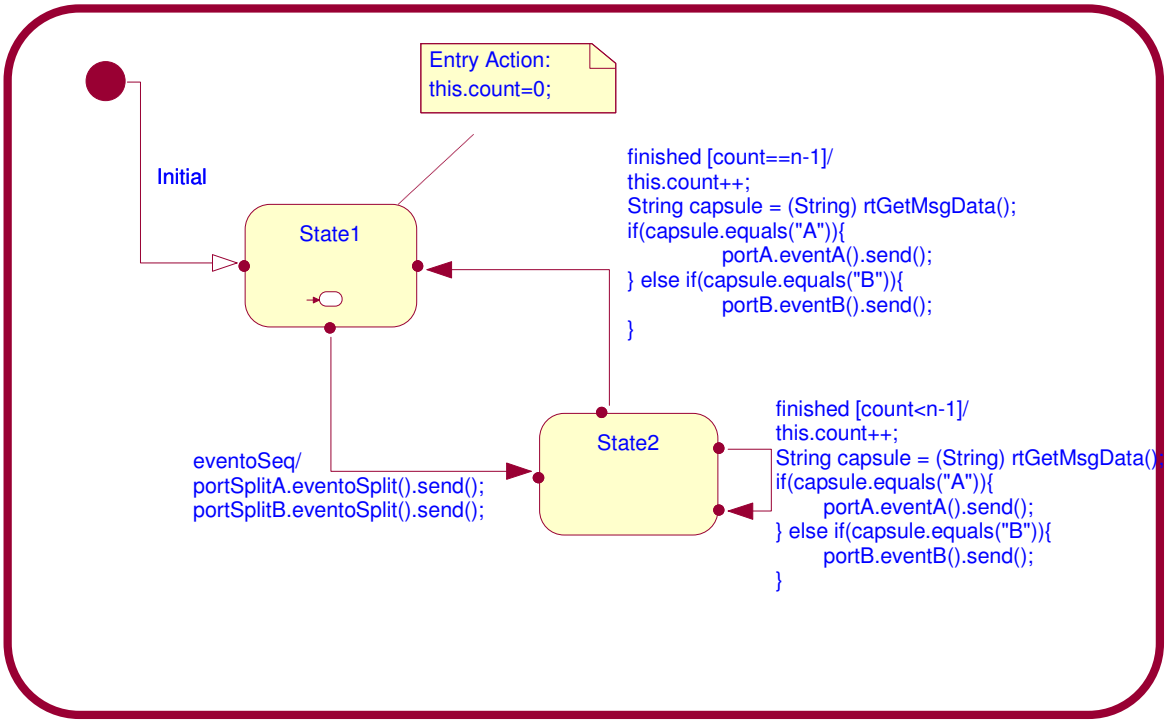
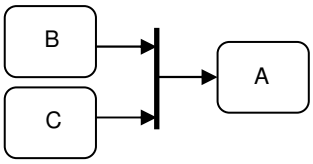


Figura 5. Diagrama de estados da cápsula *Parallel_Split_Controller*

4.1.3 Synchronization

A convergência de dois ou mais *branches* em um único *branch* tal que a thread de controle é passada ao *branch* subseqüente quando todos os *branches* de entrada forem habilitados. [4]



O padrão *Synchronization* ou *AND-join* provê um meio para convergir as threads de execução de dois ou mais *branches* paralelos. Estes *branches* são criados, em geral, usando a construção *Parallel Split* anterior no modelo do processo. A *thread* de controle é passada para a tarefa que segue imediatamente ao sincronizador uma vez que todos os *branches* de entrada forem concluídos.

Exemplo

A *reconciliação do caixa* só pode ocorrer depois que a loja estiver fechada e a fatura do cartão de crédito tenha sido impressa.

Diagrama de Classes

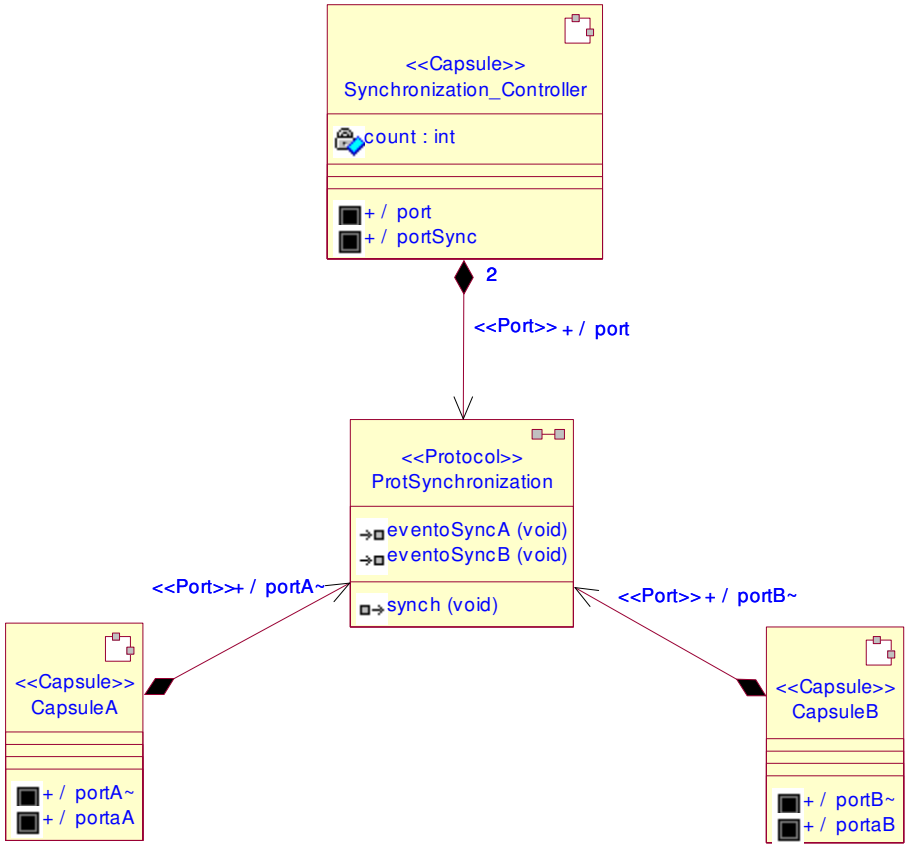


Figura 6. Diagrama de Classes do padrão *Synchronization*

- *Synchronization_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. O fluxo do processo só é continuado após o término da execução de cada thread A e B, as quais são sincronizadas e o fluxo é liberado. O atributo inteiro *count* é utilizado como sincronizador e a política de sincronização definida como a contagem de *branches* que estão prontos para sincronizar até que o número esperado seja alcançado.
- *CapsuleA*: cápsula que representa um fluxo independente anterior à sincronização.
- *CapsuleB*: cápsula que representa um outro fluxo independente anterior à sincronização.
- *ProtSynchronization:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através dos sinais de entrada *eventoSyncA(void)* e *eventoSyncB(void)*, os quais indicam que as cápsulas *CapsuleA* e *CapsuleB*, respectivamente, estão prontas para sincronizar; e do sinal de saída *synch*(*void*) que informa ao ambiente sobre a realização da sincronização.

Diagrama de Estrutura

O diagrama abaixo representa a estrutura da cápsula *Synchronization.* Esta cápsula contém as cápsulas *CapsuleA, CapsuleB* e *Synchronization_Controller.* Ela se comunica com o meio externo através das portas *portA* e *portB* que obedecem ao protocolo *ProtSequence* e *portSync* com o protocolo *protSynchronization*.

As cápsulas *CapsuleA* e *CapsuleB* se comunicam com a *Synchronization_Controller* através das portas *portA, portB* e *port* que obedecem ao protocolo *ProtSynchronization*.

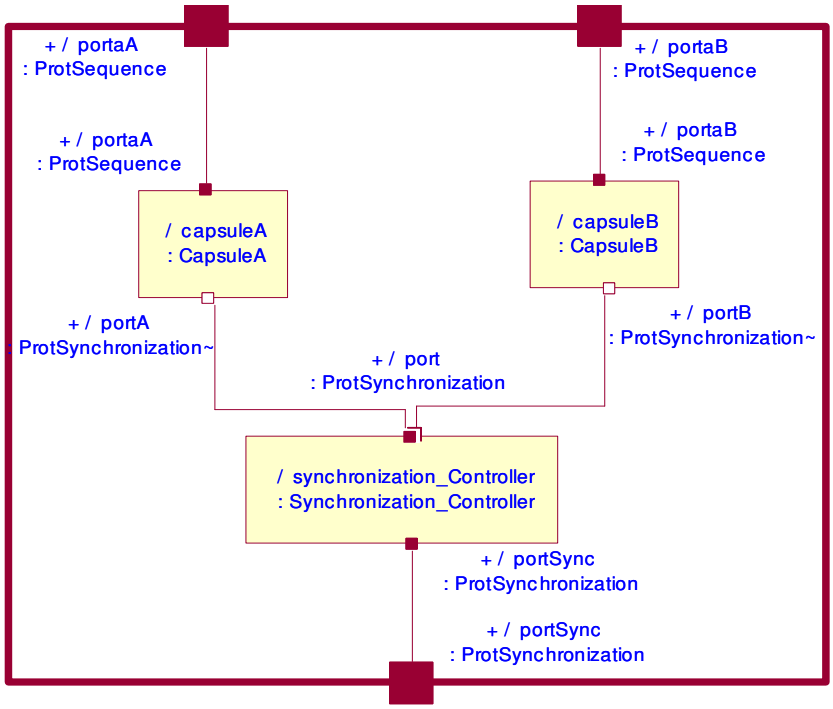


Figura 7. Diagrama de Estrutura da cápsula *Synchronization*

Diagrama de Estados

Este padrão contém um *AND-join* implícito, conhecido como sincronizador que é ativado quando recebe uma entrada em um dos *branches* de entrada e é resetado uma vez que as entradas tenham sido recebidas em todos os *branches* e o sincronizador tenha disparado.

Ele possui a seguinte condição: uma vez que o sincronizador tenha sido ativado e não tenha sido ainda resetado, não é possível que um outro sinal seja recebido no *branch* ativo ou que múltiplos sinais sejam recebidos em qualquer *branch* de entrada.

O comportamento descrito é realizado pelo seguinte diagrama de estados que possui um estado inicial *State1* o qual inicializa a variável *count* com o valor '0' e aguarda uma notificação de um *branch* que esteja pronto para sincronizar. Ao receber um *eventoSyncA(void)* ou *eventoSyncB(void)* (através da porta *port* - em UML-RT mais de um sinal pode ser recebido através da mesma porta), o contador é incrementado e o controlador vai para o *State2*.

Neste estado, novos eventos de sincronização são aguardados e a cada notificação o contador é incrementado. Ao receber a última notificação de sincronização, o fluxo é liberado, o evento *synch*(*void*) é disparado e o controlador volta ao seu estado inicial.

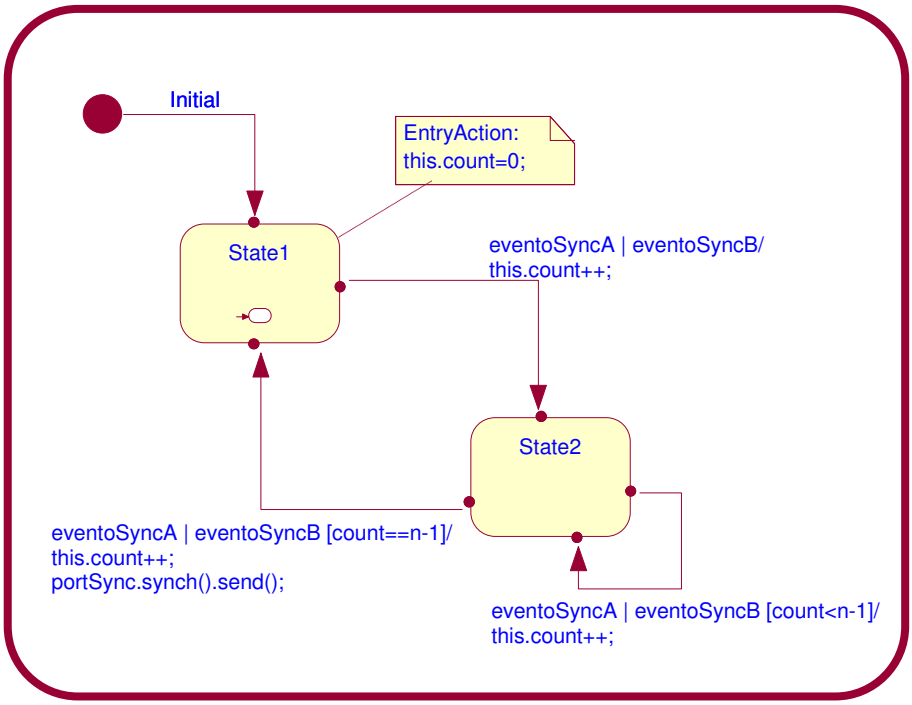
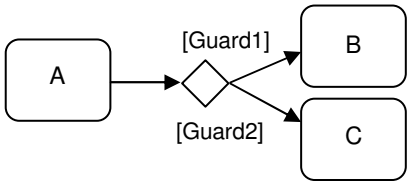


Figura 8. Diagrama de Estados da cápsula *Synchronization_Controller*

4.1.4 Exclusive Choice

A divergência de um *branch* em dois ou mais tais que quando o *branch* de entrada é habilitado, a *thread* de controle é imediatamente passada para precisamente um dos *branches* de saída baseado em um mecanismo para selecionar um dos *branches* de saída. [4]



O padrão *Exclusive Choice* ou *XOR-split* permite que a thread de controle seja direcionada a uma tarefa específica dependendo da tarefa anterior, dos valores de dados específicos do processo, dos resultados de uma avaliação de expressão ou algum outro mecanismo de seleção programado. A decisão do caminho a ser seguido é feita dinamicamente permitindo que esta seja postergada até o último momento durante a execução.

Exemplo

Após a tarefa *revisar eleição* completar, será executada a tarefa *declarar resultados* ou *recontar votos*.

Diagrama de Classes

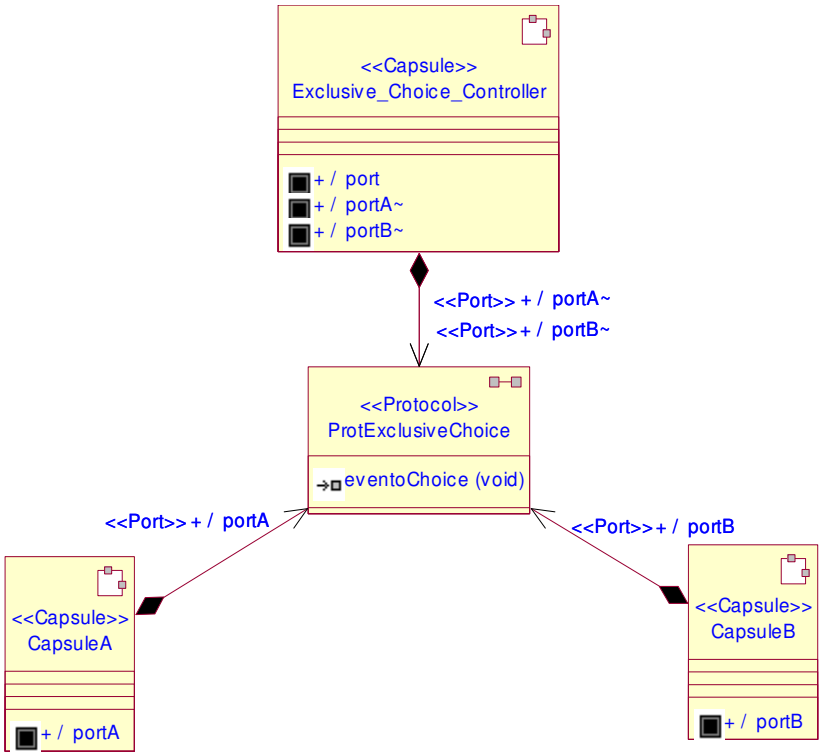


Figura 9. Diagrama de Classes do padrão *Exclusive Choice*

- *Exclusive_Choice_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. De acordo com uma pré-condição avaliada no momento da escolha, somente uma das cápsulas *CapsuleA* ou *CapsuleB* será ativada e seguirá com o fluxo do processo.
- *CapsuleA*: cápsula que representa uma possível escolha de fluxo do processo.
- *CapsuleB*: cápsula que representa uma outra possível escolha de fluxo do processo.
- *ProtExclusiveChoice:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventoChoice(void)* que indica qual das cápsulas foi escolhida para continuar o fluxo.

Diagrama de Estrutura

O diagrama abaixo representa a estrutura da cápsula *Exclusive_Choice.* Esta cápsula contém as cápsulas *Exclusive_Choice_Controller, CapsuleA* e *CapsuleB.* Ela se comunica com o meio externo através da porta *port* que obedece ao protocolo *ProtSequence.* Um sinal recebido pela porta *portSeq* indica que um componente anterior a este fluxo foi concluído este componente deverá iniciar sua execução.

A cápsula *Exclusive_Choice_Controller* se comunica com as cápsulas *CapsuleA* e *CapsuleB* através das portas *portA* e *portB* que obedecem ao protocolo *ProtExclusiveChoice*.

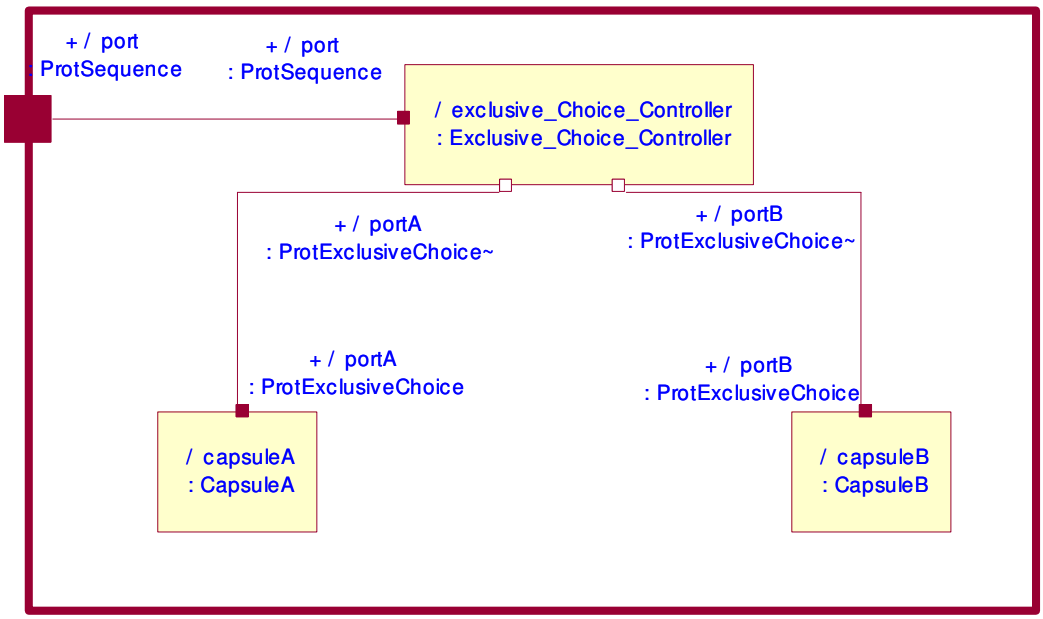


Figura 10. Diagrama de Estrutura da cápsula *Exclusive_Choice*

Diagrama de Estados

Dependendo dos resultados da expressão condicional avaliada, a thread de controle é roteada para a cápsula *CapsuleA* ou *CapsuleB*. Há uma condição de contexto associada com este padrão: o mecanismo que avalia o *Exclusive Choice* pode acessar quaisquer elementos de dados ou outros recursos necessários na determinação de para qual dos *branches* de saída a thread de controle deve ser passada.

No estado inicial *State1* o componente está pronto para executar e recebe um *eventoSeq(void)* indicando que um processo anterior finalizou e este pode iniciar. Após a transição há um *Choice Point* onde, através de um mecanismo de avaliação, será decidido qual *branch* será escolhido. Em cada transição há o envio de um *eventoChoice(void)* para apenas uma das cápsulas, através das portas *portA* ou *portB*, no caso da seleção da *CapsuleA* ou *CapsuleB*, respectivamente. O componente volta então ao seu estado inicial esperando o próximo evento para executar.

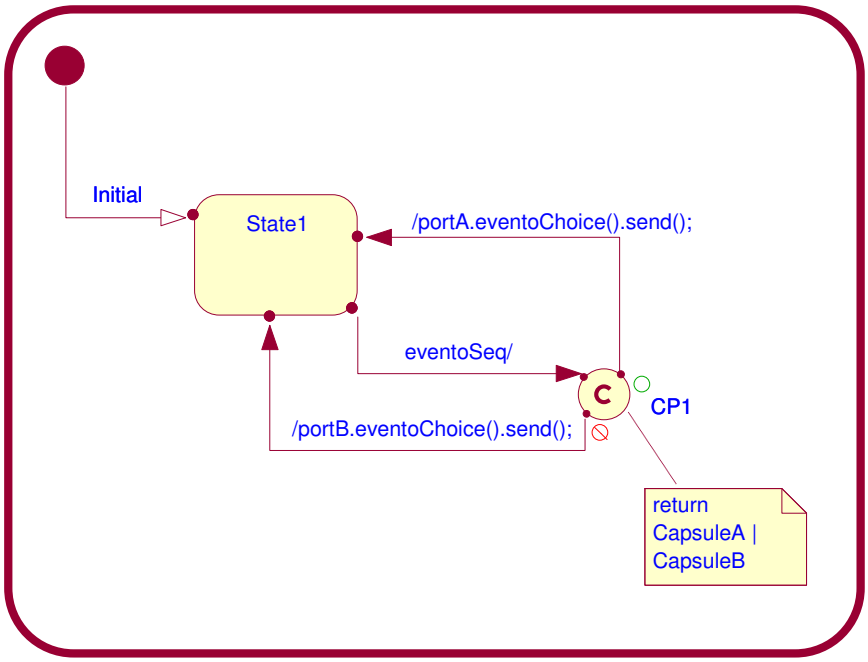
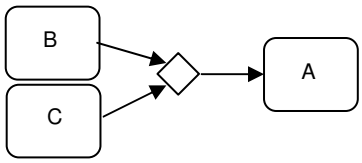


Figura 11. Diagrama de Estados da cápsula *Exclusive_Choice_Controller*

Simple Merge

A convergência de dois ou mais *branches* em um único *branch* subseqüente tal que cada habilitação de um *branch* de entrada resulta na passagem da *thread* de controle para o *branch* subseqüente. [4]



O padrão *Simple Merge* ou *XOR-join* provê um meio de unir um ou mais *branches* distintos sem sincronizá-los. Assim, ele se apresenta como uma oportunidade de simplificar um modelo de processo removendo a necessidade de explicitamente replicar uma seqüência de tarefas comum a dois ou mais *branches*. Ao invés disso, estes *branches* podem ser unidos com a construção *Simple Merge* e o conjunto comum de tarefas necessita ser descrito apenas uma vez no modelo do processo.

Exemplo

Após as tarefas *pagamento* ou *provisão de crédito*, inicia-se a tarefa *receber produto*.

Diagrama de Classes

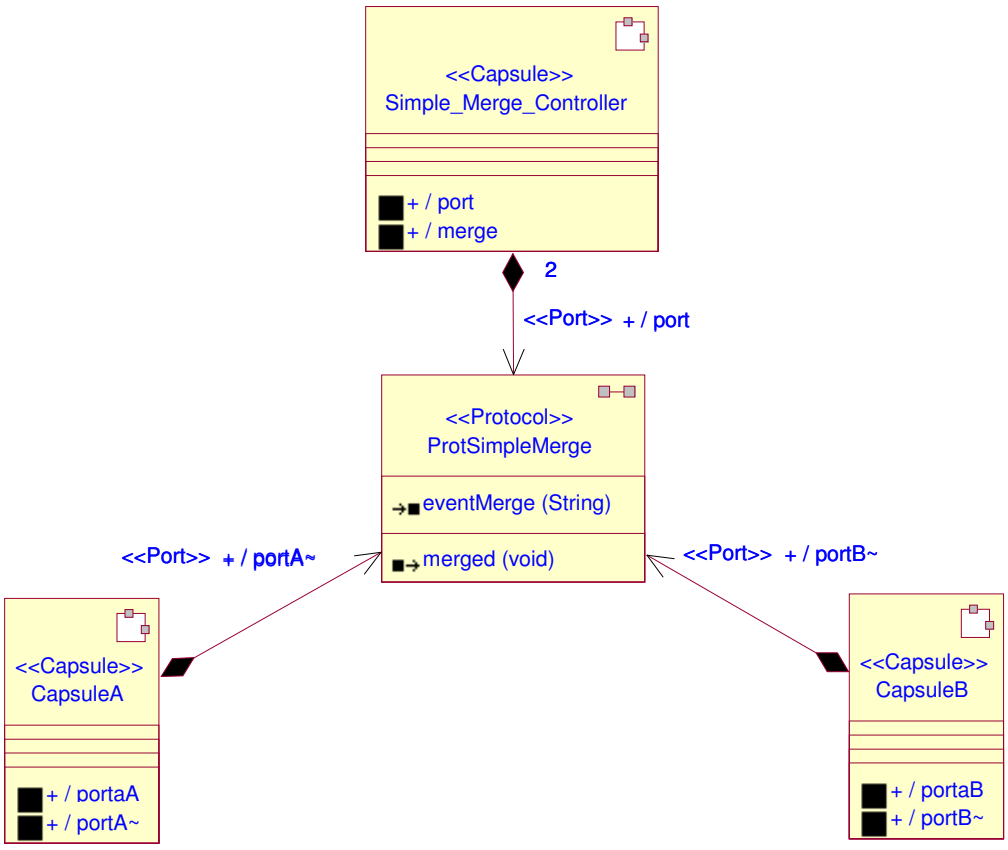


Figura 12. Diagrama de Classes do padrão *Simple Merge*

- *Simple_Merge_Controller:* é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. À medida que um branch de entrada é habilitado, a thread de controle é passada ao branch subseqüente, que representa o mesmo fluxo de atividades para qualquer branch de entrada.
- *CapsuleA*: cápsula representando um fluxo que antecede à junção em um fluxo subseqüente.
- *CapsuleB*: cápsula representando um fluxo que antecede à junção em um fluxo subseqüente.
- *ProtSimpleMerge:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventMerge(String)* que indica quando um *branch* de entrada está pronto para seguir o próximo fluxo do processo; e do sinal de saída *merged(void)* para informar ao ambiente sobre a realização do *merge*.

Diagrama de Estrutura

O diagrama a seguir representa a estrutura da cápsula *Simple_Merge.* Esta cápsula contém as cápsulas *CapsuleA, CapsuleB* e *Simple_Merge_Controller.* Ela se comunica com o meio externo através da porta *port* que obedece ao protocolo *ProtSequence* e da porta *merge* com o protocolo *ProtSimpleMerge*.

Um sinal recebido pela porta *portSeq* indica que um componente anterior a este fluxo foi concluído este componente deverá iniciar sua execução. O sinal enviado pela porta *merge* ao próximo componente indica que este componente terminou sua execução.

As cápsulas *CapsuleA* e *CapsuleB* se comunicam com a cápsula *Simple_Merge_Controller* através das portas *portA* e *portB* que obedecem ao protocolo *ProtSimpleMerge*.

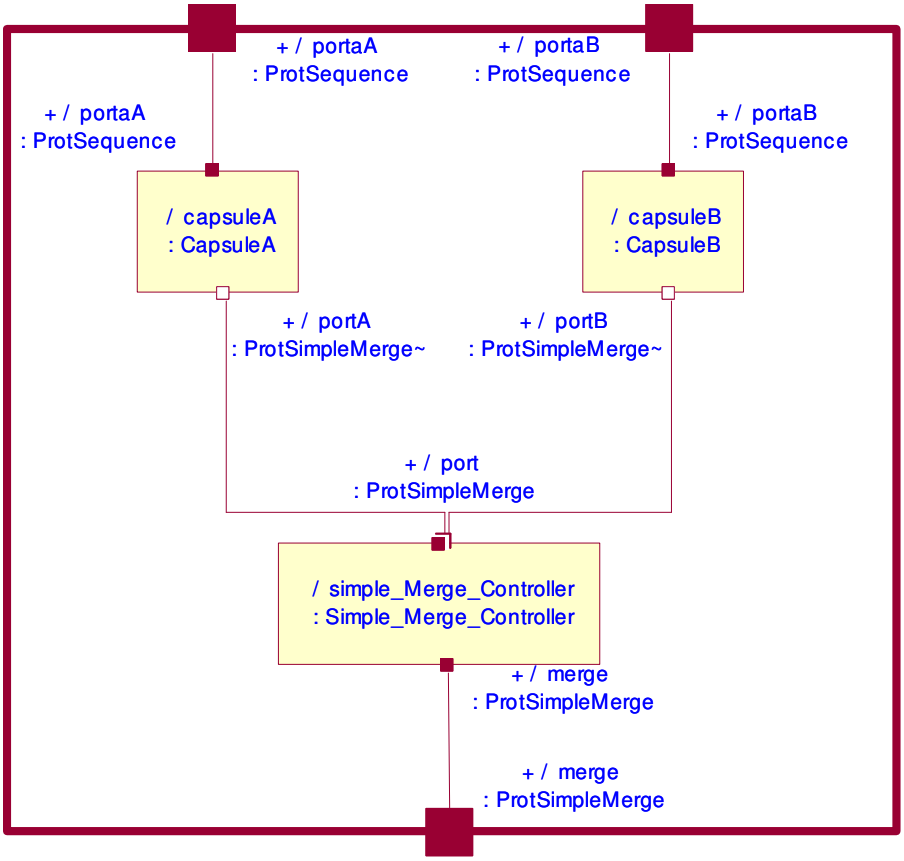


Figura 13. Diagrama de Estrutura da cápsula *Simple_Merge*

Diagrama de Estados

Imediatamente após a cápsula (*CapsuleA* ou *CapsuleB*) tiver completado, a terceira cápsula será habilitada. Não há consideração de sincronização. Há uma condição de contexto associada ao padrão: o ponto onde ocorre a junção não deve ser habilitado por mais de uma cápsula ao mesmo tempo.

No estado inicial a cápsula está pronta para executar à espera de um *eventoMerge(String)*. Apenas uma das cápsulas antecedentes estará ativa e irá disparar o *eventMerge(String)*. Após ser disparado o evento, a cápsula irá enviar um sinal *merged(void)* através da porta *merge* e retornará ao estado inicial *State1* para receber um novo sinal de *eventMerge(String)*.

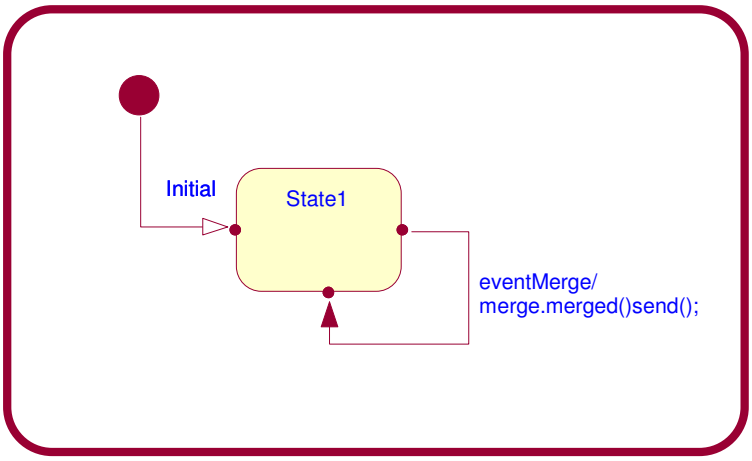


Figura 14. Diagrama de Estados da cápsula *Simple_Merge_Controller*

4.2 Advanced Branching and Synchronization Patterns

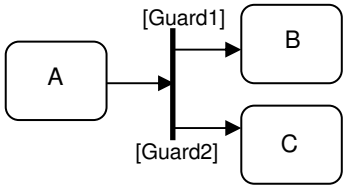
Nesta seção serão apresentados os padrões que caracterizam conceitos mais complexos de *branching* e *merging* que ocorrem em processos de negócio. Os padrões de *Control Flow* originais identificaram quatro padrões, a saber: *Multi-Choice, Synchronizing Merge, Multi-Merge* e *Discriminator.*

Para alguns destes padrões, no entanto, foram identificadas diversas maneiras distintas pelas quais eles podem funcionar. O *Synchronizing Merge* original é agora a base para os padrões *Structured Synchronizing Merge, Acyclic Synchronizing Merge e General Synchronizing Merge*. Da mesma forma, o padrão original *Discriminator* foi dividido em seis padrões distintos: *Structured Discriminator, Blocking Discriminator, Cancelling Discriminator, Structured Partial Join, Blocking Partial Join* e *Cancelling Partial Join*.

Dentre estes padrões, as descrições originais do *Synchronizing Merge* e do *Discriminator* são suplantadas pelas definições *Structured Synchronizing Merge* e *Structured Discriminator*. Dessa forma, apresentaremos os quatro padrões originalmente identificados da forma como estão definidos atualmente pela iniciativa *Workflow Patterns*.

4.2.1 Multi Choice

A divergência de um *branch* em dois ou mais tal que quando o *branch* de entrada é habilitado, a thread de controle é imediatamente passada para um ou mais *branches* de saída baseado em um mecanismo que seleciona um ou mais *branches* de saída. [4]



O padrão *Multi Choice* ou OR-split provê a capacidade à *thread* de execução divergir em diversas *threads* concorrentes com base em uma seleção. A decisão de para qual *branch* a thread de execução será passada é feita em tempo de execução e pode ser baseada em diversos fatores incluindo o término de uma tarefa precedente, valores de dados específicos do processo, resultados de uma avaliação de expressão associada ao *branch* de saída ou algum outro mecanismo de seleção programado. Este padrão é essencialmente análogo ao *Exclusive Choice,* no qual diferentes *branches* de saída podem ser habilitados.

Exemplo

Dependendo da natureza da chamada de emergência, uma ou mais das tarefas *enviar polícia*, *enviar bombeiros* ou *enviar ambulância* podem ser imediatamente iniciadas.

Diagrama de Classes

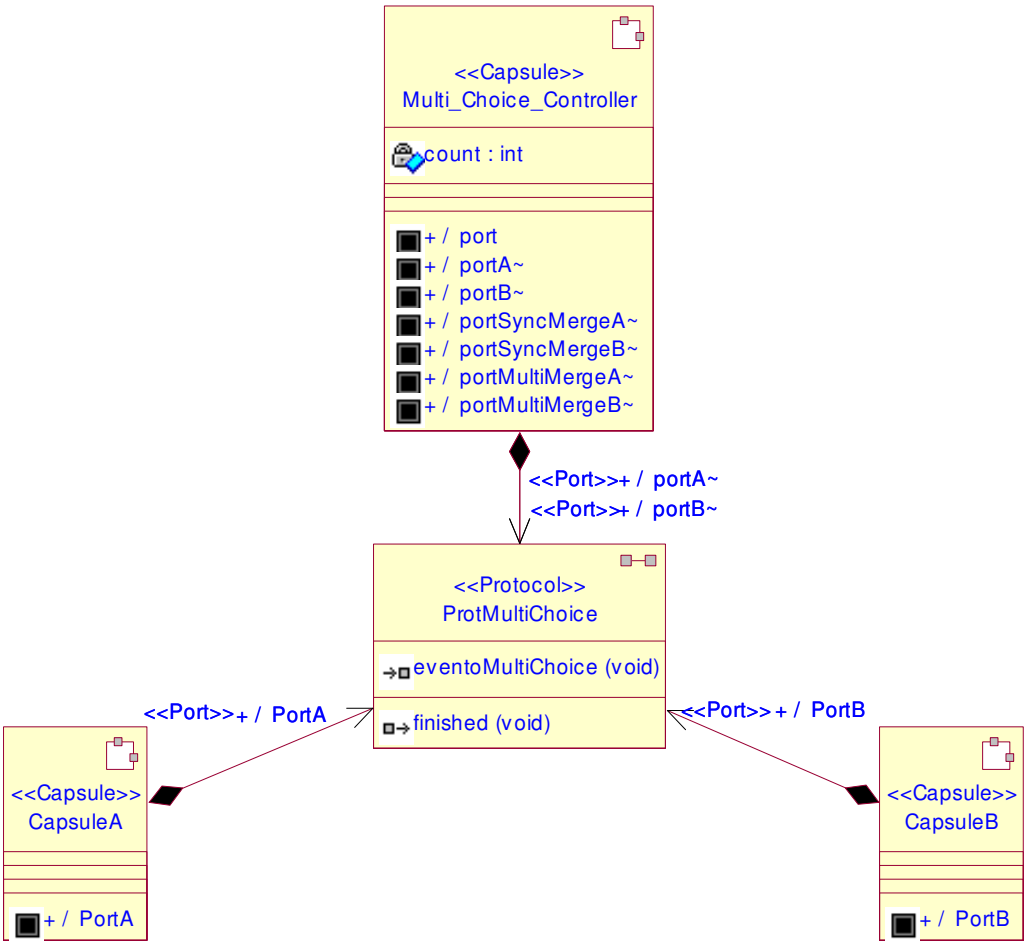


Figura 15. Diagrama de Classes do padrão *Multi Choice*

- *Multi_Choice_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. De acordo com uma pré-condição avaliada no momento da escolha, uma das cápsulas *CapsuleA* ou *CapsuleB* ou ambas serão ativadas e seguirão com o fluxo do processo. A variável inteira *count* é utilizada para indicar quantas das cápsulas que foram ativadas já terminaram suas atividades.
- *CapsuleA*: cápsula que representa uma possível escolha de fluxo do processo.
- *CapsuleB*: cápsula que representa uma outra possível escolha de fluxo do processo.
- *ProtMultiChoice:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventoMultiChoice(void)* que indica qual ou quais das cápsulas foi(foram) escolhida(s) para continuar(em) o fluxo. O sinal de saída *finished*(*void*) é utilizado para indicar a finalização da execução de cada *branch* habilitado para o controlador.

Diagrama de Estrutura

O diagrama abaixo representa a estrutura da cápsula *Multi_Choice.* Esta cápsula contém o *Multi_Choice_Controller, CapsuleA* e *CapsuleB.*

Ela se comunica com o meio externo através da porta *port* que obedece ao protocolo *ProtSequence,* das portas *portSyncMergeA* e *portSyncMergeB* que obedecem ao protocolo *ProtSynchronization* e das portas *portMultiMergeA* e *portMultiMergeB* que obedecem ao protocolo *ProtMultiMerge.* Estas últimas quatro portas são utilizadas para a comunicação deste elemento com as cápsulas pertencentes a outros padrões subseqüentes formados pela composição deste padrão com novos componentes.

A cápsula *Multi_Choice_Controller* se comunica com as cápsulas *CapsuleA* e *CapsuleB* através das portas *portA* e *portB* que obedecem ao protocolo *ProtMultiChoice*.

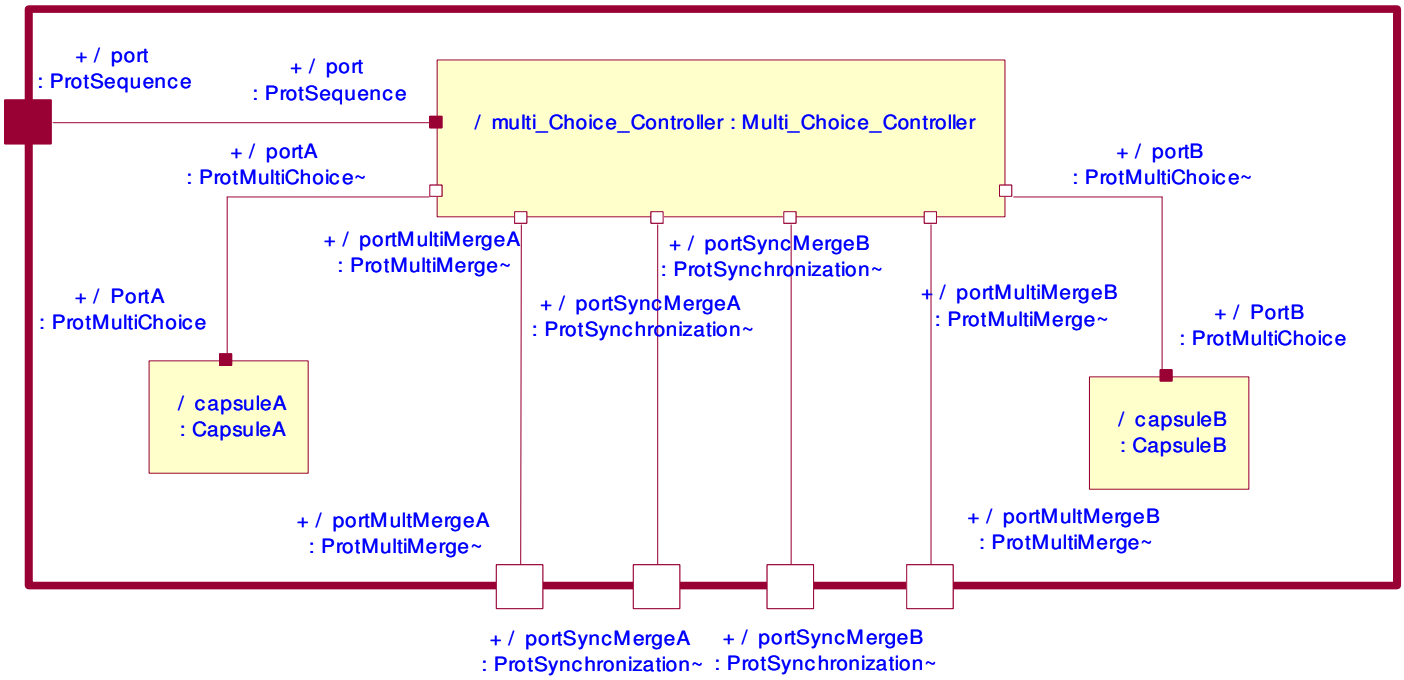


Figura 16. Diagrama de Estrutura da cápsula *Multi_Choice*

Diagrama de Estados

Após uma tarefa ser disparada, a thread de controle pode ser passada para um ou ambos os *branches* seguintes dependendo da avaliação das condições associadas a cada um deles. Há uma condição de contexto associada a este padrão: o mecanismo que avalia o *Multi Choice* pode acessar quaisquer dados ou recursos necessários na determinação de para qual dos *branches* de saída a thread de controle deve ser repassada.

No estado inicial, o componente está pronto para executar. Ao ser disparado um *eventoSeq(void)* será feita uma avaliação de qual ou quais *branches* serão ativados: A, B ou ambos. Assim temos três resultados possíveis desta avaliação. Como em UML-RT temos a construção *Choice Point* que permite apenas dois resultados de saída, fizemos uma composição de dois *Choice Points* onde o

primeiro avalia se o *branch* escolhido foi a *CapsuleA*, se não foi, o próximo *Choice Point* irá avaliar se o *branch* escolhido foi a *CapsuleB*, se não, é porque ambos os *branches* foram escolhidos.

Se a avaliação realizada no primeiro *Choice Point* resultar em uma expressão verdadeira, a *CapsuleA* será ativada com o sinal *eventoMultiChoice(void)* enviado através da porta *portA*. Além deste sinal, é enviado o sinal *eventoSyncB(void)* através da porta *portSyncMergeB* que será recebido pelos elementos do padrão *Structured Synchronizing Merge* na composição com o padrão *Multi Choice*. A razão do envio deste sinal será esclarecida na descrição do padrão *Structured Synchronizing Merge* adiante.

Se a avaliação do primeiro *Choice Point* resultar em uma expressão falsa, o segundo *Choice Point* irá avaliar se o *branch* escolhido foi o da *CapsuleB*. Se sim, o sinal *eventoMultiChoice(void)* será enviado à *CapsuleB* através da porta *portB* e o sinal *eventoSyncA(void)* através da porta *portSyncMergeA*. A razão deste último sinal será conhecida na descrição do padrão *Structured Synchronizing Merge* adiante.

Se a avaliação do segundo *Choice Point* resultar em uma expressão falsa, significa que os dois *branches* devem ser ativados e por isso o sinal *eventoMultiChoice(void)* será enviado para ambas as cápsulas *CapsuleA* e *CapsuleB* através das portas *portA* e *portB*, respectivamente.

De acordo com a cápsula que foi ativada, o controlador ficará em um dos estados *StateA*, *StateB* ou *ABSynchronization* esperando que cada uma delas termine. Se apenas a cápsula *CapsuleA* estiver ativa, o controlador ficará no estado *StateA* até que receba uma notificação do sinal *finished(void)* de que sua execução terminou. Assim os eventos *eventoSyncA* e *eventMerge("CapsuleA")* serão enviados ao ambiente através das portas *portSyncMergeA* e *portMultiMergeA*, para os elementos dos padrões *Synchronizing Merge* e *Multi Merge*, respectivamente.

Se apenas a cápsula *CapsuleB* estiver ativa, o controlador ficará no estado *StateB* até que receba uma notificação do sinal *finished(void)* de que sua execução terminou. Assim os eventos *eventoSyncB* e *eventMerge("CapsuleB")* serão enviados ao ambiente através das portas *portSyncMergeB* e *portMultiMergeB*, para os elementos dos padrões *Synchronizing Merge* e *Multi Merge*, respectivamente.

Se ambas as cápsulas estiverem ativas, o controlador ficará esperando a finalização de cada uma, incrementando a variável *count* até que o número total de cápsulas ativas tenha terminado de executar. Ao final, os sinais *eventoSyncA(void)* e *eventoSyncB(void)* serão enviados através das portas *portSyncMergeA* e *portSyncMergeB*, respectivamente, e os sinais *eventMerge("CapsuleA")* e *eventMerge("CapsuleB")* através das portas *portMultiMergeA* e *portMultiMergeB*, respectivamente. Estes sinais são utilizados nos padrões *Synchronizing Merge* e *Multi Merge* apresentados a seguir.

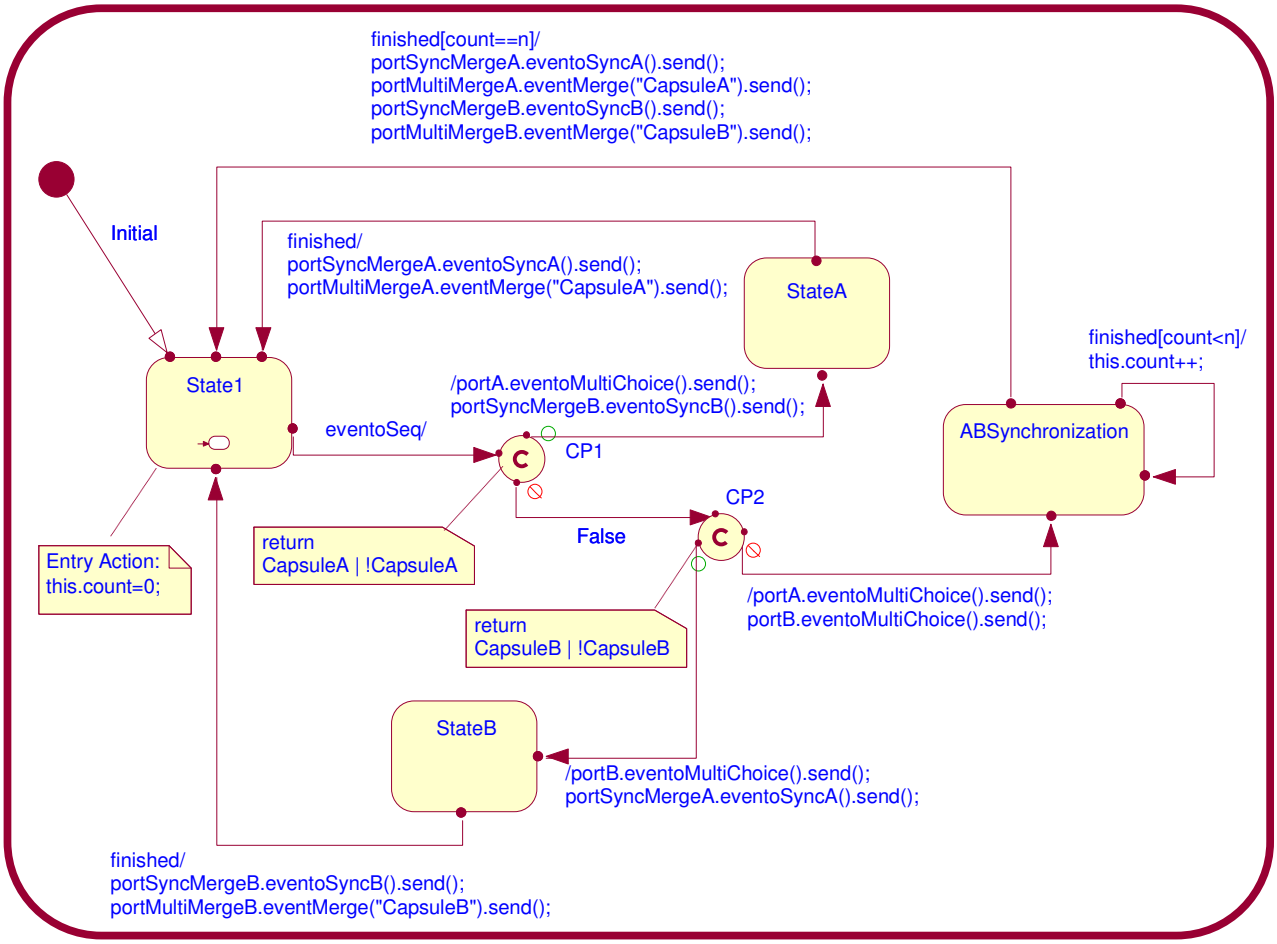
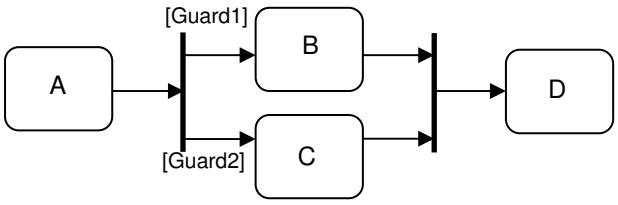


Figura 17.Diagrama de Estados da cápsula *Multi_Choice_Controller*

4.2.2 *Structured Synchronizing Merge*

A convergência de dois ou mais *branches* (que divergiram anteriormente no processo em um ponto identificável) em um único *branch* subseqüente de forma que a thread de controle é passada ao *branch* subseqüente quando cada *branch* ativo de chegada tiver sido habilitado. [4]



O padrão *Structured Synchronizing Merge* ou *Synchronizing Join* provê um meio de unir os *branches* resultantes de um *Multi Choice* (ou *OR-split*) prévio específico do processo em um único *branch*. Implicitamente nessa junção, há uma sincronização de todas as *threads* de execução resultantes do *Multi Choice* precedente.

Não é necessário que todos os *branches* que chegam ao *Structured Synchronizing Merge* estejam ativos para a construção ser habilitada, no entanto, todas as threads de controle associadas aos *branches* de entrada devem ter alcançado o *Structured Synchronizing Merge* antes que ele possa disparar.

Uma das dificuldades deste padrão é saber quando o *Structured Synchronizing Merge* pode disparar e ele deve ser capaz de resolver essa questão baseado em informação local disponível em tempo de execução.

Isto pode ser resolvido de três maneiras: (1) estruturando o modelo de processo seguinte à construção *Multi Choice* de forma que o *Structured Synchronizing Merge* sempre irá receber precisamente um *trigger* em cada um de seus *branches* de entrada; (2) fornecendo à construção do *Merge* o conhecimento do número de *branches* que requer sincronização; (3) fazendo uma meticulosa análise de possíveis estados futuros de execução para determinar quando o *Structured Synchronizing Merge* pode disparar.

A primeira das opções é implementada no padrão sendo descrito. Esta abordagem requer a adição de um "*bypass*" em torno de cada *branch* do *Multi Choice* para o *Structured Synchronizing Merge* que é habilitado quando o respectivo *branch* não é escolhido. Dessa forma, a cápsula *Multi_Choice_Controller* envia um sinal *portSyncMergeB(void)* quando apenas o *branch* A for ativado ou *portSyncMergeA(void)* quando apenas o *branch* B for ativado. Isto garante que o *Structured Synchronizing Merge* sempre receberá um *trigger* em cada um de seus *branches* de entrada e pode ser, portanto, implementado como um *AND-join* reutilizando o componente *Synchronization_Controller* do padrão *Synchronization*.

Exemplo

Dependendo do tipo de emergência, uma ou ambas as tarefas *enviar polícia* e *enviar ambulância* são iniciadas simultaneamente. Quando todos os veículos de emergência chegarem ao acidente, a tarefa *transferir paciente* é iniciada.

Diagrama de Classes

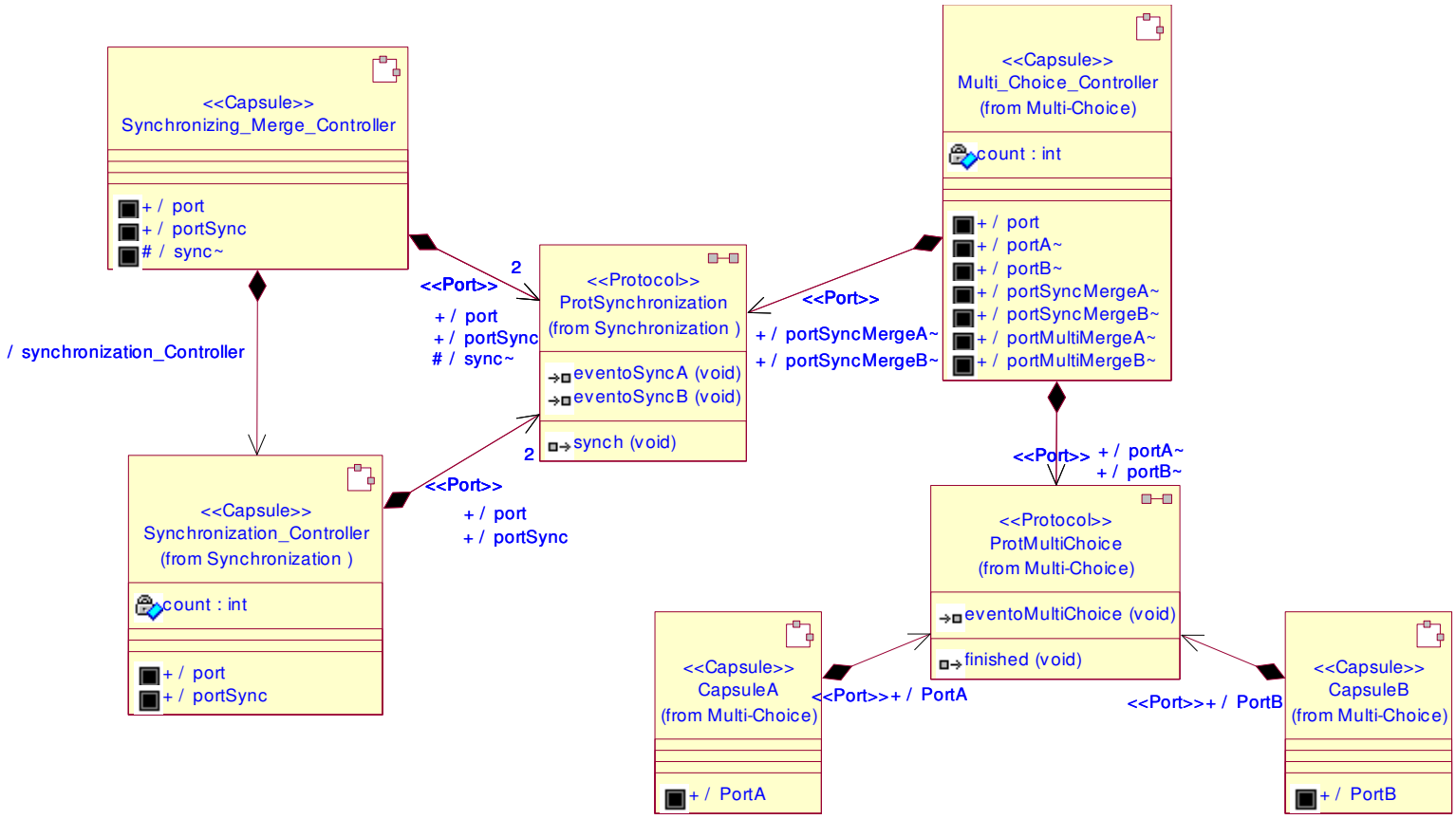


Figura 18. Diagrama de Classes do padrão *Structured Synchronizing Merge*

- *Synchronizing_Merge_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. O fluxo do processo só é continuado após o término da execução de cada thread que flui da construção *Multi_Choice*, as quais são sincronizadas e o fluxo é liberado. Mesmo que um *branch* do *Multi_Choice* não tenha sido habilitado, ele envia um sinal para o módulo *Synchronizing_Merge*, dessa forma, esta cápsula sempre receberá um *trigger* em cada uma das suas portas de entrada.
- *Synchronization_Controller*: é uma subcápsula da *Synchronizing_Merge_Controller* e é reutilizada para implementar o comportamento *AND – join* na sincronização dos *branches* de entrada.
- *ProtSynchronization:* protocolo usado para comunicar as cápsulas que colaboram no padrão através dos sinais de entrada *eventoSyncA(void)* e *eventoSyncB(void)*, os quais indicam ao controlador que as cápsulas *CapsuleA* e *CapsuleB*, respectivamente, estão prontas para sincronizar. O sinal de saída *synch(void)* serve para notificar que a sincronização foi terminada com sucesso.

- *Multi_Choice_Controller*: módulo que antecede o *Synchronizing_Merge* do qual fluem os *branches* que irão sincronizar no módulo *Synchronizing_Merge.*
- *CapsuleA*: cápsula que representa uma possível escolha de fluxo do processo.
- *CapsuleB*: cápsula que representa uma outra possível escolha de fluxo do processo.
- *ProtMultiChoice:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventoMultiChoice(void)* que indica qual ou quais das cápsulas foi(foram) escolhida(s) para continuar(em) o fluxo. O sinal de saída *finished*(*void*) é utilizado para indicar a finalização da execução de cada *branch* habilitado para o controlador.

Diagrama de Estrutura

O diagrama abaixo representa a estrutura da cápsula *Synchronizing_Merge.* Esta cápsula contém as cápsulas *Multi_Choice_Controller, CapsuleA, CapsuleB* e *Synchronizing_Merge_Controller.*

Ela se comunica com o meio externo através das portas *port* e *sync* que obedecem ao protocolo *ProtSequence*.

A cápsula *Multi_Choice_Controller* se comunica com as cápsulas *CapsuleA* e *CapsuleB* através das portas *portA* e *portB*, respectivamente. E se comunica com a cápsula *Synchronizing_Merge_Controller* através das portas *portSyncMergeA* e *porSyncMergeB* que obedecem ao protocolo *ProtSynchronization*.

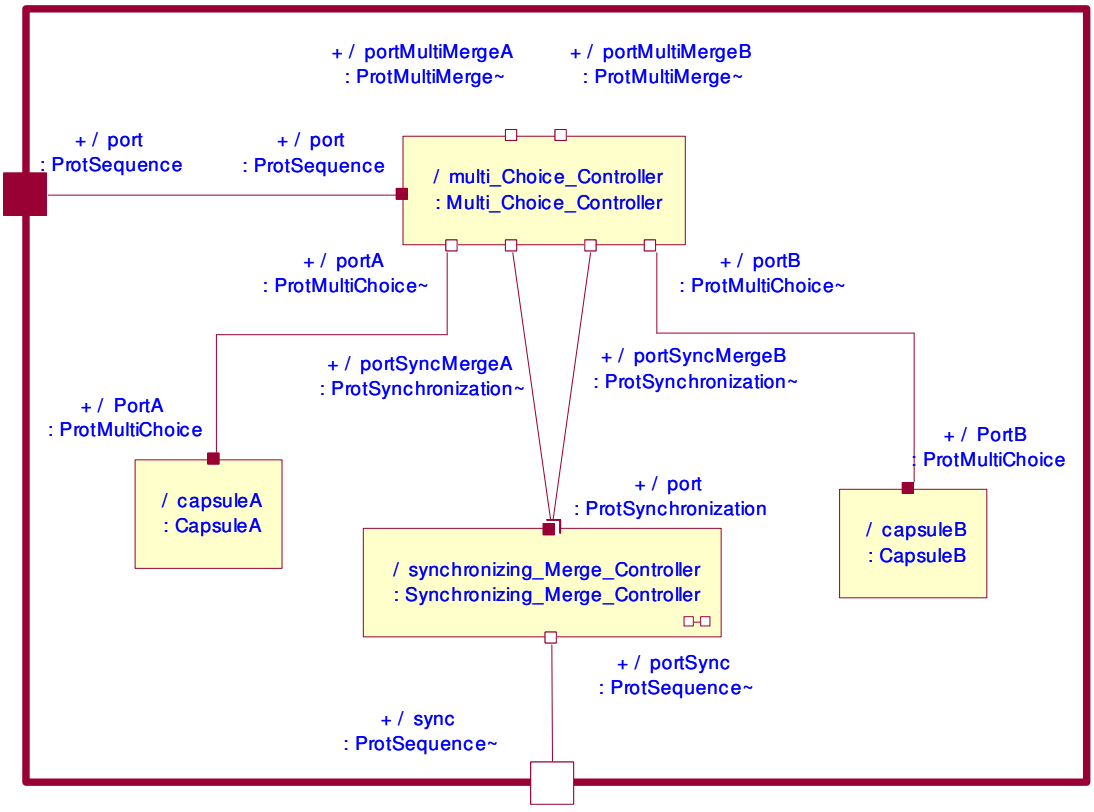


Figura 19. Diagrama de Estrutura da cápsula *Synchronizing_Merge*

A cápsula *Synchronizing_Merge_Controller* possui a estrutura mostrada a seguir. Ela contém a cápsula *Synchronization_Controller* que implementa o comportamento de sincronização necessário à realização deste padrão. O sinal recebido pela porta *port* do tipo *end*, é repassado diretamente ao componente *Synchronization_Controller* que notifica através da porta *portSync* o término da sincronização.

Para que o *Synchronizing_Merge_Controller* tenha conhecimento desta notificação, utilizamos a porta conjugada *sync* que é do tipo *relay*, para se comunicar com a máquina de estados do componente *Synchronizing_Merge_Controller*. A porta *portSync* é a porta de saída que se liga diretamente ao ambiente externo para comunicar a finalização do processo executado.

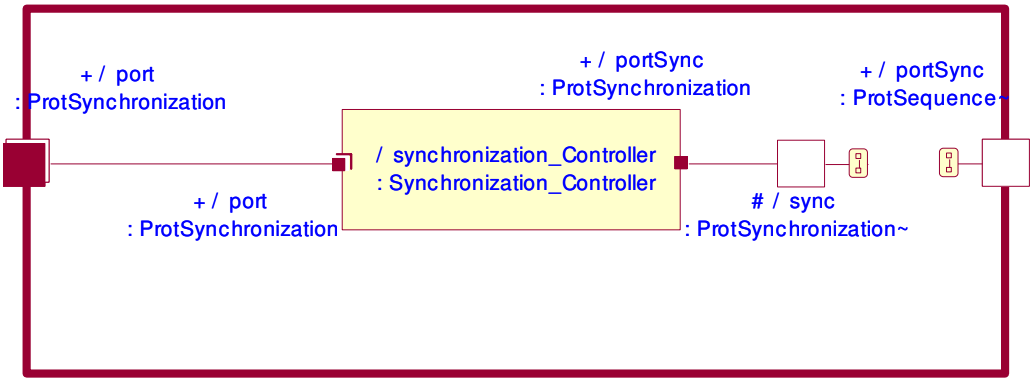


Figura 20. Diagrama de Estrutura da Cápsula *Synchronizing_Merge_Controller*

Diagrama de Estados

Este padrão está estruturado neste trabalho de forma que o *Structured Synchronizing Merge* subseqüente ao *Multi Choice* sempre recebe um *trigger* em cada um dos *branches* de entrada.

Há duas condições de contexto associadas ao uso deste padrão:

1. Uma vez que o *Structured Synchronizing Merge* foi ativado e ainda não foi resetado, não é possível que outro sinal seja recebido no *branch* ativo ou que múltiplos sinais sejam recebidos em quaisquer dos *branches* de entrada.
2. Uma vez que o *Multi Choice* foi habilitado nenhuma das tarefas nos *branches* que se ligam ao *Structured Synchronizing Merge* pode ser cancelada. A exceção é a possibilidade de todas as tarefas que se ligam ao *Structured Synchronizing Merge* serem canceladas.

Assim, o diagrama possui apenas um estado no qual será recebido o *trigger* da *Synchronization_Controller* através da porta *sync* e então um sinal *eventoSeq(void)* é enviado através da porta *portSync* para fora da estrutura.

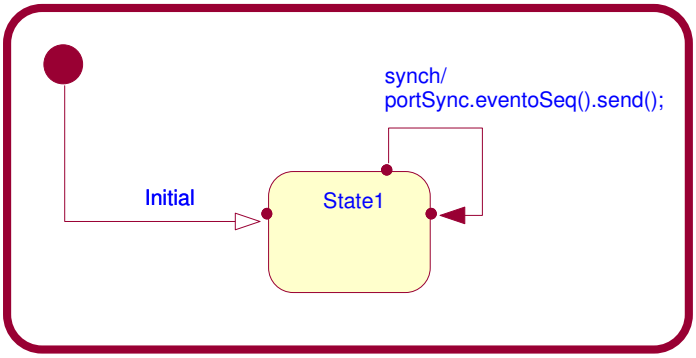
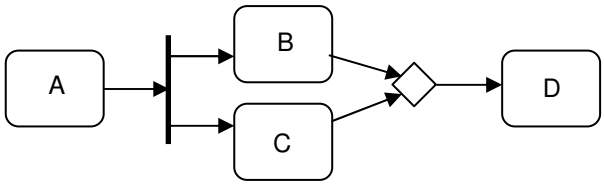


Figura 21. Diagrama de Estados da cápsula *Synchronizing_Merge_Controller*

4.2.3 Multi Merge

A convergência de dois ou mais *branches* em um único *branch* subseqüente tal que cada habilitação de um branch de entrada resulta na passagem da *thread* de controle ao *branch* subseqüente. [4]



O padrão *Multi Merge* provê um meio de unir *branches* distintos de um processo em um único *branch*. Embora vários caminhos de execução sejam unidos, não há sincronização do fluxo de controle e cada *thread* de controle que está atualmente ativo em quaisquer dos *branches* precedentes irá fluir livremente para o *branch* subseqüente.

Exemplo

As tarefas *fazer fundação*, *pedir materiais* e *alocar trabalhadores* ocorrem em paralelo como *branches* separados do processo. Após o final de cada uma a tarefa *inspeção* é executada antes da finalização do processo.

Diagrama de Classes

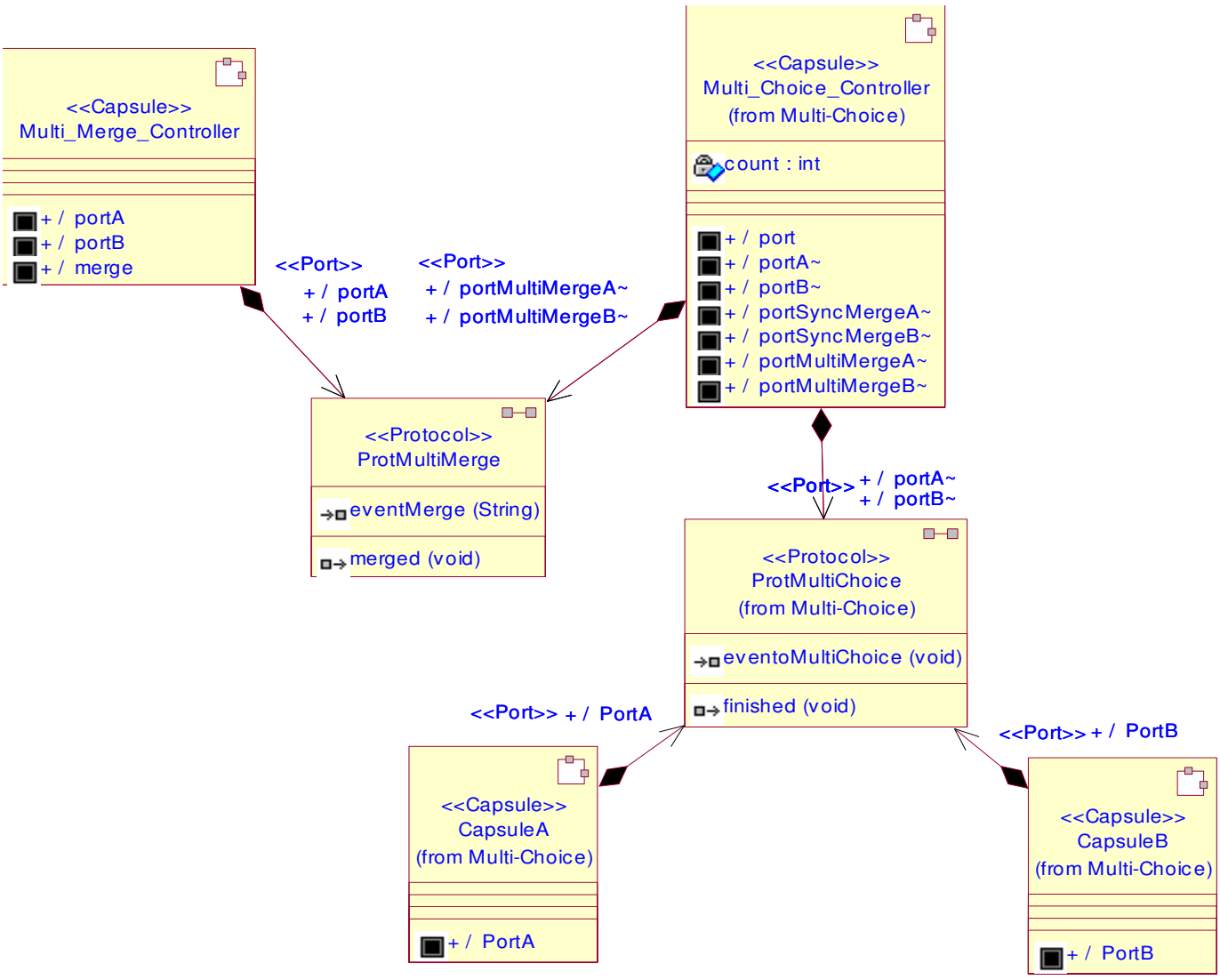


Figura 22. Diagrama de Classes do padrão *Multi Merge*

- *Multi_Merge_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. À medida que um *branch* de entrada é habilitado, a *thread* de controle é passada ao *branch* subseqüente, que representa o mesmo fluxo de atividades para qualquer *branch* de entrada.
- *ProtMultiMerge:* protocolo usado para comunicação entre as cápsulas *Multi_Merge_Controller* e *Multi_Choice_Controller* através do sinal de entrada *eventoMerge(String)* que indica quando um *branch* de entrada que flui do *Multi_Choice_Controller* está pronto para seguir o próximo fluxo do processo.
- *Multi_Choice_Controller*: módulo que antecede o fluxo do *Multi Merge* do qual os fluxos de entrada fluem para serem unidos no fluxo subseqüente.
- *CapsuleA*: cápsula que representa uma possível escolha de fluxo do processo que precede o *merge*.

- *CapsuleB*: cápsula que representa uma outra possível escolha de fluxo do processo que precede o *merge*.
- *ProtMultiChoice:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventoMultiChoice(void)* que indica qual ou quais das cápsulas foi(foram) escolhida(s) para continuar(em) o fluxo. O sinal de saída *finished*(*void*) é utilizado para indicar a finalização da execução de cada *branch* habilitado para o controlador.

Diagrama de Estrutura

O diagrama a seguir representa a estrutura da cápsula *Multi_Merge.* Esta cápsula contém as cápsulas *Multi_Choice_Controller*, *CapsuleA*, *CapsuleB* e *Multi_Merge_Controller.* Ela se comunica com o meio externo através da porta *port* que obedece ao protocolo *ProtSequence* e da porta *merge* com o protocolo *ProtMultiMerge*.

A cápsula *Multi_Choice_Controller* se comunica com as cápsulas *CapsuleA* e *CapsuleB* através das portas *portA* e *portB* que obedecem ao protocolo *ProtMultiChoice* e com a cápsula *Multi_Merge_Controller* através das portas *portMultiMergeA* e *portMultiMergeB* que se ligam às portas *portA* e *portB* e obedecem ao protocolo *ProtMultiMerge*.

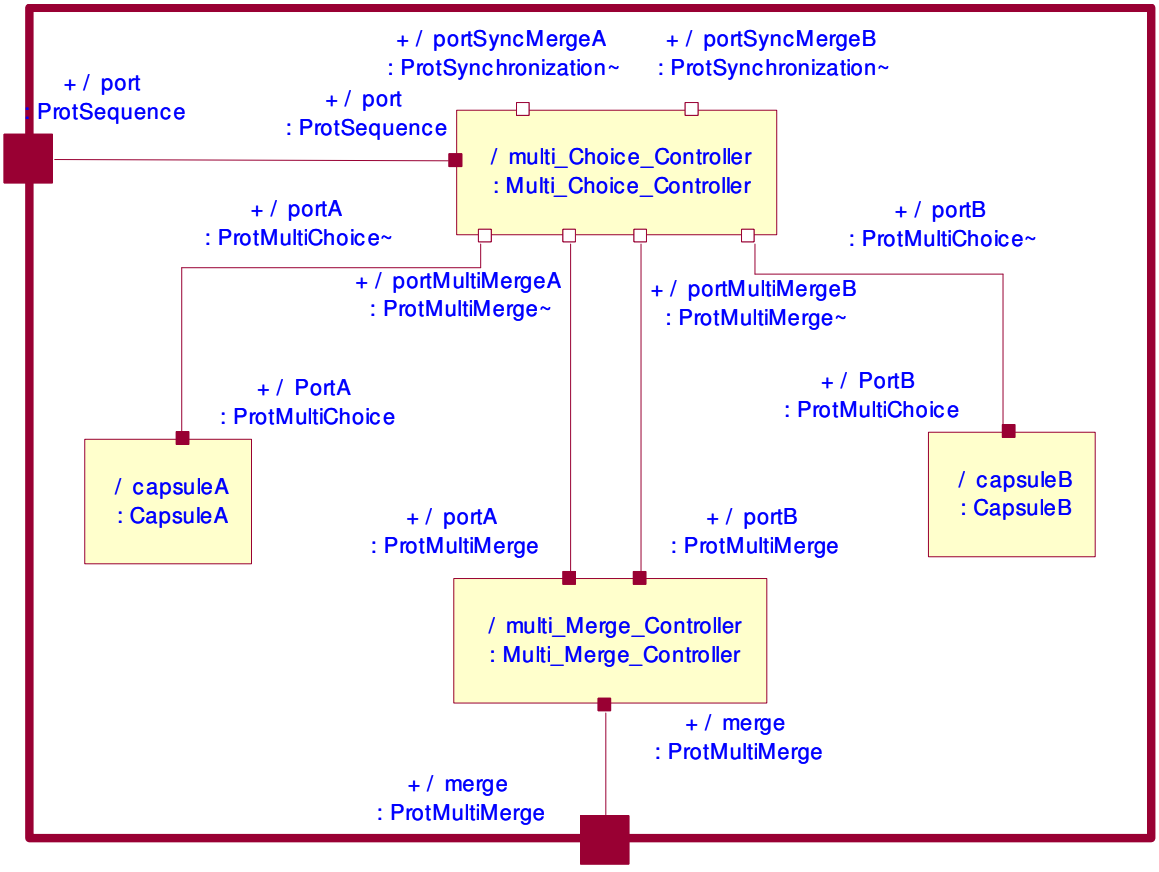


Figura 23. Diagrama de Estrutura da cápsula *Multi_Merge*

Diagrama de Estados

Quaisquer *threads* de controle chegando à estrutura *Multi Merge* devem ser passadas adiante no *branch* de saída. A distinção entre este padrão e o *Simple Merge* é que é possível ter mais de um *branch* ativo simultaneamente. Há uma condição de contexto associada a este padrão: o *Multi Merge* deve ser associado com uma construção *Multi Choice* precedente.

Este diagrama possui apenas um estado que fica esperando um *eventMerge(String)* através da *portA* ou da *portB*, uma vez que mais de um *branch* pode estar ativo simultaneamente. Se algum destes eventos disparar, as ações devidas são realizadas retornando ao estado inicial, sem que sinais vindos por outra porta sejam ignorados.

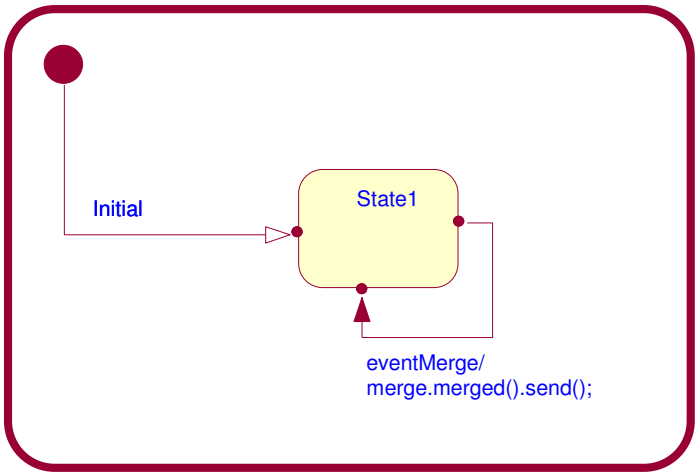
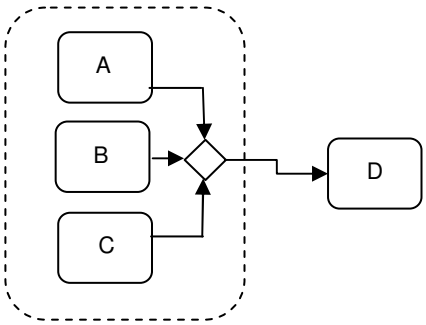


Figura 24. Diagrama de Estados da cápsula *Multi_Merge_Controller*

4.2.4 Structured Discriminator

A convergência de dois ou mais *branches* em um único *branch* subseqüente seguindo uma divergência anterior no modelo do processo tal que a thread de controle é passada ao *branch* subseqüente quando o primeiro *branch* de entrada tiver sido habilitado. Habilitações subseqüentes dos outros *branches* de entrada não resultam na passagem da *thread* de controle adiante. A construção *Structured Discriminator* reinicia quando todos os *branches* de entrada tiverem sido habilitados. Este padrão ocorre em um contexto estruturado, isto é, deve haver um módulo *Parallel Split* anterior no modelo do processo com o qual o *Structured Discriminator* é associado e deve fazer a união dos *branches* que fluem do *Parallel Split.* [4]



O padrão *Structured Discriminator* ou *1-out-of-M join* provê um meio de unir dois ou mais *branches* distintos de um processo em um único *branch* subseqüente tal que o primeiro deles a completar resulta no *branch* subseqüente sendo disparado, mas finalizações seguintes de outros *branches* de entrada não têm efeito sobre (e não disparam) o *branch* subseqüente. Assim sendo, o *Structured Discriminator* provê um mecanismo de progressão da execução de um processo uma vez que a primeira de uma série de tarefas concorrentes tenha completado.

Exemplo

Ao tratar uma parada cardíaca, as tarefas *checar respiração* e *checar pulso* executam em paralelo. Quando a primeira destas tarefas terminar, a tarefa *triagem* é iniciada. O término da outra tarefa é ignorado e não resulta em uma segunda instância da tarefa *triagem*.

Diagrama de Classes

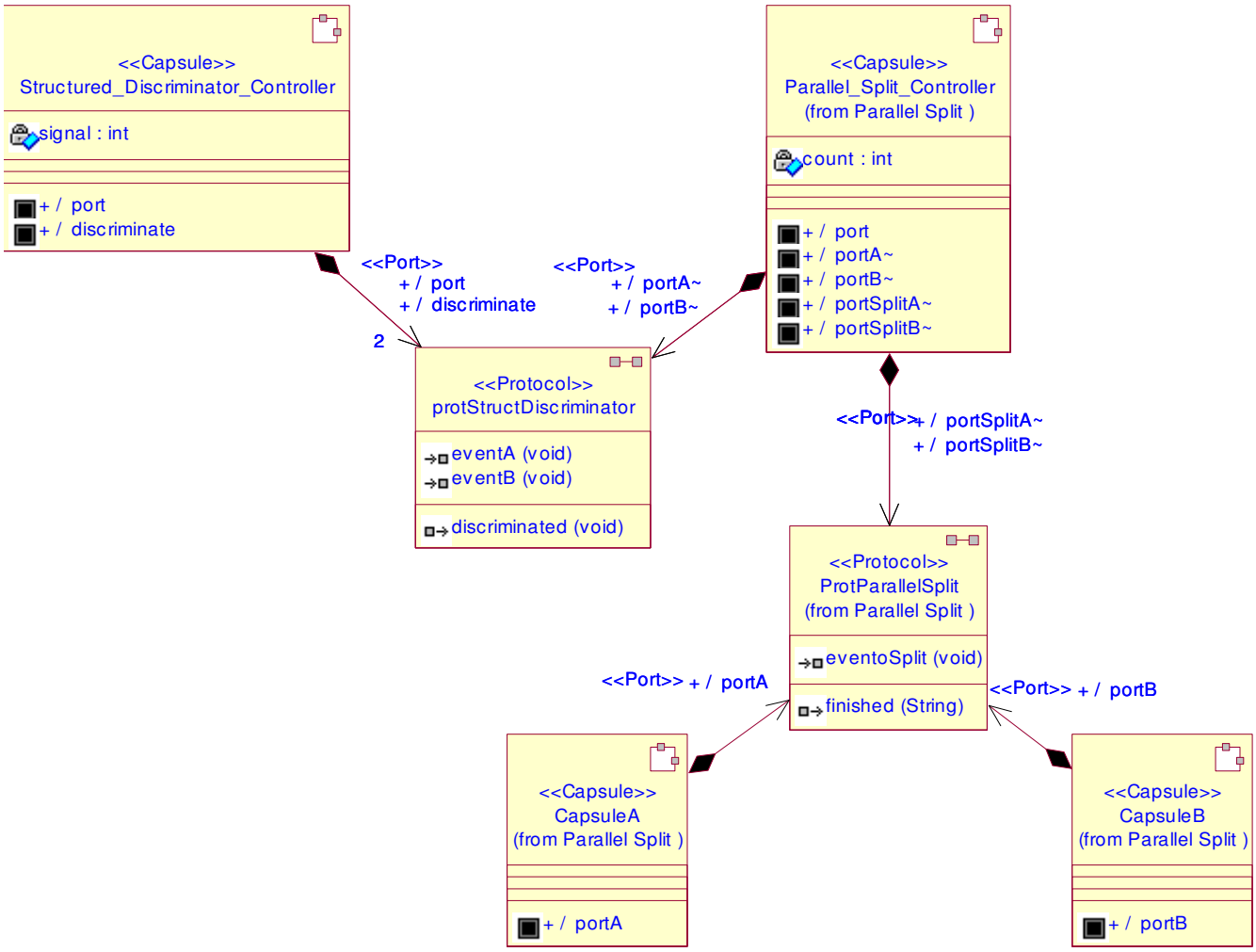


Figura 25. Diagrama de Classes do padrão *Structured Discriminator*

- *Structured_Discriminator_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. Esta cápsula recebe fluxos concorrentes que vêm do *Parallel Split* e dispara o fluxo seguinte assim que a primeira das tarefas tiver terminado, sendo os demais *branches* ignorados. A variável inteira *signal* é usada para armazenar o primeiro dos *branches* que foi finalizado para que a notificação de *branches* subseqüentes seja ignorada.
- *ProtStructDiscriminator:* protocolo que comunica as cápsulas *Structured_Discriminator_Controller* e *Parallel_Split_Controller* através dos sinais *eventA(void)* e *eventB(void)* os quais informam a finalização de cada *branch* habilitado ao *Structured_Discriminator_Controller*. O sinal de saída *discriminated(void)* informa ao ambiente que o componente *Structured_Discriminator_Controller* concluiu sua execução.
- *Parallel_Split_Controller*: cápsula que antecede o *Structured_Discriminator_Controller* provendo os fluxos concorrentes que irão entrar neste componente.

- *CapsuleA*: cápsula que representa a *thread* que executa em paralelo com a *CapsuleB*.
- *CapsuleB*: cápsula que representa a *thread* que executa em paralelo com a *CapsuleA*.
- *ProtParallelSplit:* protocolo usado para comunicar as cápsulas que através do sinal de entrada *eventoSplit(void)* o qual representa o evento a partir do qual as duas cápsulas *CapsuleA* e *CapsuleB* podem proceder concorrentemente; e do sinal de saída *finished(String)* usado para informar sobre a finalização de cada *branch* ativado.

Diagrama de Estrutura

O diagrama a seguir representa a estrutura da cápsula *Structured_Discriminator.* Esta cápsula contém as cápsulas *Parallel_Split_Controller, CapsuleA, CapsuleB* e *Structured_Discriminator_Controller.* Ela se comunica com o meio externo através da porta *port* que obedece ao protocolo *ProtSequence* e da porta *discriminate* com o protocolo *ProtStructDiscriminator.*

A cápsula *Parallel_Split_Controller* está ligada às cápsulas *CapsuleA* e *CapsuleB* por meio das portas *portSplitA* e *portSplitB* que obedecem ao protocolo *ProtParallelSplit.* Para comunicar à cápsula *Structured_Discriminator_Controller* sobre a finalização dos *branches*, o *Parallel_Split_Controller* possui as portas *portA* e *portB* ligadas à porta *port* as quais obedecem ao protocolo *ProtStructDiscriminator*. A porta *port* possui cardinalidade 2, isto é, ela pode ser conectada a duas portas distintas para receber o sinal vindo de cada um dos *branches* concorrentes ligados a ela.

A combinação da cápsula *Structured_Discriminator_Controller* e a *Parallel_Split_Controller* pode ser considerada um componente estruturado que pode ser incorporado em outros processos estruturados mantendo a estrutura geral.

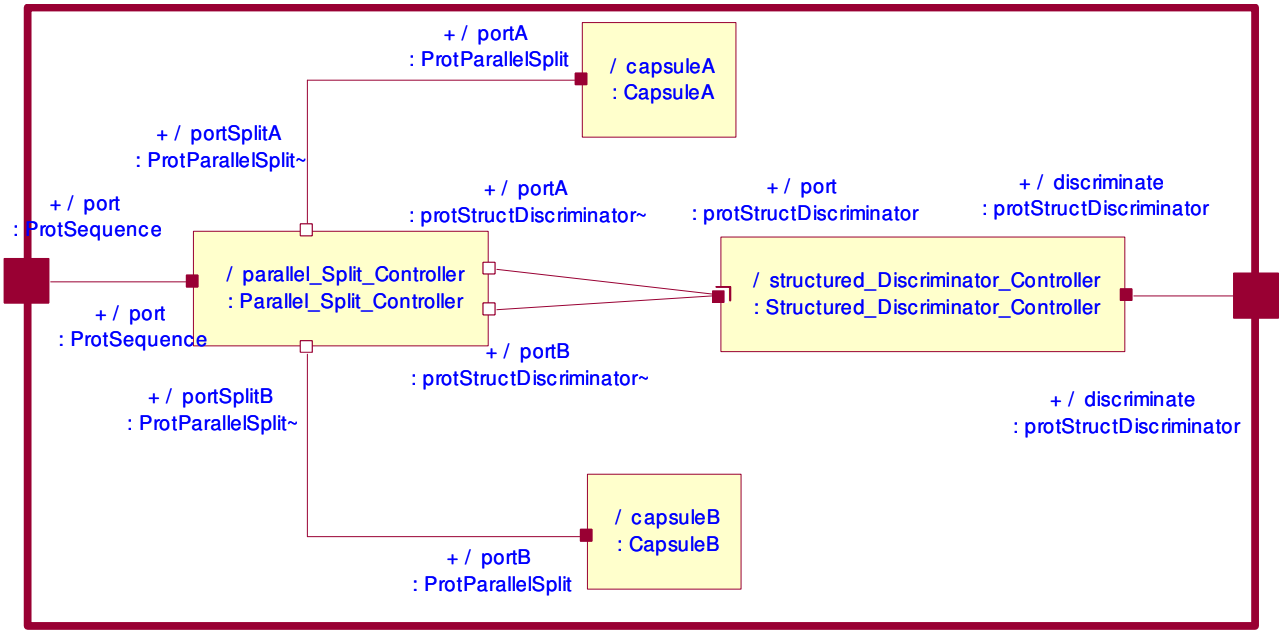


Figura 26. Diagrama de Estrutura da cápsula *Structured_Discriminator*

Diagrama de Estados

O primeiro *branch* de entrada habilitado resulta no *Discriminator* sendo notificado e o *branch* de saída é habilitado. Entradas subseqüentes habilitadas não têm efeito sobre o *Structured Discriminator* (e não resulta na ativação do *branch* de saída). Uma vez que cada *branch* de entrada tiver sido habilitado o *Structured Discriminator* reseta e pode ser reabilitado outra vez.

Há duas condições de contexto associadas ao uso deste padrão:

1. Uma vez que o *Structured Discriminator* foi ativado e ainda não foi resetado, não é possível que outro sinal seja recebido no *branch* ativado ou que múltiplos sinais sejam recebidos em qualquer *branch* de entrada.
2. Há uma estrutura *Parallel Split* correspondente e uma vez que ela foi habilitada nenhuma das tarefas nos *branches* ligando ao *Structured Discriminator* pode ser cancelada antes de ele ser disparado. A única exceção é que é possível cancelar **todas** as tarefas que estão ligadas ao *Structured Discriminator.* Podemos concluir que um comportamento correto do padrão é assegurado e se baseia apenas em informação local disponível ao *Structured Discriminator* em tempo de execução.

O diagrama de estados inicia com o estado *State1* que inicializa o inteiro *signal* com o valor '0' e espera por uma das entradas *eventA(void)* ou *eventB(void)* através da porta *port*. Com a chegada de um destes eventos é disparada a transição ao *State2*. Durante a transição o sinal que ocasionou o disparo é armazenado na variável *signal* através do método *rtGetMsgSignal()* provido pela API de UML-RT que retorna o valor inteiro associado a um determinado sinal. O evento *discriminated(void)* é enviado para a porta *discriminate* e o controlador vai para o *State2*. Neste estado, o *Structured_Discriminator_Controller* espera pela entrada que ainda não foi habilitada: *eventA(void)* ou *eventB(void)* com a guarda [*signal* != *rtGetMsgData()*] para garantir que o sinal recebido veio do *branch* que faltava para o controlador ser resetado e voltar ao estado inicial, sem realizar qualquer ação.

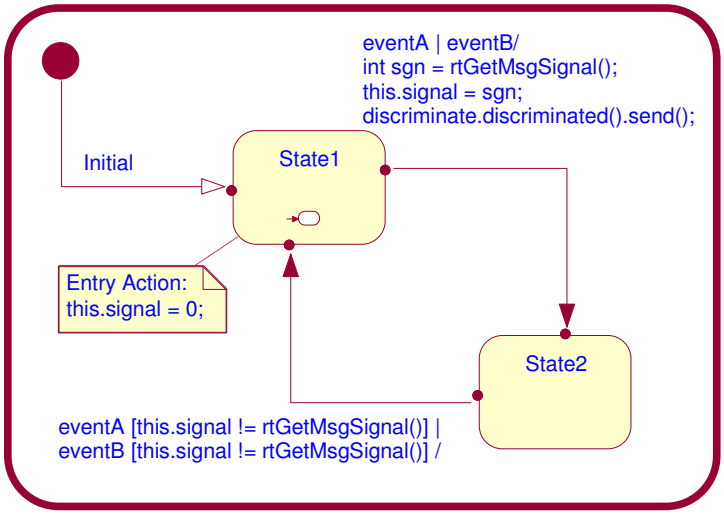


Figura 27. Diagrama de Estados da cápsula *Structured_Discriminator_Controller*

4.3 State-based Patterns

Os padrões *State-based* refletem situações cujas soluções são mais facilmente realizadas com linguagens que suportam a noção de estado. Nesse contexto, o estado do processo inclui todos os dados relevantes associados à execução atual das atividades.

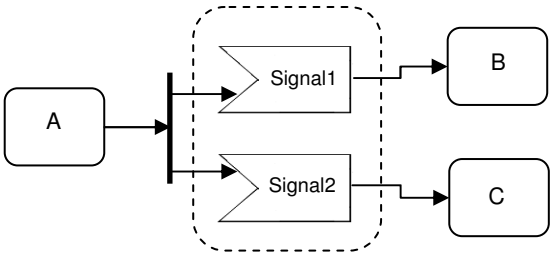
O conjunto original inclui três padrões nos quais o estado atual é o principal determinante no curso da ação que será tomada da perspectiva do controle do fluxo. Eles são: *Deferred Choice*, onde a decisão sobre qual *branch* deve ser escolhido é baseada em uma interação com o ambiente em operação; *Interleaved Parallel Routing*, no qual duas ou mais seqüências de atividades são executadas em forma de *interleaving* tal que apenas uma atividade é executada em um tempo determinado e *Milestone*, no qual a habilitação de uma dada atividade ocorre apenas quando o processo está em um estado específico.

Outros quatro padrões *State-based* foram posteriormente identificados: *Critical Section*, o qual provê a habilidade de evitar a execução concorrente de partes específicas de um processo; *Interleaved Routing*, que denota situações nas quais um grupo de atividades pode ser executado seqüencialmente em qualquer ordem; *Thread Merge* e *Thread Split* que provêm a junção e divergência de *threads* distintas de controle ao longo de um único *branch*.

Como definido para este trabalho, apresentaremos dois dos três padrões originalmente identificados, a saber: *Deferred Choice* e *Milestone,* com seus respectivos diagramas de classe, estrutura e estados.

4.3.1 Deferred Choice

Um ponto no processo onde um de vários *branches* é escolhido baseado em uma interação com o ambiente em operação. Precedendo a decisão, todos os *branches* representam possíveis cursos de execução futura. A decisão é feita pela inicialização da primeira tarefa em um dos *branches*. Após a decisão ser tomada, os demais *branches* alternativos são descartados. [4]



O padrão *Deferred Choice* provê a possibilidade de adiar o momento de escolha em um processo, isto é, o momento no qual um de vários possíveis cursos de ação deve ser escolhido é adiado até o último momento e é baseado em fatores externos à instância do processo (por exemplo, mensagens de chegada, dados do ambiente, disponibilidade de recursos, timeouts, etc.). Até o ponto no qual a decisão é feita, quaisquer das alternativas apresentadas representam cursos viáveis da ação futura.

Exemplo

Uma vez que um cliente requisitou a *entrega de um airbag*, ela é feita pelo *carteiro* ou por uma *tranportadora* dependendo de quem poderá entregar ao cliente primeiro.

Diagrama de Classes

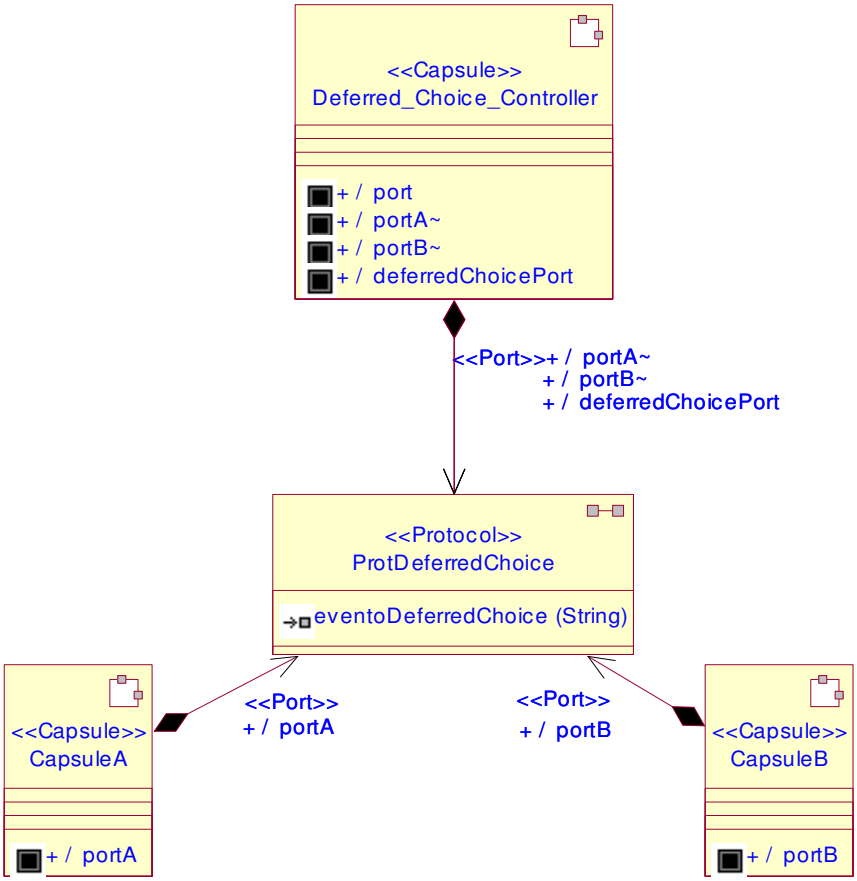


Figura 28. Diagrama de Classes do padrão *Deferred Choice*

- *Deferred_Choice_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. Ao ser iniciada ela espera que um evento externo aconteça para decidir qual dos fluxos possíveis ela irá executar, ignorando os demais.
- *ProtDeferredChoice:* protocolo usado para comunicar as três cápsulas que colaboram no padrão através do sinal de entrada *eventoDeferredChoice(String)* que indica qual dos *branches* deverá ser escolhido para continuar o fluxo do processo.
- *CapsuleA*: cápsula que representa uma possível escolha de fluxo do processo.
- *CapsuleB*: cápsula que representa uma outra possível escolha de fluxo do processo.

Diagrama de Estrutura

O diagrama a seguir representa a estrutura da cápsula *Deferred_Choice.* Esta cápsula contém as cápsulas *Deferred_Choice_Controller*, *CapsuleA* e *CapsuleB.* Ela se comunica com o meio externo através das portas: *port,* que obedece ao protocolo *ProtSequence* e *deferredChoicePort,* cujo protocolo é o *ProtDeferredChoice.* Esta última porta é o canal por onde o sinal que representa a escolha será transmitido à cápsula controladora.

A cápsula *Deferred_Choice_Controller* se comunica com as cápsulas *CapsuleA* e *CapsuleB* através das portas *portA* e *portB* que obedecem ao protocolo *ProtDeferredChoice*.

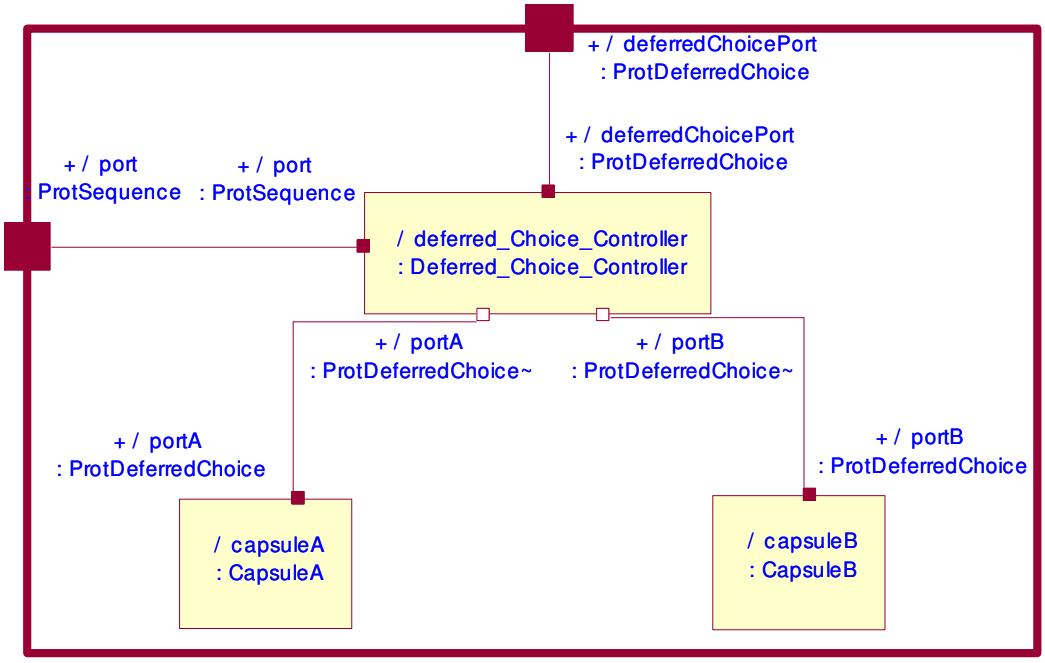


Figura 29. Diagrama de Estrutura do padrão *Deferred Choice*

Diagrama de Estados

A *thread* de controle é repassada à *CapsuleA* ou à *CapsuleB* dependendo da condição externa que decide qual dos branches será escolhido. Há uma condição de contexto associada a este padrão: apenas uma instância do *Deferred Choice* pode executar por vez.

No estado inicial *State1* o componente está pronto para executar e recebe um *eventoSeq(void)* indicando que um processo anterior finalizou e este pode iniciar. O componente fica então no *State2*, à espera do evento externo que indique o *branch* selecionado: o *eventoDeferredChoice(String)*. Uma vez ocorrido o evento, a máquina de estados irá avaliar o valor da *String* transmitida pelo sinal e transitará para o *State1*, enviando um sinal de ativação para a *CapsuleA* ou para a *CapsuleB* conforme a escolha informada.

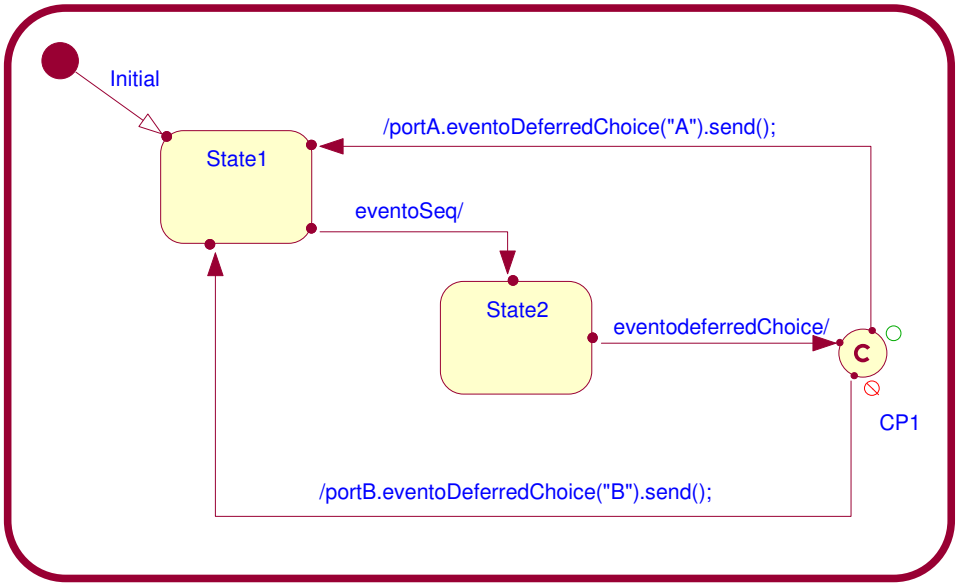
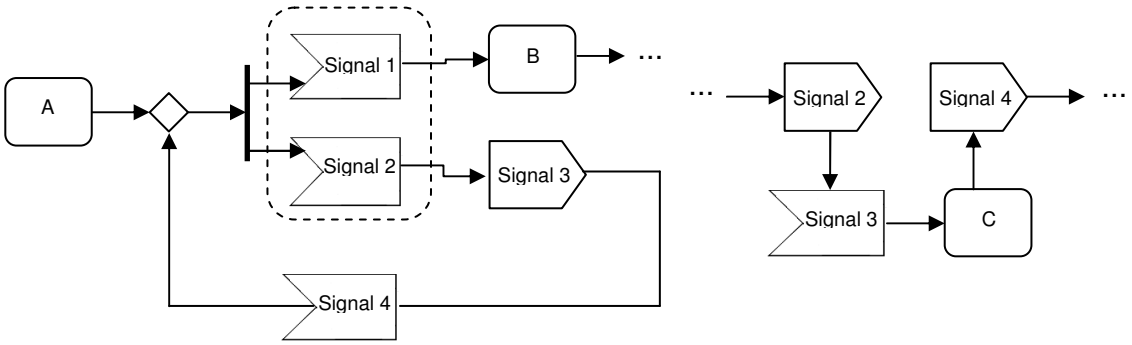


Figura 30. Diagrama de Estados da cápsula *Deferred_Choice_Controller*

4.3.2 Milestone

Uma tarefa apenas é habilitada quando a instância do processo - da qual ela faz parte – esteja em um estado específico (tipicamente um *branch* paralelo). O estado é admitido como um ponto de execução específico (conhecido como *milestone*) no modelo do processo. Quando este ponto de execução é alcançado a tarefa em questão pode ser habilitada.

Se a instância do processo progrediu além deste estado, então a tarefa não pode ser mais habilitada no momento ou em algum instante futuro (isto é, o *deadline* expirou). A execução não influencia o estado propriamente dito, ou seja, diferentemente de dependências de fluxo de controle normais, isto representa mais um teste do que um *trigger*. [4]



O padrão *Milestone* provê um mecanismo para suportar a execução condicional de uma tarefa ou subprocesso (possivelmente de forma repetida) na qual a instância do processo está em um determinado estado. A noção de estado é geralmente usada para indicar que o fluxo de controle alcançou um determinado ponto na execução da instância do processo (isto é, um *Milestone*). Assim, ele provê um meio de sincronizar dois *branches* distintos de um processo, tal que um *branch* não pode proceder até que o outro tenha alcançado um estado específico.

Exemplo

A tarefa *matricular aluno* pode executar apenas enquanto novas matrículas puderem ser aceitas. Isto acontece após a tarefa *abrir matrícula* ter completado e antes que a tarefa *encerrar matrícula* inicie.

Diagrama de Classes

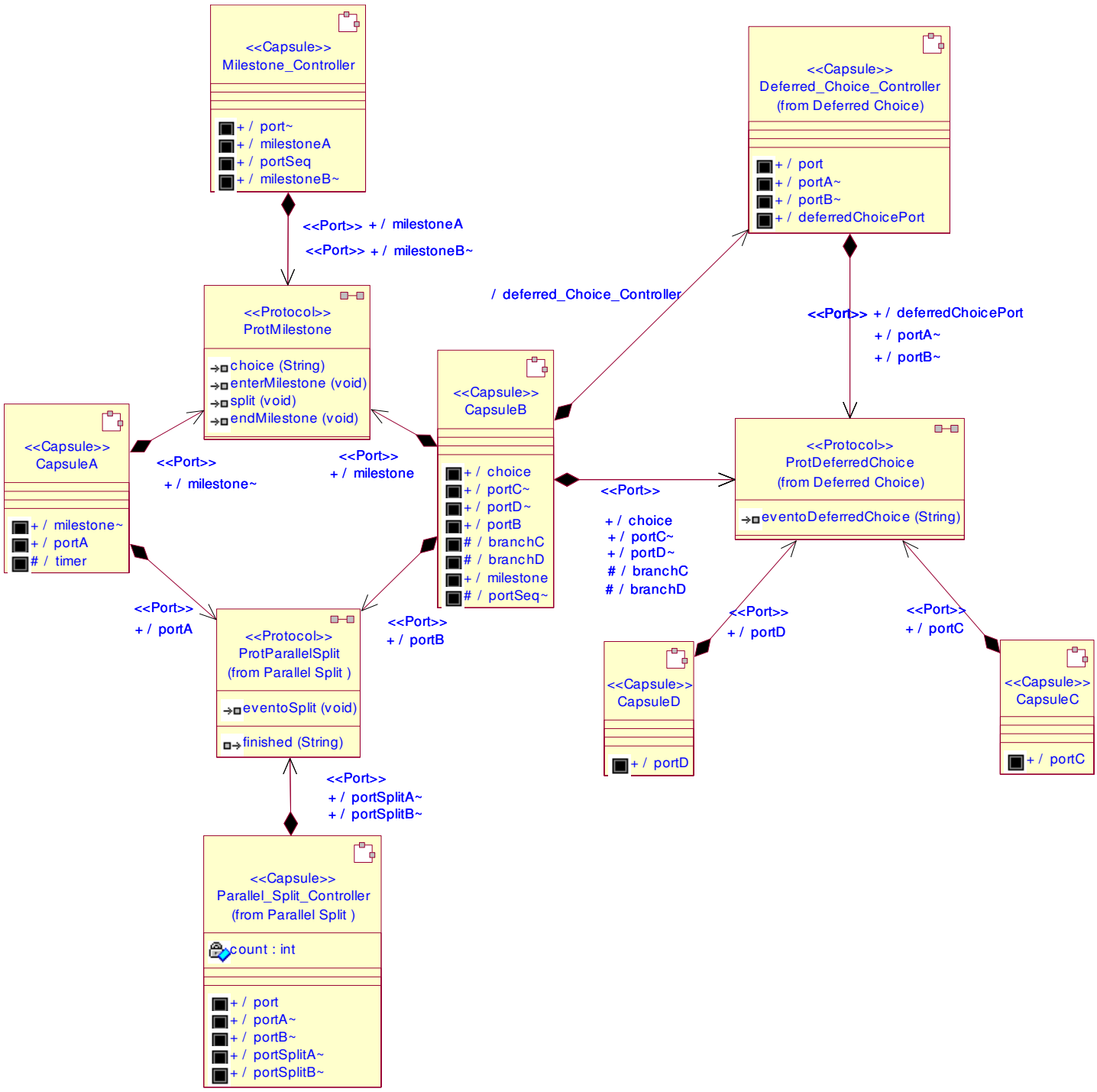


Figura 31. Diagrama de Classes do padrão *Milestone*

- *Milestone_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. Quando um dos branches atinge um determinado estado, ele permite a execução de uma tarefa específica no branch paralelo a ele. Ao sair deste estado esta tarefa não poderá mais ser executada.

- *ProtMilestone:* protocolo usado para comunicar o *Milestone_Controller* com os *branches* paralelos através dos sinais *split(void),* para disparar os *branches* paralelos, *enterMilestone(void),* para indicar que um dos branches está no estado *milestone*, *choice(String),* para indicar qual atividade será escolhida no branch paralelo e *endMilestone(void),* para indicar que o primeiro *branch* saiu do estado *milestone*.
- *Parallel_Split_Controller*: é a cápsula que dispara a execução dos *branches* paralelos, *CapsuleA* e *CapsuleB*, existentes no padrão.
- *ProtParallelSplit:* a partir do sinal de entrada *eventoSplit(void)* as duas cápsulas *CapsuleA* e *CapsuleB* procedem concorrentemente; o sinal de saída *finished(String)* informa o *Parallel_Split_Controller* sobre a finalização de cada *branch* ativado.
- *CapsuleA*: representa o fluxo de controle que pode alcançar um determinado ponto na execução da instância do processo - um *milestone* - que habilitará a execução de uma tarefa no *branch* paralelo. Ao sair deste estado, a tarefa que foi habilitada no *branch* paralelo deve ser desabilitada pelo controlador.
- *CapsuleB*: *branch* paralelo que pode ter uma tarefa habilitada pela chegada da *CapsuleA* ao milestone. Neste ponto, ela poderá prosseguir com a escolha externa de umas das tarefas que podem ser executadas neste momento. Se o primeiro *branch* não estiver no milestone, apenas uma atividade poderá ser selecionada pela escolha externa.
- *Deferred_Choice_Controller*: controlador reutilizado do padrão *Deferred Choice* para executar a escolha externa durante o milestone.
- *ProtDeferredChoice*: protocolo usado para comunicar a cápsula *Deferred_Choice_Controller* com as cápsulas *CapsuleC* e *CapsuleD* que representam as atividades a serem escolhidas durante o milestone, sendo que uma delas apenas está ativada quando a *CapsuleA* está no estado *milestone*. A comunicação acontece através do sinal de entrada *eventoDeferredChoice(String)* que indica qual dos *branches*, *CapsuleC* ou *CapsuleD*, deverá ser escolhido para continuar o fluxo do processo.
- *CapsuleC*: cápsula que representa fluxo ativado durante o estado *milestone*.
- *CapsuleD*: cápsula que representa um fluxo que está sempre ativo podendo ser executado a qualquer momento

Diagrama de Estrutura

Este diagrama representa a estrutura da cápsula *Milestone* que contém as cápsulas *Milestone_Controller*, *Parallel_Split_Controller*, *CapsuleA*, *CapsuleB*, *CapsuleC* e *CapsuleD*. Ela se comunica com o ambiente através das portas *portSeq*, que obedece ao protocolo *ProtSequence* e *deferredChoice* que obedece ao protocolo *ProtDeferredChoice*. Um sinal recebido pela porta *portSeq* indica que um componente anterior a este fluxo foi concluído este componente deverá iniciar sua execução. A porta *deferredChoice* é o canal que transmite o sinal de escolha para a *CapsuleB*.

A cápsula *Milestone_Controller* possui uma porta *port* que se liga à *Parallel_Split_Controller* para enviar um *eventoSeq* a ela que por as vez, através das portas *portSplitA* e *portSplitB* (*ProtParallelSplit*), inicia a execução em paralelo das cápsulas *CapsuleA* e *CapsuleB*, respectivamente.

A *Milestone_Controller* também possui a porta *milestoneA* que recebe o sinal da *CapsuleA* para indicar a entrada ou saída do estado *milestone* ao controlador e a porta conjugada *milestoneB* que envia o sinal de entrada ou saída do estado milestone à *CapsuleB* – ambas obedecem ao protocolo *ProtMilestone*.

A *CapsuleB* implementa o comportamento do padrão *Deferred Choice* (seu diagrama de estrutura será mostrado a seguir) e por isso possui uma porta *choice* que se liga à porta *deferredChoice* e as portas *portC* e *portD* (*ProtDeferredChoice*) que se ligam às cápsulas *CapsuleC* e *CapsuleD* para ativar um dos fluxos de execução.

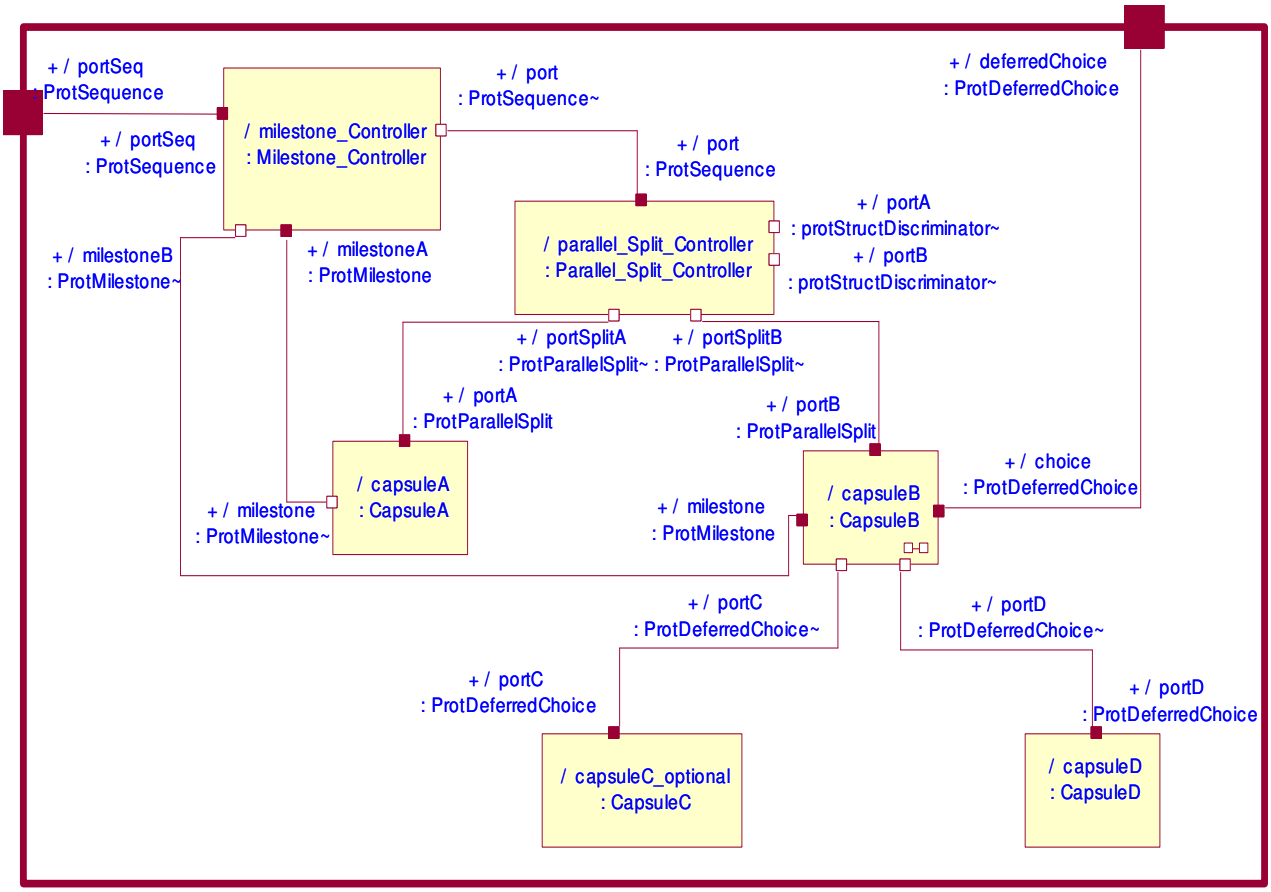


Figura 32. Diagrama de Estrutura da cápsula *Milestone*

A *CapsuleB* é composta por uma cápsula *Deferred_Choice_Controller* que implementa o comportamento do padrão *Deferred Choice* para a escolha de um *branch* a ser executado. Ela recebe a escolha diretamente do ambiente, por isso possui uma porta *relay choice* ligada diretamente à *Deferred_Choice_Controller* que irá executar e indicar em uma das portas *end branchC* ou *branchD* qual *branch* foi selecionado.

A porta *end portSeq* inicializa o *Deferred_Choice_Controller* para que ele fique à espera do evento de escolha a ser recebido do ambiente.

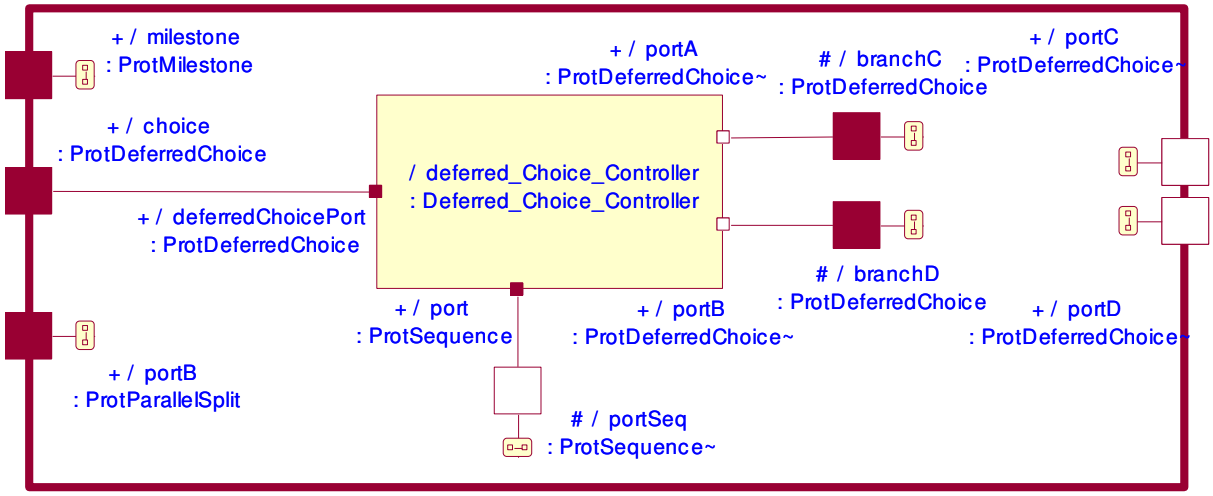


Figura 33. Diagrama de Estrutura da *CapsuleB*

Diagrama de Estados

A *CapsuleC* não pode ser habilitada, mesmo com o *branch* da *CapsuleB* ativo, a menos que o outro *branch* esteja no estado *milestone*. Esta situação presume que a instância do processo está no estado *milestone* ou estará em algum tempo futuro. É importante notar que a execução repetida da *CapsuleC* não influencia o outro *branch* paralelo.

Assim a máquina de estados do *Milestone_Controller* é inicializada no *State1* e após ser ativada, envia um *eventoSeq(void)* ao *Parallel_Split_Controller* para iniciar os branches A e B. Seu diagrama possui um estado *Milestone* o qual será alcançado quando receber um sinal *enterMilestone(void)* da *CapsuleA* e será abandonado quando receber um sinal *endMilestone(void)* da mesma. Ao receber estes sinais, o controlador avisa à *CapsuleB* quando entrou e saiu do estado *Milestone*, através dos sinais *enterMilestone(void)* e *endMilestone(void)*, respectivamente, para que a *CapsuleC* seja habilitada ou desabilitada.

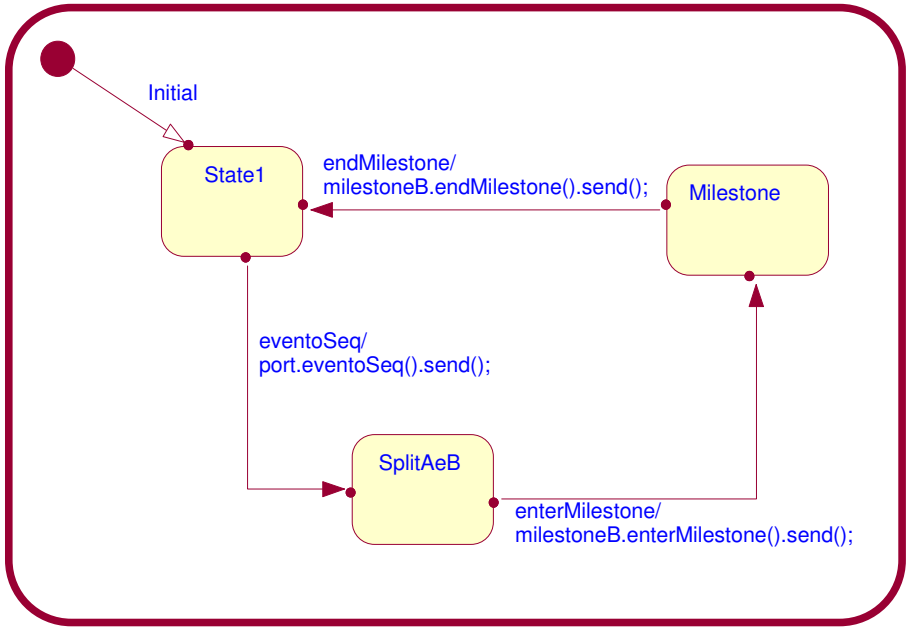


Figura 34. Diagrama de estados da cápsula *Milestone_Controller*

Como a atividade C no *branch* paralelo só pode ocorrer se a *CapsuleA* estiver no estado *milestone*, isso pode causar um possível *deadlock*, pois o *branch* paralelo poderá não prosseguir. Para evitar esta situação, definimos uma tarefa alternativa D para que o processo paralelo não fique impedido se o *branch* A sair do estado *milestone*. Nesse caso, a atividade C pode ser opcional, ou seja, não executar de jeito nenhum, ou pode ocorrer múltiplas vezes até que o controlador saia do estado *Milestone*.

Este comportamento foi implementado na máquina de estados da *CapsuleB* como mostrado abaixo.

A *CapsuleB* representa o *branch* paralelo ao que contém o estado *milestone* (neste caso a *CapsuleA*). Ela é ativada simultaneamente à *CapsuleA* através do *Parallel_Split_Controller* que envia um sinal *eventoSplit(void)* e ela inicia sua execução a partir do *State2*.

Quando a *CapsuleA* atinge o estado *milestone*, ela avisa ao *Milestone_Controller* e este repassa o sinal *enterMilestone(void)* para a *CapsuleB*. Esta cápsula contém um *Deferred_Choice_Controller* que é ativado neste momento a partir de um *eventoSeq(void)* e fica à espera de uma escolha externa para executar as atividades da *CapsuleC* ou da *CapsuleD*.

Note que o *eventoDeferredChoice("CapsuleC")* recebido através da porta *portC* só pode ser disparado quando esta cápsula estiver no estado *State3* que corresponde ao estado *milestone* da *CapsuleA*. Nesse estado, ele poderá seguir o fluxo da *CapsuleC*, a partir do *eventoDeferredChoice("CapsuleC")* ou da *CapsuleD*, a partir do *eventoDeferredChoice("CapsuleD")*.

As atividades da *CapsuleC* poderão ser executadas repetidas vezes. A *CapsuleB* só termina de executar quando a *CapsuleD* for ativada.

Se esta cápsula receber do controlador um sinal *endMilestone(void)* indicando que a *CapsuleA* saiu do estado *milestone*, a *CapsuleB* irá para o estado *State4* onde ficará esperando a ocorrência do *eventoDeferredChoice("CapsuleD")* para terminar sua execução.

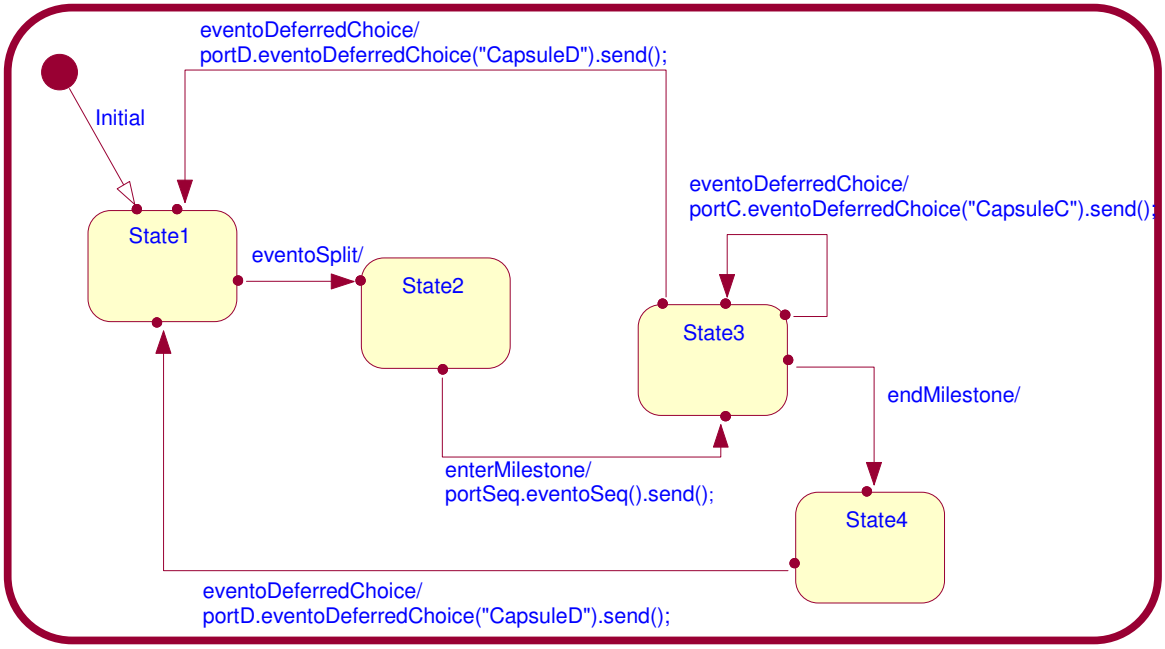


Figura 35. Diagrama de estados da cápsula *CapsuleB*

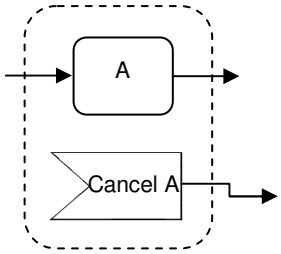
4.4 Cancellation and Force Completion Patterns

Muitos dos padrões descritos acima, como o *Structured Synchronizing Merge* e o *Structured Discriminator,* possuem variantes que utilizam o conceito de cancelamento de atividades que tenham sido habilitadas ou estejam ativas. Várias formas de tratamento de exceções em processos também são baseadas em conceitos de cancelamento.

Esta seção apresenta dois padrões de cancelamento: *Cancel Task* e *Cancel Case*. Três novos padrões de cancelamento também foram identificados: *Cancel Region*, *Cancel Multiple Instance Activity* e *Complete Multiple Instance Activity*.

4.4.1 Cancel Task

Uma tarefa habilitada é descartada antes de iniciar sua execução. Se a tarefa foi iniciada, ela é desabilitada e, quando possível, a instância atual em execução é parada e removida. [4]



O padrão *Cancel Task* provê a habilidade de remover uma tarefa que já tenha sido habilitada ou que já esteja executando. Isso garante que ela não irá começar ou completar sua execução.

Exemplo

A tarefa *acessar danos* é realizada por dois assessores da seguradora. Uma vez que o primeiro assessor tenha completado a tarefa, a segunda é cancelada.

Diagrama de Classes

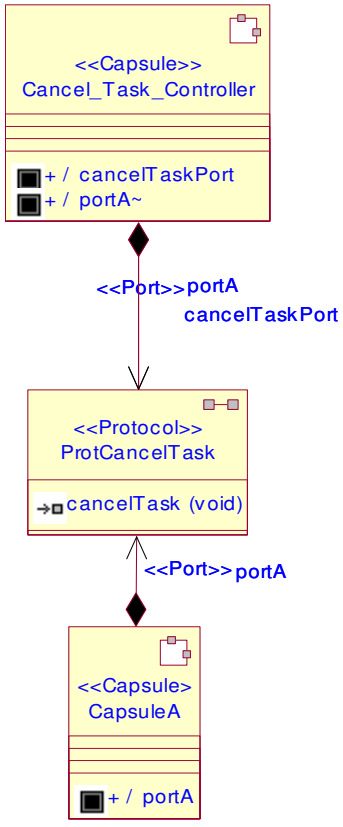


Figura 36. Diagrama de Classes do padrão *Cancel Task*

- *Cancel_Task_Controller*: é a cápsula que implementa o comportamento do padrão através de sua máquina de estados. Ao receber um sinal *cancelTask(void)* do ambiente, este controlador envia o mesmo sinal a uma cápsula a qual está ligada para cancelar suas atividades.
- *ProtCancelTask:* protocolo usado para comunicar as duas cápsulas que colaboram no padrão através do sinal de entrada *cancelTask(void)* que é recebido pelo *Cancel_Task_Controller* e repassado à *CapsuleA*.
- *CapsuleA*: cápsula que representa o fluxo que possui a atividade a ser cancelada do processo. Esta cápsula deverá possuir um estado no qual ela permaneça inativa, caso receba um sinal de cancelamento, ou seja, nenhum evento recebido é percebido pela cápsula quando ela está neste estado.

Diagrama de Estrutura

O diagrama a seguir representa a estrutura da cápsula *Cancel_Task.* Esta cápsula contém as cápsulas *Cancel_Task_Controller* e *CapsuleA*. Ela se comunica com o meio externo através da porta *cancelTaskPort,* cujo protocolo é o *ProtCancelTask.* Esta porta é o canal por onde o sinal que representa o cancelamento da tarefa é transmitido à cápsula controladora.

A cápsula *Cancel_Task_Controller* se comunica com a cápsula *CapsuleA* através da porta *portA* que obedece ao protocolo *ProtCancelTask*.

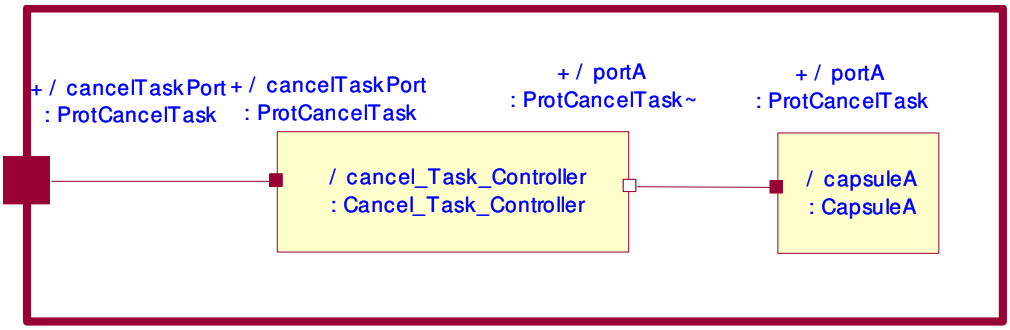


Figura 37. Diagrama de Estrutura do padrão *Cancel Task*

Diagrama de Estados

A cápsula *Cancel_Task_Controller* fica à espera de um sinal do ambiente que indique o cancelamento de uma atividade a ser feito por esta cápsula. A decisão de cancelar uma atividade pode apenas ser feita depois que ela tenha sido habilitada e antes de ter completado.

Se esta decisão for tomada, não é possível que a tarefa prossiga. Não é possível cancelar uma tarefa que não tenha sido habilitada nem que já tenha completado sua execução.

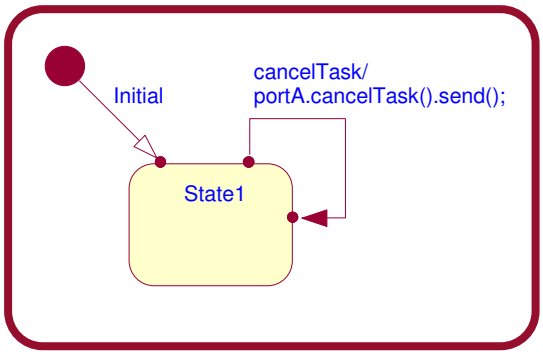


Figura 38. Diagrama de Estados do *Cancel_Task_Controller*

A cápsula a ser cancelada está habilitada no estado *Executing* no qual pode receber quaisquer eventos definidos pelas suas portas (o *evento/* mostrado na transição é apenas ilustrativo e representa os sinais que esta cápsula poderá receber através de suas portas enquanto estiver no estado *Executing*). Na ocorrência de um evento *cancelTask(void)* ela irá para o estado *Cancelled* onde não executará nenhuma ação nem receberá eventos vindos das suas portas.

Este padrão poderia ter sido implementado com o uso de cápsulas opcionais, um recurso de UML-RT que permite a inserção e remoção de cápsulas durante a sua execução. Neste caso, esta cápsula seria removida dinamicamente do processo. Por razões de simplificação, decidimos não usar este recurso e apenas isolar a cápsula de eventos externos.

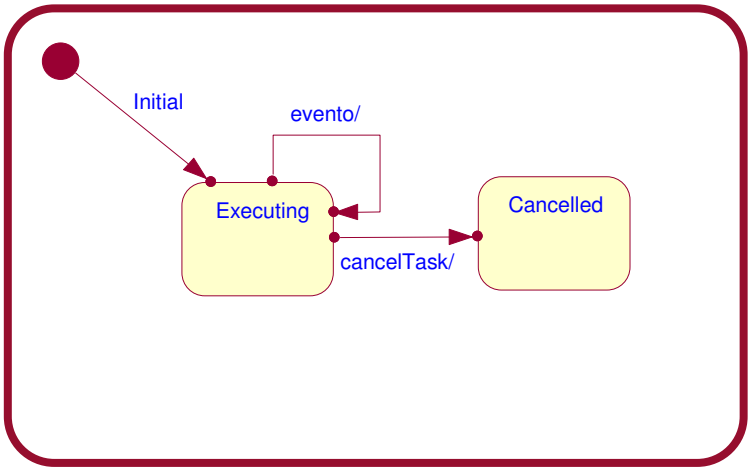
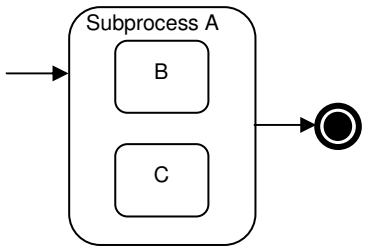


Figura 39. Diagrama de Estados da *CapsuleA*

4.4.2 Cancel Case

Uma instância completa do processo é removida. Isto inclui tarefas que estejam executando, aquelas que irão executar e todos os seus subprocessos. A instância do processo é registrada como tendo completado sua execução sem sucesso[4].



O padrão *Cancel Case* ou *Withdraw Case* provê uma maneira de parar uma instância de processo específica e remover quaisquer tarefas associadas a ela.

Exemplo

Durante um processo de *ressarcimento de seguro*, descobre-se que a apólice expirou e como conseqüência, todas as tarefas associadas à instância deste processo são canceladas.

Diagrama de Classes

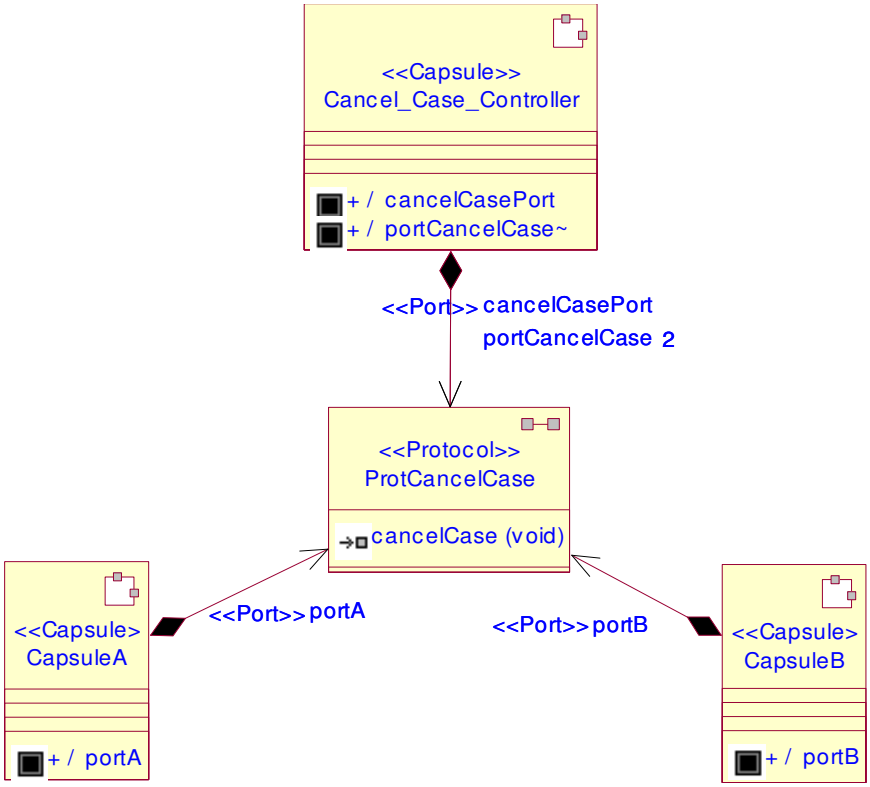


Figura 40. Diagrama de Classes do padrão *Cancel Case*

- *Cancel_Case_Controller*: implementa o comportamento do padrão com sua máquina de estados. Recebe um sinal *cancelCase(void)* e o envia para as cápsulas às quais está ligada para cancelar suas atividades.
- *Prot_Cancel_Case*: protocolo usado para comunicar as cápsulas que colaboram no padrão através do sinal de entrada *cancelCase(void)* que é recebido pelo *Cancel_Case_Controller* e repassado às cápsulas que estiverem ligadas ao controlador, neste caso, *CapsuleA* e *CapsuleB*.
- *CapsuleA*: cápsula que representa o fluxo a ser cancelado do processo. Esta cápsula deverá possuir um estado no qual ela permaneça inativa, caso receba um sinal de cancelamento, ou seja, nenhum evento recebido é percebido pela cápsula quando ela está neste estado.
- *CapsuleB*: outro fluxo dentro do mesmo processo que deve ser cancelado. Deve também possuir um estado no qual fique inativo, caso receba um sinal de cancelamento, ou seja, nenhum evento recebido é percebido pela cápsula quando ela está neste estado.

Diagrama de Estrutura

A cápsula *Cancel_Case_Controller* está definida de forma todas as cápsulas participantes do processo sejam conectadas a ela através de sua porta que pode ser de cardinalidade 'n', tal que 'n' é o número total de cápsulas do processo a ser cancelado. Desta forma, as cápsulas *CapsuleA* e *CapsuleB* representam subprocessos contidos em um processo maior que será completamente cancelado. Suas portas obedecem ao protocolo *ProtCancelCase*.

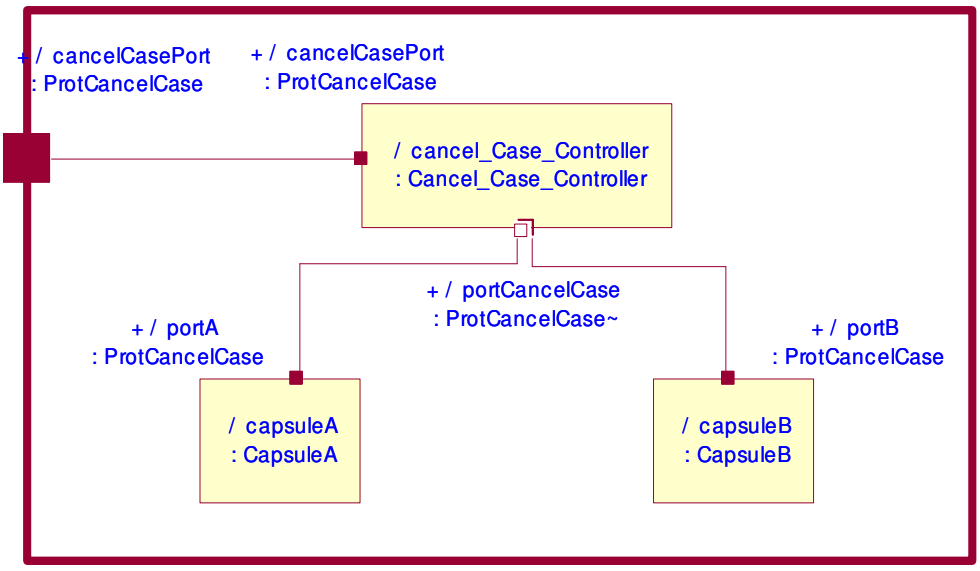


Figura 41. Diagrama de Estrutura da cápsula *Cancel Case*

Diagrama de Estados

O cancelamento de um caso inteiro envolve a desabilitação de todas as tarefas que estão habilitadas no momento. Este cancelamento do processo em execução deve ser considerado como uma finalização sem sucesso do processo. Por exemplo, se há um *log* de eventos ocorridos durante a execução do processo, o caso deve ser registrado como incompleto ou cancelado.

A cápsula *Cancel_Case_Controller* fica à espera de um sinal do ambiente que indique o cancelamento de um processo a ser feito por esta cápsula.

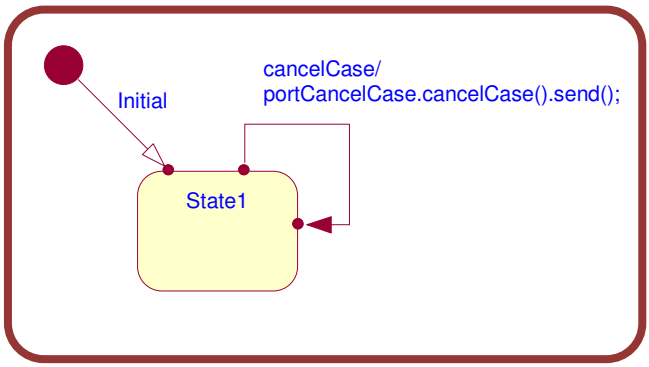


Figura 42. Diagrama de Estados da cápsula *Cancel_Case_Controller*

As cápsulas a serem canceladas estão habilitadas no estado *Executing* no qual podem receber quaisquer eventos definidos pelas suas portas (o *evento/* mostrado na transição é apenas ilustrativo e representa os sinais que estas cápsulas podem receber através de suas portas enquanto estiverem no estado *Executing*). Na ocorrência de um evento *cancelCase(void)* elas irão para o estado *Cancelled* onde não executarão nenhuma ação nem receberão eventos vindos das suas portas.

Assim como o *Cancel Task*, este padrão poderia ter sido implementado com o uso de cápsulas opcionais, um recurso de UML-RT que permite a inserção e remoção de cápsulas durante a sua execução. Neste caso, todas as cápsulas do processo seriam removidas dinamicamente. Por razões de simplificação, decidimos não usar este recurso e apenas isolar as cápsulas de eventos externos.

5. Exemplos de Aplicações dos Padrões

Estudaremos a aplicabilidade dos padrões propostos no contexto de dois exemplos específicos relacionados a uma agência de viagens, criados por Van der Aalst [27].

O primeiro exemplo representa o processo de Requisição de Viagem, no qual os clientes se comunicam com a agência para programar uma viagem com agendamento de passagens e diárias de hotel.

O segundo exemplo se refere ao processo feito pela agência de Submissão de Formulário para registro e processamento de reclamações de clientes.

Cada cápsula que representa o processo implementado é composta por elementos dos padrões sem a necessidade da especificação de suas máquinas de estados. Apenas as cápsulas que instanciam as tarefas precisam ser implementadas, seguindo a referência do modelo, na realização do processo.

No Apêndice A estão os *logs* das simulações de cada exemplo, com o registro dos passos ocorridos na execução das cápsulas envolvidas na realização do processo.

5.1 Travel Request

A agência de viagens recebe requisições de clientes pelo telefone e as armazenam em documentos de viagens. As requisições para reserva de passagens e hotel são gerenciadas por *Web Services* independentes que existem fora da agência e recebem as suas requisições.

Os serviços reportam seu sucesso ou falha na reserva das requisições para a agência. Se todas as requisições foram realizadas com sucesso, a agência compila todos os documentos da viagem para o usuário e confirma o itinerário da viagem. Se um dos serviços falharem na realização da reserva, a agência informa ao outro serviço que suspenda as reservas e entra em contato com o cliente para revisar o itinerário da viagem.

Para realizar este processo utilizamos o *Multi_Choice_Controller* para fazer a escolha entre agendar as passagens (*Book_Flight*) e reservar o hotel (*Book_Hotel*). Feita a escolha das atividades, utilizamos o *Synchronizing_Merge_Controller* para sincronizar as tarefas que estão sendo executadas e disparar um *trigger* para habilitar a realização da tarefa Pagamento (*Pay*) após a conclusão das tarefas selecionadas.

As duas cápsulas *Cancel_Task_Controller* ligadas às tarefas *Book_Flight* e *Book_Hotel* servem para suspender um serviço de reserva caso haja falha no outro serviço selecionado.

Na simulação deste exemplo consideramos dois cenários.

1. A atividade *Book Hotel* falha e há o cancelamento da tarefa *Book Flight*, não havendo, portanto, sincronização;
2. As duas tarefas *Book Hotel* e *Book Flight* são realizadas com sucesso, há a sincronização dos *branches* e a execução da tarefa *Pay*.

Os *logs* da simulação deste exemplo no *Rational Rose RealTime* se encontram no Apêndice A.

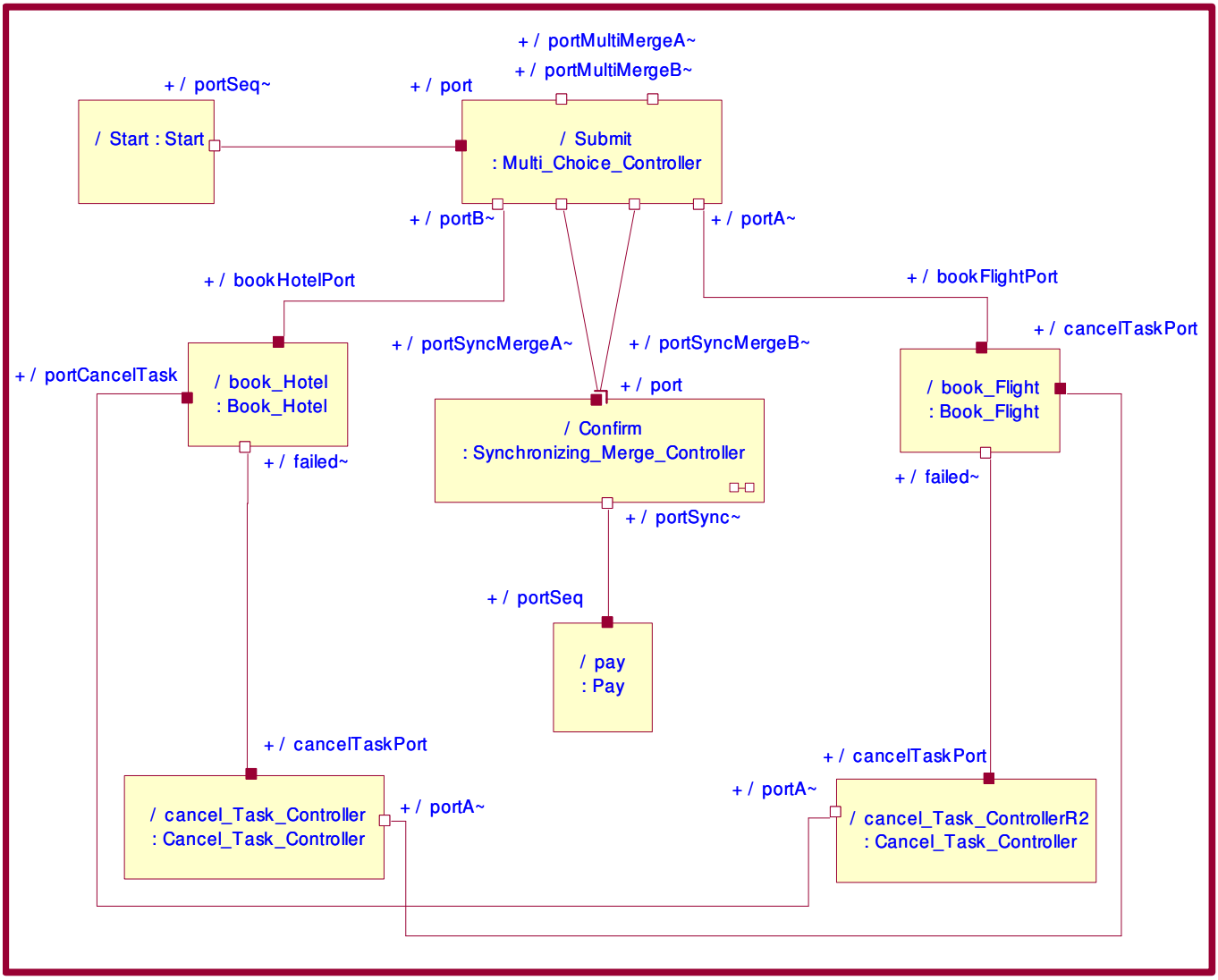


Figura 43. Diagrama de Estrutura do exemplo *Travel Request*

5.2 Submit Form

Mostraremos agora outro exemplo prático de negócio e como este caso é modelado com os padrões propostos.

A cada ano uma agência de viagens processa várias reclamações (em torno de 10000). Há um departamento especial para o processamento de reclamações (departamento R). Há também um departamento interno chamado de Logística (departamento L) que cuida do registro das reclamações que chegam e do arquivamento das reclamações processadas. O seguinte procedimento é usado para lidar com estas reclamações.

Um empregado do departamento L primeiramente registra cada reclamação que chega. Após o registro, um empregado do departamento R envia um formulário ao cliente com questões sobre a natureza da reclamação. Existem duas possibilidades: o cliente retorna o formulário dentro de duas semanas ou não. Se o formulário é retornado, ele é processado automaticamente resultando em um relatório que pode ser usado para o processamento da reclamação. Se o formulário não retorna no tempo definido, um *timeout* ocorre resultando em um relatório vazio. Isto não significa que a reclamação foi descartada.

Após o registro, isto é, em paralelo com o processamento do formulário, a preparação para o processamento da reclamação é iniciado. Primeiro, a reclamação é avaliada por um gerenciador de reclamações do departamento R. A avaliação diz se um novo processamento é ou não necessário. Esta decisão não depende do processamento do formulário.

Se não é necessário um novo processamento da reclamação e o formulário já foi processado, a reclamação é arquivada. Se um novo processamento da reclamação é necessário, um empregado do departamento R executa a tarefa 'processar reclamação'.

Para o real processamento da reclamação, o relatório resultante do processamento do formulário é utilizado (este relatório pode ser vazio). O gerenciador de reclamações checa o resultado da tarefa 'processar reclamação'. Se o resultado não estiver correto, a tarefa 'processar reclamação' é executada novamente. Isto é repetido até que o resultado seja aceitável. Se o resultado for aceito, um empregado do departamento C executa as ações propostas. Logo após, a reclamação processada é arquivada por um empregado do departamento L.

Este exemplo é modelado com o padrão *Milestone* utilizando o *Milestone_Controller* e as implementações de referência das cápsulas auxiliares para instanciação das atividades deste processo. Além do *Milestone_Controller* utilizamos o *Deferred_Choice_Controller* para decidir se a Avaliação (*Evaluate*) do formulário se dará após o Processamento do Formulário (*ProcessForm*) pelo usuário ou após o tempo de espera haver expirado (*TimeOut*).

Na execução da tarefa *Evaluate* o problema poderá ser processado na atividade *Process_Complaint* e logo após, o documento gerado pelo processamento deve ser revisado na atividade *Review* que decidirá se o problema será novamente processado (podendo esta atividade ocorrer repetidas vezes) ou será arquivado (*Archive*) e finalizado.

A seguir está o Diagrama de Estrutura que implementa este processo.

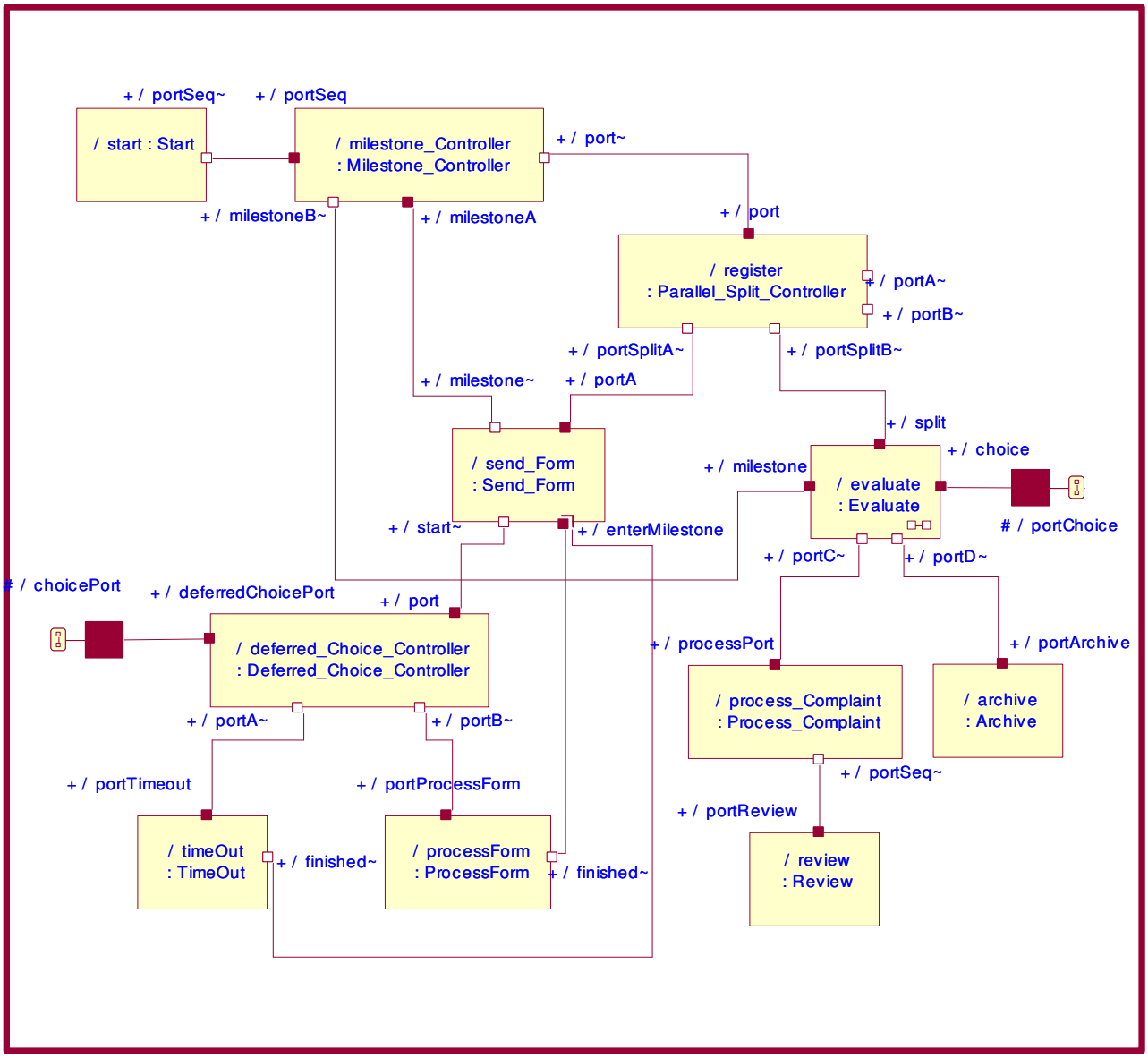


Figura 44. Diagrama de Estrutura do processo Submit Form

O log da simulação deste exemplo no *Rational Rose RealTime* se encontra no Apêndice A.

6. Considerações Finais: Trabalhos Relacionados, Conclusão e Trabalhos Futuros

6.1 Trabalhos Relacionados

Há muitas contribuições sobre padrões de projeto para componentes, como [22] que identifica alguns padrões fundamentais que se aplicam a processos comunicantes em geral (sistemas que utilizam processos, objetos, componentes, *device drivers* e outras entidades – interconectados de várias maneiras e com diferentes propósitos) e componentes de software em particular. Eles estão na base da hierarquia de padrões com interação dinâmica, pois são padrões básicos que não estão associados a um tipo específico de sistema de software. Ele apresenta diagramas simples de interação dos componentes e uma breve descrição textual sobre seu comportamento, além de mostrar exemplos práticos para cada padrão. No entanto, ele não apresenta um direcionamento específico de como eles devem ser implementados, ao contrário do nosso trabalho que o faz por meio dos diagramas de estados.

Em [23] são apresentados cinco padrões para tornar mais fácil a junção de componentes que se comunicam, colaboram e coordenam para realizar uma tarefa, uma vez que a construção de um sistema constituído de componentes customizados e pré-construídos pode ser difícil devido a dependências não visíveis, interações complexas e projetos incompreensíveis. Assim como no livro *Real Time Design Patterns* [30], vários destes padrões são inspirados em padrões clássicos encontrados em sistemas operacionais e embarcados, como barramento de dados e *pipelines*. A maioria deles foca em uma abordagem endógena que requer que os componentes tenham ciência de sua participação global no padrão. Nosso trabalho aqui foca em padrões de coordenação exógenos [25], onde os componentes possuem um papel no padrão, mas não precisam ser adaptados ao contexto do padrão.

Sobre a utilização de UML para apoiar regras de negócios ou empresariais, [24] mostra como as técnicas e recursos de UML tais como diagramas de caso de uso, diagramas de atividades, diagramas de classes, diagramas de estados, diagramas de seqüência, casos de teste e prototipagem podem ser utilizados para descrever processos de workflow e facilitar a automatização dos processos nas empresas.

A adequação de notações baseadas em máquinas de estados para especificação de workflow é reconhecida em muitos estudos. Em [16] argumenta-se que os profissionais da área consideram as máquinas de estados mais intuitivas e fáceis de entender do que notações formais alternativas como as redes de Petri e ainda possui uma semântica igualmente rigorosa.

Neste trabalho, entendemos que se a solução para a crise do *software* é aumentar a produtividade, então precisamos encontrar formas de criar componentes que sejam facilmente reusáveis. A base para um software mais reusável é a definição de interfaces e das dinâmicas de interações entre seus componentes.

Desta maneira, a utilização UML-RT como linguagem para a descrição de nossos padrões facilita sua utilização por desenvolvedores de software, diferentemente de outros padrões exógenos para componentes de software que possuem notações próprias, como Reo [26].

6.2 Conclusão

Os padrões selecionados para este trabalho formam os fundamentos para uma implementação bem sucedida de sistemas de Gerenciamento de *Workflow* que requerem flexibilidade, consistência e manutenibilidade de seus processos de negócios. A partir deste catálogo de padrões, Sistemas de Gerenciamento de *Workflow* poderão ser desenvolvidos mais rapidamente, uma vez que eles podem ser utilizados para estender as ferramentas de modelagem e execução de *workflows*.

Além disso, uma das maiores contribuições deste trabalho é a descrição de novos padrões para o paradigma de Desenvolvimento Baseado em Componentes. Trazidos da disciplina de *Business Process Management*, eles estão descritos com uma linguagem que demonstra como devem realmente ser realizados na prática, ao contrário de linguagens comumente usadas para descrevê-los como BPMN, XPDL e diagramas de atividades de UML.

Como unidades de *software* independentes, os componentes criados encapsulam seu projeto e implementação, oferecendo interfaces bem definidas para o meio externo. O projeto dos componentes foi conduzido de tal forma que além de tornar sua execução correta e eficiente, eles são genéricos e adaptáveis a vários propósitos definidos posteriormente nos fluxos de *workflow*.

Neste modelo, cada padrão fornece uma referência de implementação a partir dos controladores, que podem ser reusados diretamente na implementação do *workflow*, e das cápsulas auxiliares, que são modelos para a instanciação das atividades do processo. Os padrões podem ser usados sozinhos ou em combinações, permitindo que padrões mais complexos sejam construídos a partir de padrões mais simples.

Apresentamos cada um dos padrões usando diagramas de atividade de UML e derivamos a partir dele e das descrições em Redes de Petri os modelos sugeridos nos diagramas de classes, de estrutura e de estados. Estas definições foram apoiadas pela ferramenta CASE *Rational Rose RealTime* que permite a definição da estrutura e do comportamento dos padrões e a geração de código Java para execução dos padrões modelados.

Os estudos de caso selecionados demonstram a utilização dos padrões propostos em exemplos práticos de processos de negócio. A partir destes exemplos pudemos definir o modelo em UML-RT para os fluxos de trabalho existentes em uma organização de forma rápida e ágil utilizando a composição dos componentes de controle pré-definidos e instanciando os componentes de atividades conforme a referência proposta no padrão.

Com a implementação destes exemplos utilizando os padrões, observamos que a definição de um *workflow* se torna mais eficiente e compreensível para desenvolvedores, em geral já familiarizados com a sintaxe e semântica deste tipo de construção. Este facilitador cognitivo reduz a curva de aprendizagem e,

portanto facilita o processo de desenvolvimento de Sistemas de Gerenciamento de *Workflow*.

6.3 Trabalhos Futuros

Como próximos passos para o prosseguimento deste trabalho, poderemos incluir novos modelos de *Workflow Patterns* que não foram selecionados para este catálogo, como por exemplo, os padrões de *Multiple Instance*, que permitem a reconfiguração dinâmica dos componentes ativos durante sua execução e fazem parte do grupo de padrões *Control Flow*. Para completar o catálogo, poderemos também incluir os outros tipos de perspectivas de *Workflow Patterns* como os padrões *Data*, *Resource* e *Exception Handling*.

Outras especificações deste modelo também poderão ser realizadas por intermédio, por exemplo, da definição de um *profile* para UML 2.0 com conceitos e notações apropriados para a especificação de componentes reativos e com o apoio de uma ferramenta, como já é possível com UML-RT.

Futuramente, uma integração destes modelos com uma ferramenta para especificação e composição de processos de *workflow* seria uma alternativa interessante para a implementação de engenhos de *workflow*, de forma a introduzir o modelo de Desenvolvimento Baseado em Componentes nos Sistemas de Gerenciamento de Workflow.

A partir desta integração poderemos definir uma avaliação qualitativa e quantitativa dos resultados, aplicando este processo de modelagem em um ambiente real e extraindo métricas de ganho de produtividade.

APÊNDICE A

Logs das simulações dos exemplos de Workflow no *Rational Rose RealTime*

Travel Request

SEM SUCESSO

Rational Rose RealTime Java Target Run Time System
Release 6.51.C.00
Copyright (c) 2000-2003 Rational Software
MultiChoice - S1 - this.count = 0
BOOK FLIGHT
BOOK HOTEL
Synchronization - State 1 – this. count=0
SynchronizingMerge - State 1
Start
Cancel Task - State1
Cancel Task - State1
PAY
Multi Choice - Choice Point 1
Choice Point 1 - False
Multi Choice - Choice Point 2
Choice Point 2 - False – foram escolhidas CapsuleA e CapsuleB
Multi Choice envia eventoMultiChoice() para CapsuleA e CapsuleB
BOOK FLIGHT inicia execucao
BOOK FLIGHT
BOOK HOTEL iniciou execucao
BOOK HOTEL
Falha na tarefa BOOK HOTEL
BOOK HOTEL
Cancel Task – envia Cancel para capsula
Cancel Task - State1
Tarefa BOOK FLIGHT cancelada

COM SUCESSO

Rational Rose RealTime Java Target Run Time System
Release 6.51.C.00
Copyright (c) 2000-2003 Rational Software
MultiChoice - State1 - this.count = 0
BOOK FLIGHT
BOOK HOTEL
Synchronization - State 1 – this. count=0
Synchronizing Merge - State 1
Start
Cancel Task - State1
Cancel Task - State1
PAY
Multi Choice - Choice Point 1
Choice Point 1 - False
Multi Choice - Choice Point 2
Choice Point 2 - False - foram escolhidas CapsuleA e CapsuleB
Multi Choice envia eventoMultiChoice() para CapsuleA e CapsuleB
BOOK FLIGHT inicia execucao
BOOK FLIGHT
BOOK HOTEL iniciou execucao
BOOK HOTEL
BOOK FLIGHT terminou com sucesso
BOOK FLIGHT
Synchronization - T1 - count++: 1
BOOK HOTEL terminou com sucesso
BOOK HOTEL
Multi Choice - transition: finished CapsuleA e CapsuleB
Multi Choice – envia eventoSyncA() para Synchronizing Merge
Multi Choice - envia eventMerge(CapsuleA) para Multi Merge
Multi Choice - envia eventoSyncB() para Synchronizing Merge
Multi Choice - envia eventMerge(CapsuleB) para Multi Merge
MultiChoice - S1 - this.count = 0
Synchronization – recebe eventoSync
Synchronization - State 2
Synchronization – sincronizou CapsuleA e CapsuleB - count: 1
Synchronization - State 1
Synchronizing Merge - envia eventoSeq()
Synchronizing Merge - State 1
PAY triggered
PAY

Submit Form

Rational Rose RealTime Java Target Run Time System
Release 6.51.C.00
Copyright (c) 2000-2003 Rational Software
ParallelSplit - State1 - this.count=0
Start - State1
REVIEW
ARCHIVE
SEND FORM
Deferred Choice - State1
EVALUATE
Deferred Choice - State1
TIME OUT
PROCESS FORM
PROCESS COMPLAINT
Start - eventoSeq para comecar Milestone
Milestone_Controller - eventoSeq
Milestone_Controller- eventoSeq() para comecar Split
ParallelSplit - T1
eventoSplit() enviado
split SEND FORM e EVALUATE - SEND FORM
SEND FORM - start Deferred Choice
split SEND FORM e EVALUATE - EVALUATE
Deferred Choice - eventoSeq
Deferred Choice - State2
Start - eventoDeferredChoice para escolher TIME OUT
Deferred Choice - Choice Point
Choice A
Deferred Choice - State1
TIME OUT triggered
TIME OUT
SEND FORM - enviar enterMilestone() para Milestone Controller
Milestone Controller - CapsuleA está no Milestone - enviar Milestone para CapsuleB
EVALUATE recebeu milestone
Iniciou Deferred Choice - Escolher PROCESS COMPLAINT ou ARCHIVE
EVALUATE esperando escolha do Deferred Choice
Deferred Choice - eventoSeq
Deferred Choice - State2
Start - eventoDeferredChoice para escolher PROCESS COMPLAINT
Deferred Choice - Choice Point
Choice A
Deferred Choice - State1
EVALUATE esperando escolha do Deferred Choice
PROCESS COMPLAINT triggered
PROCESS COMPLAINT
Deferred Choice - eventoSeq

Deferred Choice - State2
REVIEW triggered
REVIEW
SEND FORM - enviar endMilestone() para Milestone Controller
SEND FORM
Milestone Controller - CapsuleA saiu do Milestone - enviar endMilestone para CapsuleB
EVALUATE - EndMilestone
Start - eventoDeferredChoice para escolher PROCESS COMPLAINT
Deferred Choice - Choice Point
Choice A
Deferred Choice - State1
Start - deferredChoice.eventoDeferredChoice para escolher PROCESS COMPLAINT
Start - deferredChoice.eventoDeferredChoice para escolher PROCESS COMPLAINT
Start - deferredChoice.eventoDeferredChoice para escolher PROCESS COMPLAINT

7. Bibliografia

[1] Fowler, M. *Analysis Patterns: Reusable Object Models*, 1997, Addison-Wesley

[2] van der Aalst W.M.P., ter Hofstede A.H.M., Kiepuszewski B., e Barros A.P. *Workflow Patterns*. Distributed and Parallel Databases, vol. 14, n01, pp. 5-51, 2003.

[3] Gamma E., Helm R., Johnson R., e Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[4] van der Aalst W.M.P., ter Hofstede A.H.M., Kiepuszewski B., and Barros A.P. *Workflow Patterns Home Page*.
Disponível em <http://www.workflowpatterns.com>

[5] TOGAF™ - *The Open Group Architecture Framework*, Architecture Patterns. Disponível em
<http://www.opengroup.org/architecture/togaf8-doc/arch/chap28.html>

[6] OpenWFE - O*pen source Workflow Engine*.
Disponível em <http://www.openwfe.org>

[7] van der Aalst W.M.P., ter Hofstede A.H.M., Kiepuszewski B., e Weske M. *Business Process Management: A Survey*. Business Process Management, Proceedings of the First International Conference. Springer Verlag, 2003.

[8] WFMC.org Homepage, *WFMC.org Standards – Reference Model*.
Disponível em <http://www.wfmc.org/standards/referencemodel.htm>

[9] Ferreira, P. M. *Geração Automática de Modelos UML-RT a partir de Especificações CSP*. 2006.

[10] Sun, Y., Levy, D. *Suggestions on Pattern Transformation in UML-RT*, *icsea*, p. 17, International Conference on Software Engineering Advances, 2007.

[11] Douglass, B. P. *Real-Time Uml: Developing Efficient Objects for Embedded Systems*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[12] Rational/IBM. *Rose Real-time Development Environment*.

[13] WfMC. *Workflow Management Coalition - Terminology and Glossary*. Technical report, The Workflow Management Coalition. Document Number WFMC-TC-1011, 1999.

[14] Hollingsworth, D. *Workflow Management Coalition - The Workflow Reference Model*. Document Number TC00-1003. Document Status - Issue 1.1. 19-Jan-95.

Disponível em <http://www.wfmc.org/standards/docs/tc003v11.pdf>

[15] OMG. *UML 2.0 superstructure specification*, version 2.0, documents ptc/03-08-02 and ptc/04-10-02. Object Management Group, 2004.

[16] The Object Management Group. UML Extensions for Workflow Process Definition, RFP-bom/2000-12-11.
Disponível em <ftp://ftp.omg.org/pub/docs/bom/00-12-11.pdf>

[17] Aalst, W.M.P. van der. *The application of Petri nets to workow management*. The Journal of Circuits, Systems and Computers, 8(1):21{66, 1998.

[20] Dumas, M. and Hofstede, A. H. *UML Activity Diagrams as a Workflow Specification Language*. In *Proceedings of the 4th international Conference on the Unified Modeling Language*, Modeling Languages, Concepts, and Tools (October 01 - 05, 2001). M. Gogolla and C. Kobryn, Eds. Lecture Notes In Computer Science, vol. 2185. Springer-Verlag, London, 76-90, 2001.

[21] C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[22] Faison, T. *Interaction Patterns for Communicating Processes*. The Conference on Pattern Languages of Programs (PLoP 98), 1998.

[23] Eskelin, P. *Component Interaction Patterns*. In Proceedings of the Conference on the Pattern Languages of Programs (PLOP '99), 1999.

[24] Oliveira , K. R. and Pessôa, M. S. P. Implementando a linguagen UML como uma ferramenta de definição de processo de workflow.
Disponível em < http://www.imageware.com.br/down/paper_wml_wf.pdf>

[25] Papadopoulos, G. A. and Arbab, F. *Coordination Models and Languages*, in Advances in Computers, Academic Press, vol 46, 1998.

[26] Arbab, F. *Reo: a channel-based coordination model for component composition*. *Mathematical. Structures in Comp. Sci., Cambridge University Press, vol* 14, 2004, 329-366

[27] Transflow Nederland BV, Workflow Patterns, 2003. Disponível em <http://is.tm.tue.nl/bpm2003/download/COSA%20workflow%20patterns%202003.PDF>

[28] Wohed P., van der Aalst W.M.P., Dumas, M., ter Hofstede A.H.M., and Russell, N. *Pattern-based Analysis of UML Activity Diagrams*. BETA Working Paper Series, WP 129, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.
Disponível em <http://www.bpm.fit.qut.edu.au/projects/babel/docs/p242.pdf>

[29] Russell, N. and van der Aalst, W.M.P. , ter Hofstede A. H.M. , Wohed P. *On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling*. Disponível em: <http://citeseer.ist.psu.edu/cache/papers/cs2/468/http:zSzzSzcrpit.comzSzconfpaperszSzCRPITV53Russell.pdf/russell06suitability.pdf

[30] Douglass, B. P. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* Pearson. ISBN: 0201699567

[31] Selic, B. *Using UML for Modeling Complex Real-Time Systems*. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools For Embedded Systems F. Mueller and A. Bestavros, Eds. Lecture Notes In Computer Science, vol. 1474. Springer-Verlag, London, 1998, 250-260.