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

2008.1

TRABALHO DE GRADUAÇÃO

UMA ESTRATÉGIA PARA MONITORAMENTO DO FLUXO DE EXECUÇÃO DE SERVIÇOS

DAVID MENEZES CARDOSO

ORIENTADOR: AUGUSTO CEZAR ALVES SAMPAIO (acas@cin.ufpe.br)

RECIFE
JUNHO DE 2008

"A dúvida é o princípio da sabedoria."
– ARISTÓTELES
(384 A.C. - 322 A.C.)

A meus pais.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a meus pais pelo amor e carinho que nunca me faltaram e pelo apoio que me permitiu seguir em frente nos momentos difíceis

A meus irmãos, em especial a Daniel, que com seu senso crítico aguçado, contribui diariamente para que eu esteja em processo de autocrítica contínua.

À minha namorada Rebeca, pela paciência ao me escutar falar sobre este trabalho – muitas vezes em momentos inapropriados ☺ –, pela inspiração que proporciona e pela confiança depositada em meu potencial.

Aos professores do Centro de Informática, através dos quais obtive grande parte do meu aprendizado em computação.

Ao professor e orientador deste trabalho, Augusto Sampaio, pelo apoio, disponibilidade e paciência. Por ter me dado a oportunidade de desenvolver este trabalho e ter me ajudado a manter a calma à medida que o prazo de entrega se aproximava.

A Belo, companheiro de trabalho e amigo, pela compreensão, apoio e cobertura nos momentos em que precisei me ausentar para trabalhar em atividades acadêmicas ao longo de toda a graduação.

Por fim, aos meus amigos do Centro de Informática e companheiros de projetos desenvolvidos ao longo da graduação, Breno Neil, Caio Tizei, Eliaquim Lima, Djaci Alves, Nicole Sultanum, Rodrigo Lumack, Thiago Pacheco, membros da Commit Solutions e todos aqueles com quem me relacionei ao longo do curso e contribuíram para a minha formação.

RESUMO

Uma demanda que tem se tornado comum nas aplicações comerciais mais recentes é a de permitir, ao usuário, acompanhar com detalhes o progresso e o estado do fluxo de execução de um serviço requisitado ao sistema. Para realizar esta tarefa da maneira adequada, este requisito deve ser considerado durante a etapa de definição da arquitetura do sistema. Caso contrário, pode-se incorrer na geração de código de difícil manutenção e elevar, desnecessariamente, o grau de complexidade da aplicação. Este problema se agrava ainda mais quando se trata de aplicações inerentemente complexas, cujas operações envolvem o uso de diversos componentes integrados, com muitos trechos de código reusáveis e métodos executados de forma encadeada.

Nesse contexto, este trabalho propõe um framework que traz facilidades para o desenvolvimento de aplicações com suporte a monitoramento. Em particular, duas bibliotecas de componentes são desenvolvidas utilizando o framework: uma oferece suporte à programação dos serviços com monitoramento e outra oferece suporte à visualização das informações de monitoramento. O uso destas bibliotecas é ilustrado através de uma aplicação.

As bibliotecas foram implementadas utilizando a Plataforma .Net e a linguagem C#, da Microsoft.

Palavras-chave: Monitoramento, fluxo de execução, serviços, arquitetura de sistema, *framework*.

ABSTRACT

Newly developed commercial applications commonly allow users to monitor running services progress details and the state of execution flow. In order to properly accomplish this task, one should consider monitoring requirements while defining system architectures. Otherwise, ignoring this requirement in an early development stage would lead to generation of code hard to maintain, undesired workarounds and unnecessary increase of application complexity. The more inherently complex the application, the harder it is to add these monitoring capabilities. Services may involve use of many different integrated components, reusable code or a complex method invocation chain.

This work proposes a framework which provides development facilities and support for building applications with monitoring capabilities. Furthermore, two component libraries are developed on top of this framework: one provides support for monitorable programming services while the other provides support for visualization of monitoring information. The use of these libraries is illustrated by the development of a sample application.

The implementation is based on the Microsoft .Net Framework and C# language.

Keywords: Monitoring, execution flow, services, system architecture, framework.

SUMÁRIO

1 INTRODUÇÃO 1

1.1 OBJETIVOS E DESAFIOS 4
1.2 TRABALHOS RELACIONADOS 5
1.3 VISÃO GERAL 7

2 FRAMEWORK DE MONITORAMENTO 9

2.1 CONCEITOS 9
2.2 REQUISITOS 12
2.3 DIAGRAMAS DE CASOS DE USO 13
2.4 ESTRUTURA DO FRAMEWORK 15
2.5 EVENTOS NA LINGUAGEM C# 15
2.6 PACOTE OPERATIONS 16
2.6.1 Classe Operation 16
2.6.2 Estrutura ProgressStatus 17
2.6.3 Enumeração EOperationStatus 18
2.7 PACOTE NOTIFICATIONS 19
2.7.1 Classe Notification 20
2.7.2 Enumeração ENotificationStatus 21
2.7.3 Classe SimpleNotification 21
2.7.4 Classe OperationBoundNotification 21
2.8 PACOTE INTERFACES 22
2.8.1 Interface INotificationsProvider 22
2.8.2 Interface IServicesMonitor 23
2.8.3 Interface IMonitorableServicesProvider 23
2.9 IMPACTO NA ARQUITETURA DA APLICAÇÃO 23

3 BIBLIOTECAS SERVICEMON PROVIDERTOOLKIT E CONSUMERTOOLKIT 26

3.1 BIBLIOTECA SERVICEMON PROVIDERTOOLKIT 27
3.1.1 A interface IDisposable e a instrução using 27
3.1.2 A classe ServicesMonitor 29
3.1.3 Contextos de Operação 31
3.1.3.1 ManagedOperationContext 32
3.1.3.2 OperationContextBinding 35
3.1.3.3 LinkedOperationContext 36
3.1.3.4 Operações recursivas 38
3.1.4 Notificações 39
3.1.4.1 ManagedOperationNotification 40
3.1.4.2 ManagedSimpleNotification 41
3.1.5 Cancelamento 41
3.2 BIBLIOTECA SERVICEMON.CONSUMERTOOLKIT 42
3.2.1 Componente ProgressNotifier 43
3.2.2 Componente NotificationsViewer 44

4 APLICAÇÃO DAS BIBLIOTECAS 45

4.1 VISÃO GERAL 45
4.2 CLASSE *SERVICESPROVIDER* 47
4.2.1 Método Service1 ..... *47*
4.2.2 Método Service2 ..... *48*
4.2.3 Método Service3 ..... *49*

5 CONCLUSÃO 51

5.1 RESULTADOS OBTIDOS E CONTRIBUIÇÕES 51
5.2 DIFICULDADES E LIMITAÇÕES 52
5.3 TRABALHOS FUTUROS 53

APÊNDICE A – NOTAÇÃO DOS DIAGRAMAS 55

LISTA DE FIGURAS

Figura 2.1 – Casos de uso do Consumidor 13
Figura 2.2 – Casos de uso do Provedor 13
Figura 2.3 – O *framework* e sua relação com provedor/consumidor de serviços 14
Figura 2.4 – Diagrama de subpacotes do pacote *Fundamentals* 15
Figura 2.5 – Classes do pacote *Operations* 16
Figura 2.6 – Diagrama mudança de estados da operação 19
Figura 2.7 – Classes do pacote *Notifications* 20
Figura 2.8 – Interfaces do monitoramento 22
Figura 2.9 – Exemplo de fachada tradicional, não monitorável 24
Figura 2.10 – Interface da fachada adequada ao modelo de monitoramento 25
Figura 3.1 – Dependências entre bibliotecas e *framework* 26
Figura 3.2 – Interface *IDisposable* 28
Figura 3.3 – Exemplo de código que utiliza classe *IDisposable* 28
Figura 3.4 – Exemplo utilizando instrução *using* 28
Figura 3.5 – Classe *ServicesMonitor* 29
Figura 3.6 – Classes para representação de contextos de operação 32
Figura 3.7 – Exemplo de serviço com operações 35
Figura 3.8 – Exemplo de chamada a outro serviço da mesma classe 36
Figura 3.9 – Exemplo de chamada a outro serviço monitorável em outra entidade 37
Figura 3.10 – Exemplo de criação de operação recursiva 39
Figura 3.11 – Diagrama de classes que representam notificações 40
Figura 3.12 – Exemplo serviço gerando notificações de alerta 41
Figura 3.13 – Estrutura de um serviço passível de cancelamento 42
Figura 3.14 – Componentes da biblioteca *ConsumerToolkit* 43
Figura 3.15 – Componente *ProgressNotifier* 43
Figura 3.16 – Componente *NotificationsViewer* 44
Figura 4.1 – Entidades das camadas de serviços e dados 46
Figura 4.2 – Interface gráfica da aplicação 46
Figura 4.3 – *Loop* gerando notificações 47
Figura 4.4 – *Loop* com atualização dinâmica do progresso 47
Figura 4.5 – Execução de serviços encadeados na mesma classe 48
Figura 4.6 – Encadeamento do serviço 1 dentro do serviço 2 48
Figura 4.7 – Execução de serviço em uma outra camada 49
Figura 4.8 – Criação de suboperação com custo dinâmico 49
Figura 4.9 – Exemplo de método recursivo, fatorial 50
Figura 4.10 – Árvore de notificações do método recursivo 50

CAPÍTULO 1

INTRODUÇÃO

Com o crescente surgimento de novas tecnologias e a intensa difusão dos sistemas computacionais na sociedade, as pessoas passaram a exigir da indústria de software a produção de sistemas cada vez mais complexos e de alta qualidade. Estes sistemas fornecem aos seus usuários uma vasta gama de serviços que, freqüentemente, apresentam um alto grau de complexidade e podem envolver a realização combinada de operações de diferentes subsistemas, acesso a informações da Internet, realização de transações sobre bancos de dados ou processamento de algoritmos custosos que fazem com que estes serviços não possam ser concluídos instantaneamente.

A execução de um serviço pode requerer mais tempo de processamento do que o usuário esperava. Nesses casos, é interessante que a aplicação permita monitorar o andamento do fluxo de execução do serviço, em tempo real, à medida que os algoritmos são processados. A noção de indicadores de progresso não é nova em sistemas computacionais: eles são largamente utilizados em operações de transferência de arquivos na rede, instalação de aplicativos e outros tipos de serviços relacionados com transferência de dados[1]. Esta propriedade de permitir o monitoramento da execução tem se sofisticado e já é bastante comum, nos mais variados tipos de serviços e aplicações recentes.

Inserido neste contexto, este trabalho apresenta uma abordagem geral do problema da construção de aplicações passíveis de monitoramento e propõe uma solução, implementada com base na *Plataforma .Net*, da *Microsoft*, capaz de eliminar muitos dos problemas inerentes ao desenvolvimento deste tipo de aplicação, reduzir a quantidade de esforço de programação requerido e, conseqüentemente, reduzir o tempo.

O aumento da demanda pela construção de sistemas de grande porte, com exigências que os tornam continuamente mais complexos, representa um desafio para a Engenharia de Software no mundo inteiro e, em particular, para a área de Arquitetura de Software. Esta, por sua vez, precisa alinhar-se às novas necessidades através da

criação de novas tecnologias, padrões e anti-padrões de projeto e arquiteturais, paradigmas de desenvolvimento, *frameworks*, entre outros.

Decisões de arquitetura acontecem bem no início do ciclo de vida do projeto e, por isso, suas modificações podem causar um grande impacto no mesmo. Assim, essas decisões são as mais críticas e influentes no desencadear das outras etapas do projeto[2]. Por isso, é importante dimensionar adequadamente o tempo para que esta etapa seja realizada com qualidade. Ao trabalhar nesta atividade, o arquiteto deve considerar todos os requisitos e restrições do software, avaliando se a arquitetura do sistema pode ser previamente preparada para atender ao requisito dado.

A definição da arquitetura de uma aplicação é um processo criativo e, portanto, consiste de atividades que variam significativamente dependendo do tipo de sistema que está sendo desenvolvido e da experiência do arquiteto envolvido. Assim, hoje, este processo não é passível de automação e precisão. Por exemplo, é inviável criar um procedimento que, dados os requisitos e restrições de uma aplicação como entrada, gere, como saída, a especificação de uma arquitetura ótima para o sistema. Um procedimento deste tipo exigiria um grau de formalismo muito elevado na especificação dos requisitos e restrições do projeto.

É uma boa estratégia fazer uso de padrões e modelos catalogados, com eficiência comprovada pelo uso bem sucedido em outros projetos, a fim de evitar a ocorrência de erros de concepção de arquitetura bem conhecidos, já tratados anteriormente por outros arquitetos[3]. O mesmo princípio que motiva o reuso de padrões e modelos também vale, em geral, para componentes de software, incluindo *frameworks* e bibliotecas que, quando bem escolhidos e integrados corretamente, evitam retrabalho e simplificam o desenvolvimento de novas aplicações.

Infelizmente, novos problemas surgem constantemente e nem sempre é possível encontrar um modelo ou componente de software aderente às necessidades da aplicação. A impossibilidade de reusar o conhecimento de terceiros em projetos anteriores tem um grande impacto no desenvolvimento da aplicação: aumenta esforço, tempo e custo necessários para a sua construção, compromete a produtividade[4] e aumenta o risco natural da ocorrência de erro de concepção, uma vez que se trata da implementação de uma idéia nova, possivelmente complexa, desenvolvida sob

demanda, nunca antes testada e validada por outros arquitetos, como é o caso dos modelos e componentes catalogados[5].

No caso do monitoramento da execução de serviços da aplicação, o impacto na análise da arquitetura é claro: precisamos criar estruturas de controle que permitam armazenar o contexto da execução dos serviços e permitir a troca de informação sobre o *status* da execução de cada rotina processada, em tempo real, entre os módulos, subsistemas e componentes que fazem parte da aplicação.

Apesar do elevado grau de maturidade em que se encontra a disciplina de Engenharia de Software, no que se refere à variedade de padrões de projeto catalogados, modelos de arquitetura, componentes de software reusáveis, *frameworks* e linguagens de programação voltadas para resolução de problemas de domínio específico [4], o material disponível e concebido especificamente para resolução do problema do monitoramento ainda é escasso. Há uma carência de soluções completas para o fim de suportar o monitoramento da execução de serviços das aplicações.

Dentro desse contexto, este trabalho visa à concepção de um modelo e implementação de componentes de software que possam ser facilmente reusados por terceiros e dêem suporte ao monitoramento da execução de serviços.

Este trabalho tem um escopo genérico: não queremos limitar o modelo desenvolvido a tipos específicos de arquitetura ou padrões de projeto, sabendo que existem muitas aplicações de naturezas diferentes, que utilizam uma arquitetura peculiar. Por isso mesmo, temos que construir um modelo genérico que possa ser facilmente utilizado por aplicações típicas, em diferentes tipos de arquitetura e com poucas limitações.

A necessidade de criar uma solução genérica, passível de extensão e customização, determinam a escolha de um *framework* como estratégia para implementação de uma solução de monitoramento [6]. Assim, podemos representar apenas os conceitos básicos relacionados e interfaces simples que precisam ser implementadas nas aplicações com suporte a monitoramento.

1.1 OBJETIVOS E DESAFIOS

O principal objetivo deste trabalho consiste em desenvolver um *framework* baseado na *Plataforma .Net* e na linguagem C#, genérico, expansível, aderente a diversos tipos de arquiteturas de aplicações, que proporcione facilidade e dê suporte ao desenvolvimento de aplicações que atendam aos requisitos de monitoramento. Além disso, desenvolver duas bibliotecas específicas, baseadas no *framework*. Uma das bibliotecas deve fornecer suporte à programação dos serviços com monitoramento. A outra biblioteca deve fornecer componentes que permitam a visualização das informações de monitoramento dos serviços em execução. Existem alguns desafios associados ao atendimento deste objetivo.

Ao avaliar a gama de aplicações que operam hoje no mercado, deparamo-nos com os mais variados tipos de sistema de informação. São muitas as possibilidades de configuração de ambiente e naturezas de aplicação. Por isso é muito difícil desenvolver um *framework* que seja ao mesmo tempo estritamente simples e suficientemente genérico para atender a todos os casos. Embora o *framework* proposto neste trabalho tenha caráter genérico e funcione bem para aplicações típicas, pode ser estendido e melhorado para utilização em aplicações de natureza específica.

Além disso, alguns serviços de aplicações têm natureza inerentemente não-determinística e aleatória que impedem a previsão das etapas necessárias para realização de um serviço. Nesses casos, as operações que serão realizadas podem mudar no decorrer da execução do serviço, tornando complexo o monitoramento do indicador de progresso que mede o percentual de execução concluído, uma vez que as operações restantes não ficam previamente definidas em função das entradas do serviço. O uso dos componentes desenvolvidos neste trabalho não elimina a complexidade inerente a este problema.

O funcionamento de um programa orientado a objetos acontece através da interação entre os diversos objetos instanciados no sistema (entidades da aplicação). Esta interação acontece através de chamadas de métodos (solicitações de serviços) realizados no sistema. Assim, entidades possuem serviços (métodos públicos) que podem ser consumidos por outras entidades que se relacionem com ela. Por exemplo, em uma aplicação com arquitetura em camadas que utilize o padrão de projetos *Facade*, os

principais serviços oferecidos por cada camada estarão representados através dos métodos públicos existentes na Fachada [3]. Esta visão do sistema, como um conjunto de serviços fornecidos e consumidos por entidades [7] [8], é fundamental para o escopo deste trabalho, uma vez que é justamente o fluxo de execução destes serviços que desejamos monitorar. Para ter um acompanhamento detalhado do estado da execução de um serviço, é necessário utilizar estruturas de dados que gerenciem a pilha de execução da aplicação e atualizem o estado do progresso à medida que os métodos encadeados são executados.

O elemento central do monitoramento é entidade provedora de serviços da aplicação. Uma vez que todas as entidades envolvidas em um serviço estiverem passíveis de monitoramento, a integração entre elas deve ser simples. A programação de cada entidade deve ser independente das outras. Ou seja, o desenvolvedor responsável por implementar um serviço de uma entidade não precisa conhecer detalhes da implementação de outras entidades para torná-lo monitorável. Isto é particularmente importante porque, em projetos grandes, tipicamente, existem muitas pessoas envolvidas nas atividades de desenvolvimento e cada uma é responsável por trabalhar em um pequeno conjunto de serviços e entidades da aplicação.

1.2 TRABALHOS RELACIONADOS

Rao e Rajaraman desenvolveram uma metodologia para a realização da estimativa do tempo restante para transferência de dados entre pontos distantes em uma rede [1]. O foco do trabalho foi a criação de um modelo matemático para a realização desta estimativa, que levava em consideração o tempo de execução das operações envolvidas na transferência. Embora tenhamos propósitos bem diferentes, parte da abordagem utilizada neste modelo pode ser aproveitada neste trabalho. Em especial, a abordagem que permite a visualização de um serviço como um conjunto de operações encadeadas, que será aproveitada neste trabalho.

Existem muitas ferramentas de *Profiling* no mercado. Em especial, para aplicações desenvolvidas em *Java* e para a *Plataforma .Net* [9] [10]. Estas ferramentas são bastante completas e permitem o monitoramento detalhado das aplicações. Elas permitem acompanhar todo o fluxo de execução dos métodos em uma aplicação, aferir tempos de execução, identificar gargalos, utilização da memória e do processamento, uso de

recursos do computador, etc.[11]. Embora sejam ferramentas de monitoramento bastante completas, não resolvem o problema proposto neste trabalho. Estas são ferramentas voltadas para desenvolvedores e analistas responsáveis pela realização de testes detectarem problemas em uma aplicação. Uma vez desenvolvido o produto, estas ferramentas são utilizadas separadamente para monitorar a execução dos mesmos. O monitoramento é realizado sobre métodos e não sobre o conceito mais alto nível de operações. As informações prestadas por essas ferramentas são estritamente técnicas e não têm muita utilidade para um usuário típico. Ainda assim, apesar das diferenças, o conhecimento sobre técnicas de *Profiling* nos ajuda a identificar o formato do fluxo de execução das operações nas aplicações e desenvolver um modelo análogo ao utilizado por estas ferramentas, que se baseiam na construção de árvores de execução de métodos. Aproveitaremos esta abordagem para a representação de uma árvore de contextos de operação.

As ferramentas de *Profiling* também podem ser utilizadas para auxiliar o programador na aferição dos custos de execução das operações realizadas pela aplicação. Com estas ferramentas, o programador pode medir o tempo de execução de cada operação e utilizá-los como base para parametrizá-las no *framework*.

Para a geração de notificações para o usuário, em tempo de execução, existem algumas ferramentas desenvolvidas que permitem a criação de *log* com mensagens geradas à medida que os serviços são executados [12]. Por exemplo, a classe EventLog da *Plataforma .Net* [13] e a interface *ILog* do *framework* Okapi [14]. Normalmente, o *log* gerado por estas ferramentas é armazenado em um arquivo que precisaria ser acessado posteriormente pelo usuário. Além disso, essas ferramentas apresentam as mensagens para o usuário de forma plana, como uma seqüência de mensagens descontextualizadas da operação. Cada mensagem possui associada apenas a data e hora de sua emissão. Em nosso trabalho, associaremos cada mensagem de notificação ao contexto de operação em que ela foi executada. O resultado, para o usuário, será a visualização de uma árvore de notificações onde será possível perceber uma hierarquia entre as mensagens.

Vamos aproveitar partes destes trabalhos relacionados que sejam aderentes ao escopo do nosso trabalho.

1.3 VISÃO GERAL

O capítulo 2 aborda o desenvolvimento do *framework* de monitoramento, baseado na *Plataforma .Net*, da *Microsoft*, através da introdução de conceitos envolvidos com a problemática de monitoramento, apresentação de detalhes técnicos da arquitetura e diagramas de classe do projeto.

O capítulo 3 aborda o desenvolvimento de duas bibliotecas baseadas no *framework* apresentado no capítulo anterior. As decisões de projeto são apresentadas detalhadamente através de diagramas de classe e exemplos de código fonte.

No capítulo 4, a fim de reforçar a eficácia e a aplicabilidade dos modelos desenvolvidos, é apresentado um exemplo de aplicação construída especificamente com o intuito de utilizar os recursos dos componentes abordados nos capítulos anteriores. Serão apresentados trechos de código fonte para um melhor entendimento.

Por fim, no capítulo 5, apresentamos os resultados finais obtidos, as limitações identificadas e as sugestões de possíveis trabalhos futuros, relacionados com o problema do monitoramento, que poderão ser desenvolvidos para resolver limitações de projeto e aumentar a abrangência dos componentes criados.

CAPÍTULO 2

FRAMEWORK DE MONITORAMENTO

Neste capítulo, apresentamos o desenvolvimento de um *framework* voltado especificamente para permitir o suporte ao monitoramento da execução de serviços das aplicações.

O objetivo do *framework* é mapear todos os conceitos relacionados com o problema de monitoramento em classes e interfaces fundamentais que devem ser reusadas/estendidas por aplicações em que se deseje ter o monitoramento da execução. O *framework* abstrai heurísticas e algoritmos específicos para a resolução do problema, que podem variar entre diferentes aplicações e, se necessário, seriam implementados através da realização de extensões sobre as classes do *framework*, que tem o propósito de ser genérico e servir de base para qualquer tipo de aplicação em que se queira ter monitoramento.

A construção deste *framework* representa o núcleo essencial deste trabalho e será fundamental para o desenvolvimento das bibliotecas e componentes de software apresentados nos capítulos posteriores.

Assim, neste capítulo, vamos detalhar os principais conceitos envolvidos no problema de monitoramento, definir os principais requisitos e casos de uso do *framework* e, finalmente, diagramas de classes e interfaces que demonstram as principais entidades e abstrações criadas, como elas se relacionam no modelo e qual é o propósito de sua existência no contexto da aplicação.

2.1 CONCEITOS

Para entender os conceitos envolvidos no problema do monitoramento, é importante resgatar a noção introduzida no capítulo anterior, em que uma aplicação pode ser vista como um conjunto de serviços fornecidos e consumidos por entidades. É exatamente o fluxo de execução destes serviços que desejamos monitorar. Por esta visão,

temos claramente os papéis de Consumidor e Provedor dos serviços nas aplicações e é exatamente a relação entre eles que determinará os requisitos do *framework* e a arquitetura de software capaz de representar bem os conceitos envolvidos.

Freqüentemente, como conseqüência natural do paradigma de programação orientada a objetos, a implementação de um serviço envolve a interação entre diversas entidades e o consumo, de forma encadeada, de outros serviços menores. Esta noção de serviço como combinação de outros serviços menores também deve ser considerada em nosso modelo, uma vez que, para monitorar a execução de um serviço da aplicação, também será necessário monitorar, indiretamente, todos os serviços executados em cadeia. O principal desafio é quando a execução de um serviço envolve, de forma direta ou indireta, o próprio serviço (de forma recursiva).

O *framework* proposto neste trabalho pode ser utilizado pelos vários programadores da aplicação com serviços monitoráveis. O programador responsável pela codificação de um dos serviços da aplicação reusa outros serviços implementados por programadores diferentes. Neste contexto, o *framework* precisa permitir que cada programador possa abstrair os detalhes de codificação do monitoramento dos serviços implementados por terceiros.

Quase sempre, é possível dividir a execução de um serviço, logicamente, em *Operações* cujo monitoramento pode ser acompanhado pelo consumidor. Por exemplo, um serviço de uma aplicação que atualiza um banco de dados através da realização de consultas à internet pode ser logicamente dividido em várias *Operações* cuja execução seria monitorada pelo usuário: estabelecer conexão com a internet; realizar autenticação no *site*; realizar download das informações; processar informações; estabelecer conexão com banco de dados; criar transação com banco de dados; inserir registros nas tabelas do banco de dados, e assim por diante.

É importante perceber que nem sempre existe uma relação direta entre esta visão do serviço em *operações* e a estratégia de modularização utilizada pelos programadores para implementar o serviço. Ou seja, tipicamente, não haverá um mapeamento um-para-um entre as operações que compreendem o ponto de vista do usuário e os serviços e métodos internos utilizados pelos arquitetos e programadores na modularização do sistema. Assim, uma operação pode ser entendida como um trecho da implementação de

um serviço, podendo contemplar a chamada de vários pequenos serviços de outras entidades ou um conjunto de instruções e estruturas de linguagem que não sejam métodos modularizados (*loops*, condições, atribuições, etc.).

Precisamos criar classes para representar o conceito de *Operação* em nosso *framework* e permitir o seu gerenciamento e controle pela aplicação.

Uma operação leva um tempo para ser executada. No contexto de um serviço, cada operação tem um custo associado. Durante a execução de um serviço, as mudanças de estado das operações realizadas precisam ser controladas. Pelo menos dois aspectos precisam ser gerenciados:

(1) Controle do status – permitir o controle sobre qual é o status da operação (parada, em execução, com solicitação de cancelamento, cancelando, cancelada, interrompida e concluída);
(2) Controle do progresso – permitir o controle sobre o percentual de progresso realizado pela operação;

É comum que a execução de um serviço tenha como objetivo a geração de informações requisitadas pelo usuário. Durante a execução desses serviços, é possível que a aplicação tenha passado por situações não convencionais ou simplesmente já tenha processado uma parte das informações que poderiam ser disponibilizadas para o usuário antes do término completo da execução do serviço.

Assim, é importante que o consumidor receba *Notificações* (mensagens informativas, alertas e erros encontrados durante o processamento) à medida que as informações são processadas. Nesse contexto, nosso *framework* deve facilitar a emissão assíncrona das notificações sobre o andamento da execução das operações realizadas pelo serviço.

Notificações são geradas à medida que as operações são executadas. Como o processamento das operações ocorre de forma encadeada, devemos considerar a existência de uma hierarquia para as notificações. Cada notificação pode ter um conjunto de notificações filhas, associadas em função do encadeamento das operações. Precisaremos criar classes para representar o conceito de *Notificação* em nosso *framework* e permitir o seu gerenciamento e controle pela aplicação.

Além disso, nosso *framework* precisa dar suporte ao cancelamento das operações executadas. Por exemplo, o consumidor pode requisitar a execução de um serviço e, posteriormente, enquanto o serviço ainda estiver em execução, solicitar o cancelamento da operação corrente. Para que isto seja possível, o provedor de serviços precisa estar preparado para detectar as solicitações de cancelamento e realizar os procedimentos necessários para abortar a operação corrente. O nosso *framework* deve ter estruturas de controle que facilitem o desenvolvimento de serviços passíveis de cancelamento.

Precisamos construir o *framework* de maneira que seja facilmente utilizado e integrado à arquitetura da aplicação. Se tivermos uma aplicação inteiramente construída sem a preocupação com o monitoramento, deve ser fácil adequar o código fonte para utilizar o *framework*.

2.2 REQUISITOS

Com base na experiência adquirida ao longo de anos de desenvolvimento de projetos de software e no conhecimento adquirido ao longo da graduação no curso de Ciência da Computação, os seguintes requisitos funcionais foram levantados:

[RF01] Permitir o acompanhamento do progresso de execução dos serviços;

[RF02] Permitir o recebimento de notificações durante a execução dos serviços;

[RF03] Fornecer estrutura que dê suporte ao cancelamento da execução;

[RF04] Permitir o acompanhamento e controle das operações realizadas;

[RF05] Permitir o monitoramento das operações executadas em cadeia;

[RF06] Permitir um serviço seja monitorado por várias entidades.

Alguns requisitos não funcionais também foram elicitados:

[RNF01] A emissão de notificações deve ser realizada assincronamente, sem interferir no fluxo de execução do serviço.

[RNF02] O modelo deve ser fácil de integrar às arquiteturas de aplicações existentes e não deve aumentar significativamente a complexidade da programação dos serviços.

[RNF03] A emissão de notificações e controle do *status* das operações não deve interferir significativamente no desempenho do serviço.

2.3 DIAGRAMAS DE CASOS DE USO

Os diagramas da Figura 2.1 e Figura 2.2 ilustram os principais casos de uso do *framework*, em conformidade com os conceitos apresentados e requisitos levantados e descritos na seção anterior.

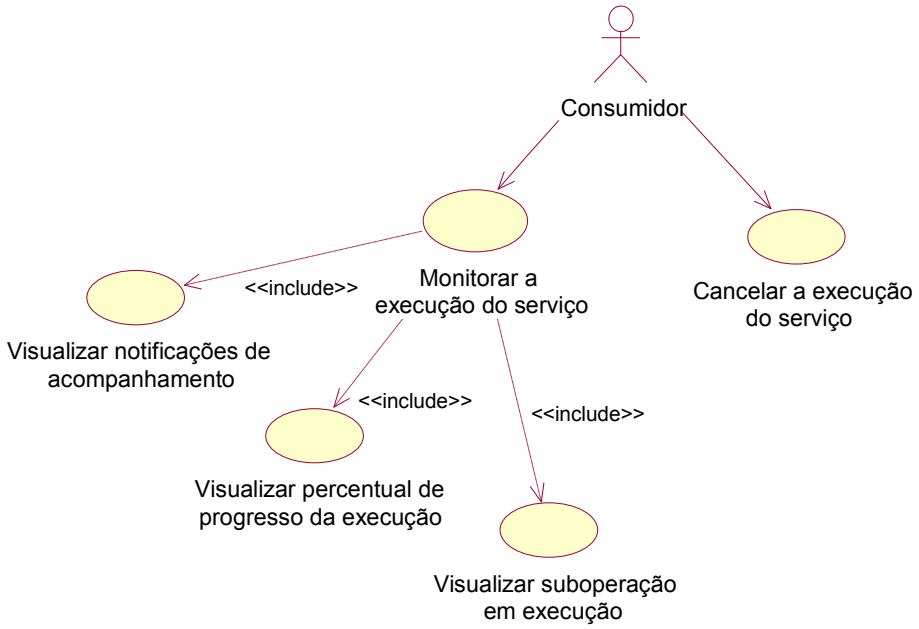


Figura 2.1 – Casos de uso do Consumidor

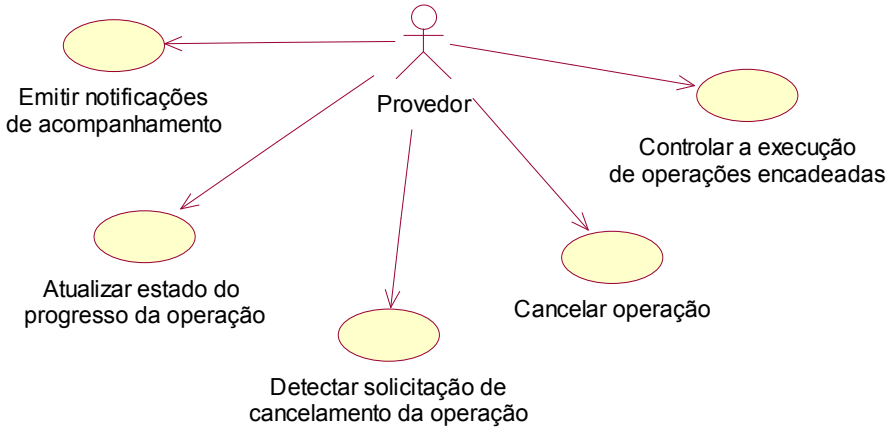


Figura 2.2 – Casos de uso do Provedor

Claramente, como pode ser percebido nos diagramas de caso de uso apresentados, o nosso modelo deverá dar suporte ao monitoramento sob dois pontos de vista:

(1) Do provedor de serviços – fornecimento do conjunto de componentes necessários para que o programador implemente serviços passíveis de monitoramento.

(2) Do consumidor de serviços – fornecimento do conjunto de componentes necessários para que um serviço possa ser monitorado e as informações resultantes do monitoramento sejam percebidas.

De maneira aderente a ambos os pontos de vista, de consumidor e provedor de serviços, precisamos especificar um conjunto de classes capazes de mapear os conceitos fundamentais discutidos, inerentes ao modelo. Além disso, precisamos estabelecer um conjunto de interfaces que definam o padrão que deve ser implementado por classes com serviços monitoráveis e permitam, ao consumidor, acessar as informações de monitoramento.

Vamos chamar nosso *framework* com classes e interfaces fundamentais para o suporte ao monitoramento de *ServiceMon Fundamentals* (codinome para *Service Monitoring Fundamentals*).

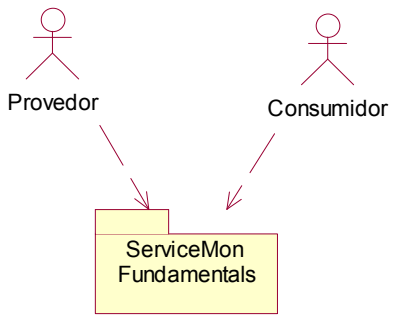


Figura 2.3 – O *framework* e sua relação com provedor/consumidor de serviços

A seguir, explicaremos como o *ServiceMon.Fundamentals* foi estruturado em pacotes e descreveremos cada uma das entidades (classes, interfaces e enumerações) que compõem o modelo.

2.4 Estrutura do framework

O *framework* está organizado em três pacotes independentes – Operations, Notifications e Interfaces –, conforme pode ser visualizado na figura a seguir.

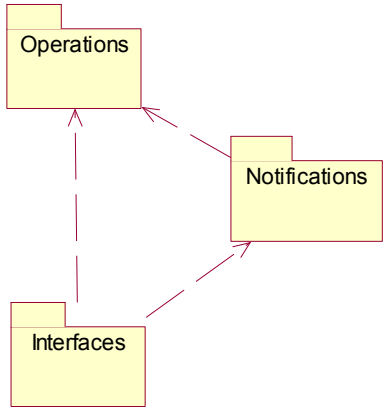


Figura 2.4 – Diagrama de subpacotes do pacote *Fundamentals*

As seções seguintes descrevem cada um dos subpacotes ilustrados.

Por ser baseado na *Plataforma .Net*, da *Microsoft*, nosso *framework* utiliza estruturas que não fazem parte do padrão UML: propriedades, eventos, *structs* e enumerações. Por isso, maior parte dos diagramas são apresentados em notação específica, desenvolvida pela *Microsoft*, baseada em UML e projetada para a descrição da modelagem de aplicações baseadas na *Plataforma .Net* [15]. Caso considere necessário conhecer melhor a notação utilizada, é possível consultar o Apêndice A, que apresenta as principais representações utilizadas nos diagramas deste trabalho e compara com a notação UML tradicional.

2.5 Eventos na Linguagem C#

Eventos são elementos da linguagem da C# bastante úteis para o nosso *framework* e que servem de alternativa ao padrão *Observer* [3], comumente utilizado em aplicações. Os eventos são mecanismos que podem ser utilizados por classes que notificam *clients* sobre o acontecimento de coisas interessantes. Os eventos fornecem uma maneira particularmente útil de os objetos enviarem um sinal de que o seu estado foi alterado [16].

Assim, por exemplo, quando uma classe X possui um evento, qualquer outro objeto da aplicação pode registrar algum de seus métodos neste evento. Desta maneira,

sempre que um objeto da classe X disparar o evento, todos os métodos de objetos registrados são automaticamente disparados. Eventos foram utilizados nas classes do *framework* que necessitam disparar notificações sobre as atualizações de estado.

2.6 Pacote Operations

O pacote Operations reúne o conjunto de entidades necessárias para representação do conceito de Operação. A figura ilustra as entidades deste pacote.

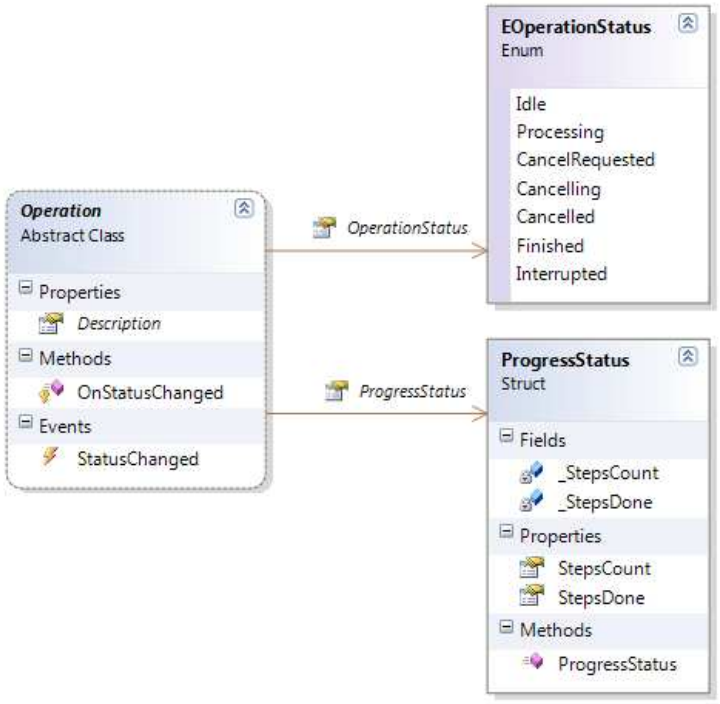


Figura 2.5 – Classes do pacote *Operations*

Conforme explicado, um serviço pode ser entendido como a execução de um conjunto de operações. Para a representação dessas operações, criamos três entidades no modelo: (1) Classe abstrata *Operation*; (2) Enumeração *EProgressStatus*; (3) Estrutura *ProgressStatus*. As seções a seguir descrevem o papel destas entidades no modelo.

2.6.1 Classe Operation

A classe abstrata *Operation* representa uma operação que faz parte de um serviço. Existirá pelo menos uma instância do tipo *Operation* (representando a operação principal do serviço) associada a um serviço monitorável. Através dela, o consumidor poderá consultar o percentual de progresso realizado e consultar o estado atual da operação.

Os membros que compõem a classe *Operation* estão descritos a seguir:

(1) Description (propriedade do tipo String) – uma descrição (texto) da operação.

(2) *ProgressStatus (propriedade do tipo ProgressStatus)* – permite consultar o estado corrente do progresso da execução da operação.

(3) OperationStatus (propriedade do tipo EOperationStatus) – permite consultar o estado corrente do processamento da operação.

(4) StatusChanged (evento) – evento disparado sempre que ocorre uma mudança no estado da operação (alterações na Descrição, ProgressStatus ou OperationStatus).

(5) OnStatusChanged (método privado) – método utilizado internamente para disparar o evento *StatusChanged* quando da mudança de estado da operação.

2.6.2 Estrutura ProgressStatus

A estrutura *ProgressStatus* representa o estado do progresso da execução de uma operação, com base na quantidade de passos executados e na quantidade total de passos necessários para a conclusão da operação.

O modelo não estabelece uma regra definida nem métrica específica para a determinação da quantidade de passos de um serviço. A estrutura *ProgressStatus* resolve estritamente o problema de possibilitar acompanhar o percentual de progresso de execução das operações.

A quantidade de passos executados por uma operação deve ser arbitrada pelo desenvolvedor do serviço, mediante análise dos algoritmos executados e do custo das suboperações processadas, de acordo com as particularidades de cada caso. Não faz parte do escopo deste trabalho estabelecer procedimentos e unidades específicas para o estabelecimento do custo das operações. A semântica utilizada para os valores utilizados como referência para a contagem de passos de cada operação deve ser definida conforme o programador de aplicação considerar mais adequado ou conveniente.

Os membros da estrutura *ProgressStatus* são:

(1) *StepsCount (propriedade do tipo Int32)* – valor inteiro que representa a quantidade total de passos necessários para a execução da operação.

(2) *StepsDone (propriedade do tipo Int32)* – valor inteiro que representa a quantidade de passos já realizados no decorrer do processamento da operação.

Assim, o percentual de progresso pode ser claramente obtido através da operação aritmética de divisão do valor inteiro em *StepsDone* pelo valor em *StepsCount*.

2.6.3 Enumeração EOperationStatus

A enumeração *EOperationStatus* representa todos os possíveis estados para uma operação da aplicação.

(1) Idle – Estado ocioso da operação, indicando que a sua execução ainda não foi iniciada.

(2) Processing – Estado de execução normal da operação, indicando que esta se encontra em andamento.

(3) CancelRequested – Estado que indica que o cancelamento da operação foi realizado pelo usuário.

(4) Cancelling – Estado que indica que a requisição de cancelamento foi percebida pelo controle de execução do serviço e este se encontra realizando os procedimentos necessários para o cancelamento.

(5) Cancelled – Estado que indica que a operação encontra-se ociosa e que a operação foi cancelada com sucesso.

(6) Finished – Estado que indica que a operação encontra-se ociosa porque a operação foi concluída normalmente.

(7) Interrupted – Estado que indica que a operação encontra-se ociosa porque foi interrompida devido à ocorrência de uma falha ou erro inesperado que impediu a continuidade da operação.

A figura a seguir ilustra o diagrama de estados de uma operação e explicita os eventos que condicionam as transições possíveis entre os estados, no decorrer da execução da operação.

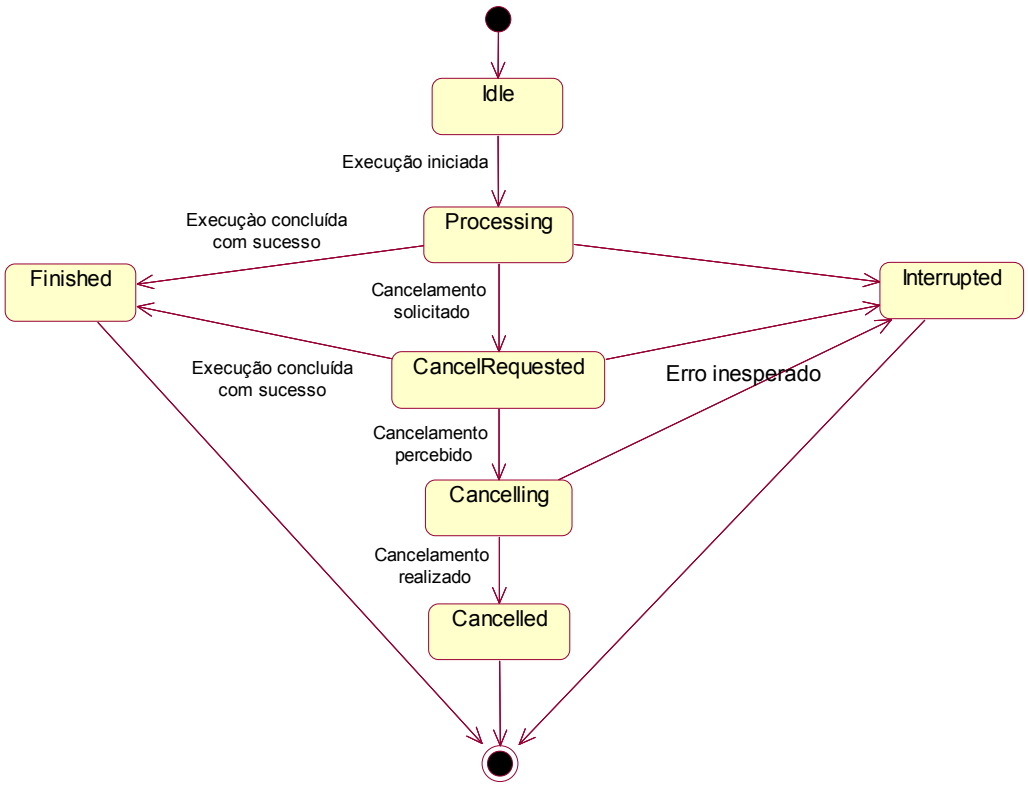


Figura 2.6 – Diagrama mudança de estados da operação

2.7 Pacote Notifications

O pacote Notifications reúne o conjunto de entidades necessárias à representação do conceito de Notificação. A Figura 2.7 ilustra as entidades deste pacote.

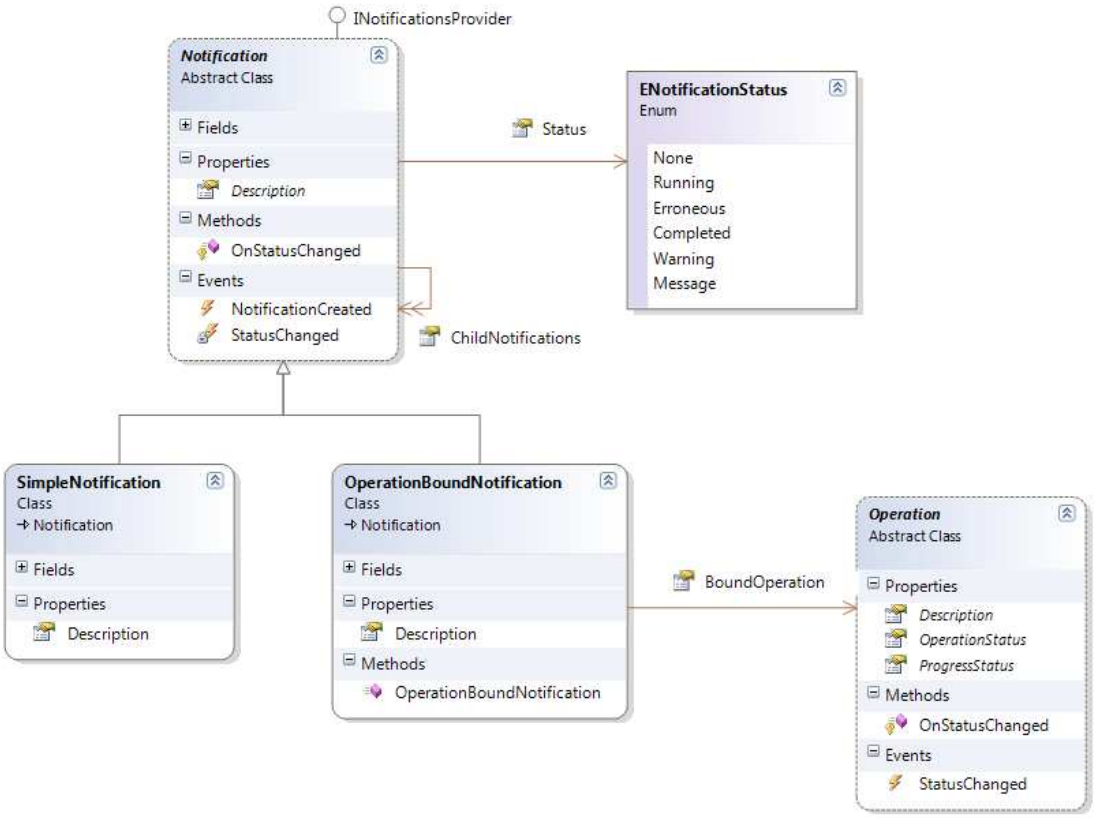


Figura 2.7 – Classes do pacote *Notifications*

As entidades criadas para a representação das notificações são: (1) Classe abstrata *Notification*; (2) Enumeração *ENotificationStatus*; (3) Classe *SimpleNotification*; (4) Classe *OperationBoundNotification*.

2.7.1 Classe Notification

A classe abstrata *Notification* representa uma notificação genérica emitida por um serviço em execução. Esta classe implementa a interface *INotificationsProvider*, o que significa que cada notificação também pode ser uma fonte de novas notificações (filhas), conforme explicado mais adiante.

Os principais membros da classe *Notification* são:

(1) *Description (propriedade do tipo String)* – uma descrição (texto) da operação.

(2) *Status (propriedade do tipo ENotificationStatus)* – permite consultar o tipo de notificação.

(3) *ChildNotifications (propriedade do tipo List<Notification>)* – permite consultar a coleção de notificações filhas, encadeadas, associadas à notificação.

(4) *NotificationCreated (evento)* – evento disparado sempre que uma nova notificação filha for criada.

(5) *StatusChanged (evento)* – evento disparado sempre que o estado da notificação for modificado.

2.7.2 Enumeração ENotificationStatus

A enumeração *ENotificationStatus* representa todos os tipos possíveis de notificação.

(1) *Running* – Indica que a notificação se refere a uma operação cuja execução está em andamento.

(2) *Erroneous* – Indica que a notificação representa um erro ou falha na execução do serviço.

(3) *Completed* – Indica que a notificação se refere a uma operação cuja execução foi concluída.

(4) *Warning* – Indica que a notificação é um alerta para o usuário.

(5) *Message* – Indica que a notificação é uma mensagem com informações específicas inerentes ao domínio do serviço.

2.7.3 Classe SimpleNotification

Representa uma notificação informativa, simples (mensagens com informações inerentes ao domínio, alertas, erros, indicadores, etc.).

(1) *Description (propriedade do tipo String)* – texto gerado pelo serviço emissor e que descreve a notificação.

2.7.4 Classe OperationBoundNotification

A classe *OperationBoundNotification* representa uma notificação associada a uma operação realizada pelo serviço.

(1) *Description (propriedade do tipo String)* – texto que descreve a notificação (gerado em função da descrição da operação associada).

(2) *BoundOperation (propriedade do tipo Operation)* – permite o acesso à Operação que está vinculada à notificação.

2.8 Pacote Interfaces

Uma vez definidas as classes básicas necessárias para a representação de todos os conceitos envolvidos, precisamos definir as interfaces que estabelecem o modo como o Consumidor dos serviços tem acesso às informações de monitoramento, através do pacote *Interfaces*: (1) interface *INotificationsProvider;* (2) interface *IServicesMonitor*; (3) interface *IMonitorableServicesProvider*.

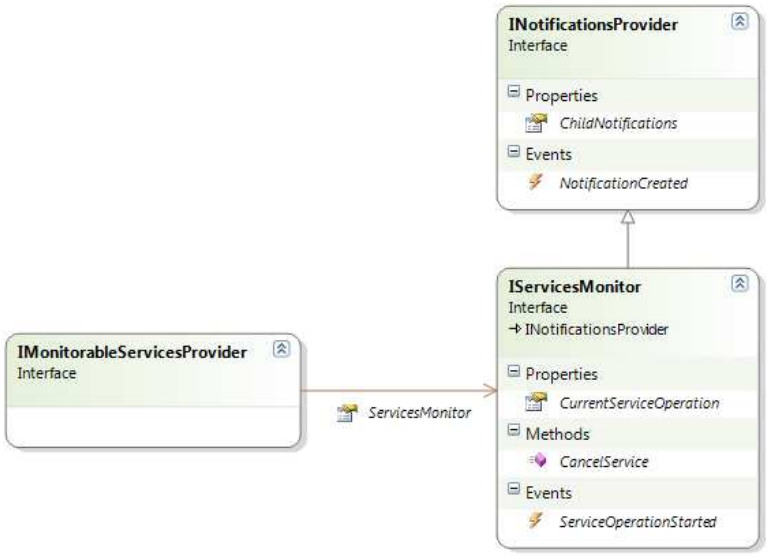


Figura 2.8 – Interfaces do monitoramento

2.8.1 Interface INotificationsProvider

A interface *INotificationsProvider* representa uma entidade passível de emissão de notificações. Através desta interface, o consumidor pode ter acesso ao conjunto de notificações geradas até o momento, além de ser notificado no momento exato em que novas notificações forem geradas. É importante observar que a classe *Notification*, definida anteriormente, também implementa esta interface, uma vez que cada notificação pode possuir notificações filhas, através da emissão de suas próprias notificações.

(1) *ChildNotifications (propriedade do tipo List<Notification>)* – permite o acesso à coleção de notificações emitidas pelo serviço desde o início de sua execução.

(2) NotificationCreated (evento) - evento disparado para indicar ao consumidor o fato de que uma nova notificação foi emitida.

2.8.2 Interface IServicesMonitor

A interface *IServicesMonitor* representa uma entidade que provê informações de monitoramento de serviços. Através dela, o consumidor pode ter acesso às informações da operação que se encontra em andamento, ser avisado do momento que uma nova operação for iniciada, cancelar a execução do serviço corrente além de ter acesso aos serviços de notificação definidos em *INotificationsProvider*.

(1) *CurrentServiceOperation (propriedade do tipo Operation)* – permite o acesso à operação corrente.

(2) *ServiceOperationStarterd (evento)* – evento disparado para indicar ao consumidor o fato de que a operação do serviço iniciou.

(3) *CancelService (método)* – permite que o consumidor do serviço requisite o cancelamento da operação, durante a sua execução.

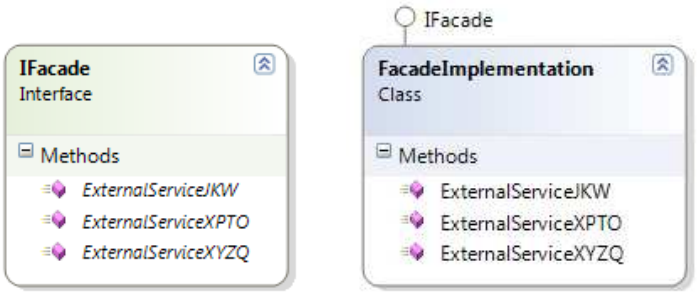
2.8.3 Interface IMonitorableServicesProvider

A interface *IMonitorableServicesProvider* representa uma entidade cujos serviços são passíveis de monitoramento e, portanto, disponibiliza para o consumidor uma entidade do tipo *IServicesMonitor* através da qual as informações de monitoramento poderão ser acessadas.

(1) *ServicesMonitor (propriedade do tipo IServicesMonitor)* – permite o acesso à entidade que coordena as informações de monitoramento dos serviços fornecidos.

2.9 IMPACTO NA ARQUITETURA DA APLICAÇÃO

Para entender melhor como a interface *IMonitorableServicesProvider* deve ser utilizada na prática, consideremos uma camada de serviços de uma aplicação, projetada de acordo com o padrão fachada. Se quiséssemos fazer com que os serviços desta camada da aplicação fossem passíveis de monitoramento, utilizando o nosso modelo, teríamos que fazer a fachada implementar *IMonitorableServicesProvider*. Assim, a classe que implementa a fachada passaria a conter uma propriedade *ServiceMonitor* através da qual o consumidor poderia acessar as informações de monitoramento, ilustrado pelas figuras a seguir.

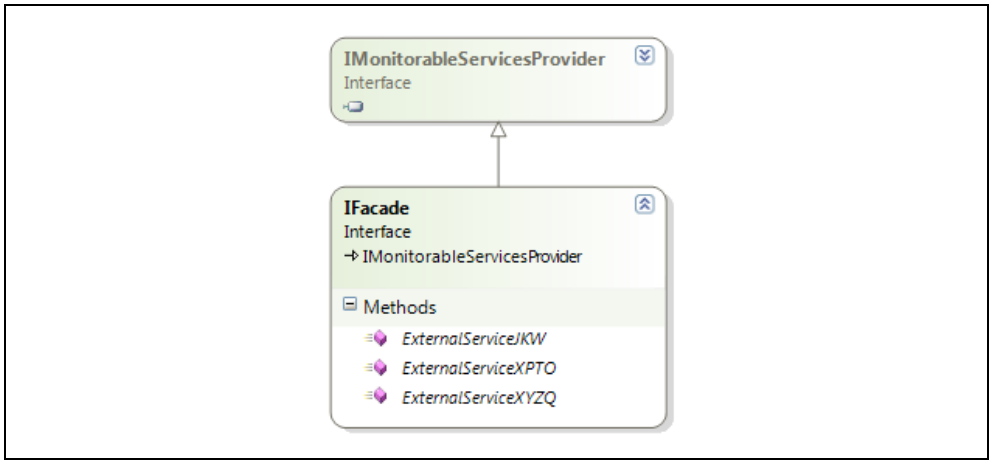


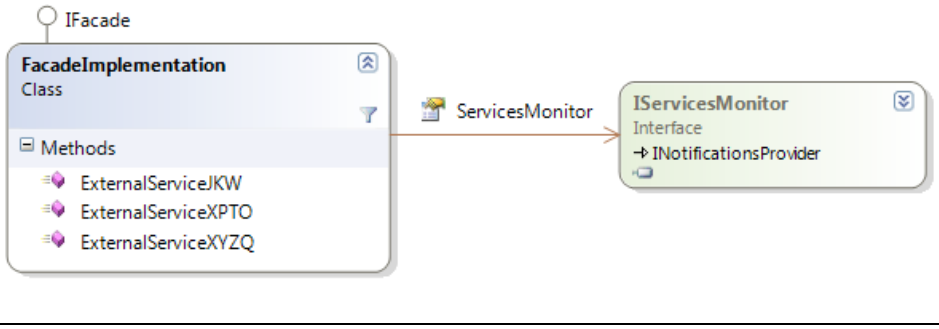

```
public interface IFacade
{
	void ExternalServiceXPTO();
	void ExternalServiceXYZQ();
	void ExternalServiceJKW();

}
class FacadeImplementation : IFacade
{
	public void ExternalServiceXPTO()
	{
		// Implementação do serviço XPTO.
	}
	public void ExternalServiceXYZQ()
	{
		// Implementação do serviço XYZQ.
	}
	public void ExternalServiceJKW()
	{
		// Implementação do serviço JKW.
	}

}
```

Figura 2.9 – Exemplo de fachada tradicional, não monitorável






```
public interface IFacade : IMonitorableServicesProvider
{
	void ExternalServiceXPTO();
	void ExternalServiceXYZQ();
	void ExternalServiceJKW();
}

class FacadeImplementation : IFacade
{
	public void ExternalServiceXPTO()
	{
		// Implementação do serviço XPTO.
	}
	public void ExternalServiceXYZQ()
	{
		// Implementação do serviço XYZQ.
	}
	public void ExternalServiceJKW()
	{
		// Implementação do serviço JKW.
	}

	private IServicesMonitor _Monitor;

	public Fundamentals.IServicesMonitor ServicesMonitor
	{
		get { return _Monitor; }
	}

}
```

Figura 2.10 – Interface da fachada adequada ao modelo de monitoramento

É importante observar que a implementação das interfaces definidas no *framework* acarreta pequeno impacto na implementação da Fachada. Assim, uma aplicação que utilize este padrão pode ser facilmente adaptada para utilizar o *framework* e contemplar o monitoramento dos serviços.

CAPÍTULO 3

Bibliotecas ServiceMon ProviderToolkit e ConsumerToolkit

No capítulo anterior, apresentamos um *framework* de monitoramento que define as classes e interfaces fundamentais para o modelo de monitoramento. Porém, na prática, se utilizarmos apenas o *framework*, o trabalho necessário para a criação de aplicações monitoráveis ainda é dispendioso. Ainda seria preciso implementar todos os mecanismos de criação, controle de encadeamento, atualização de estados, disparo de eventos e exibição para o usuário das operações e notificações decorrentes da execução de cada serviço da aplicação.

A criação do *framework* representa um passo fundamental, mas não suficiente para o atendimento completo dos objetivos deste trabalho: precisamos de componentes de software que tornem fácil e produtiva a criação de aplicações monitoráveis.

Vamos utilizar o *framework ServiceMon Fundamentals* como base para implementar bibliotecas específicas, voltadas para facilitar a implementação de consumidor e provedor de serviços da aplicação: *ServiceMon ProviderToolkit* e *ServiceMon ConsumerToolkit*, conforme ilustrado no diagrama abaixo.

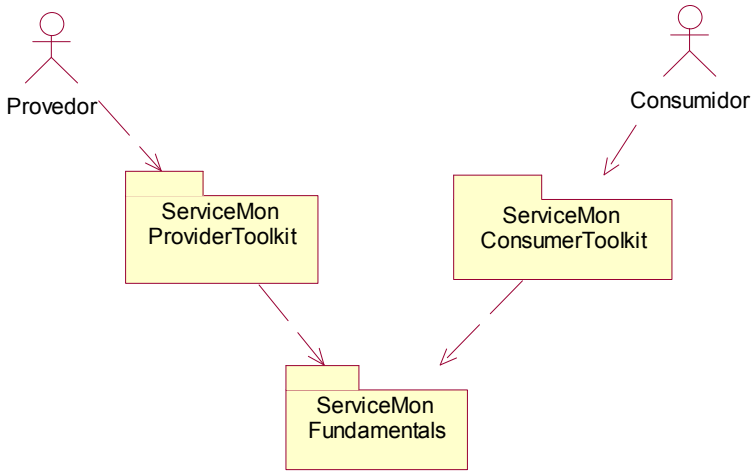


Figura 3.1 – Dependências entre bibliotecas e *framework*

A seguir, descrevemos detalhes de modelagem e implementação de cada uma destas bibliotecas e apresentamos conceitos peculiares da *Plataforma .Net* e da linguagem C# necessários para o entendimento das classes e interfaces as compõem.

3.1 Biblioteca ServiceMON ProviderToolkit

O objetivo da biblioteca *ServiceMon ProviderToolkit* é fornecer componentes que tornem fácil a implementação de serviços monitoráveis, através da disponibilização de mecanismos concretos de controle e manipulação das operações e notificações definidas em *ServiceMon Fundamentals* e estruturas que visem à simplificação da implementação das interfaces de monitoramento, como *IMonitorableServicesProvider*. A biblioteca deve encapsular toda a lógica de criação e encadeamento de operações e notificações, permitindo ao desenvolvedor do serviço um maior poder de abstração sobre o monitoramento das operações e minimizando o impacto sobre o código fonte da classe que implementa os serviços.

Utilizando o *framework*, a dificuldade inerente ao desenvolvimento de um serviço monitorável consiste da implementação da interface *IMonitorableServicesProvider* pela classe provedora dos serviços.

Uma classe que implemente *IMonitorableServicesProvider* precisa disponibilizar, para quem a consome, uma propriedade pública do tipo *IServicesMonitor* através da qual é possível ter acesso às informações de monitoramento.

A biblioteca *ProviderToolkit* oferece classes que implementam estas interfaces.

3.1.1 A interface *IDisposable* e a instrução *using*

Antes de iniciarmos a descrição das classes que compõem esta biblioteca, vamos apresentar conceitos importantes para o nosso trabalho: (1) a interface *IDisposable*, da Plataforma .Net, e (2) a instrução *using*, da linguagem C# [17].

A interface *IDisposable* é utilizada, na Plataforma .Net, para representar objetos que alocam recursos de memória e, após serem utilizados, devem ter seus recursos explicitamente liberados pelo programador. De forma mais genérica, podemos entender que, se um objeto implementa a interface *IDisposable*, significa que, após ser utilizado, o programador deve avisar que já terminou de utilizá-lo explicitamente, através de uma chamada ao método *Dispose()* [18]. A interface está representada na Figura 3.2.

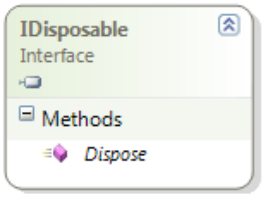


Figura 3.2 – Interface *IDisposable*

Ao criar uma instância de uma classe do tipo *IDisposable*, portanto, o programador deve lembrar de executar o método *Dispose()*. Assim, para uma classe XPTO qualquer, que implemente *IDisposable*, por exemplo, torna-se bastante comum o seguinte tipo de construção de linguagem:

```
XPTO myObject = new XPTO();
try
{
	// ...
	// Utiliza a instância myObject
	// ...
}
finally
{
	myObject.Dispose();
}
```

Figura 3.3 – Exemplo de código que utiliza classe *IDisposable*

No exemplo, o método *Dispose()* é executado em um bloco *finally* para garantir que a instância do objeto será descartada, mesmo que um erro inesperado ocorra durante o fluxo de execução e utilização do objeto.

Para tornar mais limpo o código fonte do programa e facilitar a vida do programador, a linguagem C# inclui uma estrutura nativa de linguagem, o *using*, para realizar esta tarefa. Assim, o mesmo código do exemplo passado poderia ser substituído por:

```
using(XPTO myObject = new XPTO())
{
	// ...
	// Utiliza a instância myObject
	// ...
}
```

Figura 3.4 – Exemplo utilizando instrução *using*

O código fonte exibido na figura anterior tem o mesmo efeito daquele que utiliza a estrutura *try-finally*. Nas seções seguintes, vamos explicar como tomamos proveito desta facilidade da Plataforma .Net e da linguagem C# na construção de operações.

3.1.2 A classe *ServicesMonitor*

A classe *ServicesMonitor* é o elemento central da biblioteca. Ela possui uma implementação completa para interface *IServicesMonitor* e, além disso, disponibiliza métodos para que o desenvolvedor dos serviços da classe possa controlar a criação das operações e notificações.

A idéia é que uma classe provedora de serviços – por exemplo, a implementação da fachada de uma camada qualquer da aplicação – crie uma instância da classe *ServicesMonitor* e utilize internamente seus métodos para criar as operações.

A figura abaixo exibe os membros desta classe.

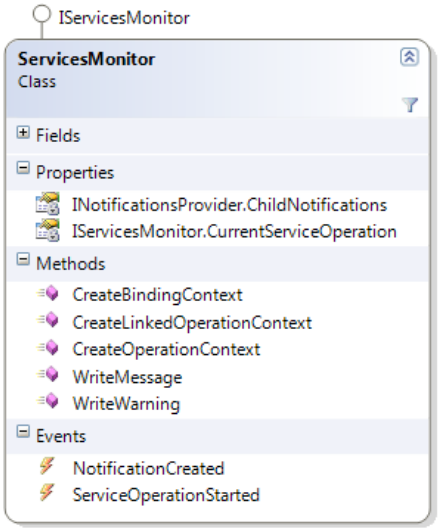


Figura 3.5 – Classe *ServicesMonitor*

Os membros decorrentes da implementação da interface *IServicesMonitor* e *INotificationsProvider* são:

(1) *ChildNotifications (propriedade do tipo List<Notification>)* – permite o acesso à lista de notificações criadas durante a execução do serviço, até o momento.

(2) *CurrentServiceOperation (propriedade do tipo Operation)* – permite o acesso à instância da classe *Operation* que representa a operação principal que representa o serviço em execução.

(3) *NotificationCreated (evento)* – Evento disparado sempre que uma nova notificação for criada, durante a execução do serviço.

(4) *ServiceOperationStarted (evento)* – Evento disparado sempre que um serviço for executado pela aplicação. Quando o evento é disparado, significa que a propriedade *CurrentServiceOperation* pode ser acessada para obter informações sobre o serviço recém iniciado.

Os membros (1) a (4) acima são implementados pela classe *ServicesMonitor* automaticamente, em função dos métodos de controle das operações abaixo:

(5) *CreateOperationContext* – método que deve ser utilizado pelo desenvolvedor da classe provedora de serviços para determinar a criação de um novo Contexto de Operação.

(6) *CreateLinkedOperationContext* – método que deve ser utilizado pelo desenvolvedor da classe provedora de serviços para criar uma operação vinculada a um serviço de uma outra classe provedora de serviços. A atualização do estado de execução e progresso da operação será realizada automaticamente em função das alterações no estado do serviço da outra classe.

(7) *CreateBindingContext* – método que deve ser utilizado pelo desenvolvedor da classe provedora de serviços para criar uma operação vinculada à execução de um outro método da mesma classe provedora de serviços. As operações criadas pelo outro método serão encadeadas automaticamente como suboperações da operação principal. A atualização do estado de execução e progresso da operação será a realizada automaticamente conforme a operação do método executado em cadeia é atualizada.

(8) *WriteMessage* – método que deve ser utilizado para enviar uma notificação de texto genérica. A notificação será associada automaticamente à operação corrente na árvore de execução.

(9) *WriteWarning* – método que deve ser utilizado para enviar uma notificação de alerta de texto. A notificação será associada automaticamente à operação corrente na árvore de execução.

3.1.3 Contextos de Operação

Quando discorremos sobre o conceito de *Operação,* no contexto de monitoramento deste trabalho, vimos que podemos dividir a execução de um serviço em operações e suboperações que realizam uma parte do trabalho total realizado do serviço.

Assim, os trechos do código fonte da implementação de um serviço pode ser agrupado de modo a determinar quais são as operações executadas.

Para simplificar e realizar automaticamente o encadeamento das operações executadas, esta biblioteca introduz a noção de Contexto de Operação. Associada à cada operação criada, existirá um contexto de execução – um bloco de código que conterá o trecho de algoritmo da operação.

Para implementar um serviço monitorável utilizando a biblioteca, temos que seguir as seguintes considerações:

(1) Todo serviço deve criar um contexto de operação principal que representará o estado completo de execução do serviço.
(2) Associado a um contexto de operação, existem dois valores inteiros que representam o custo total e o custo executado pela operação até o momento.
(3) Outros contextos de operação podem ser criados dentro de um contexto de operação existente. Existem três possibilidades:
    a. Contexto de operação criado para agrupar um conjunto de instruções do código fonte;
    b. Contexto de operação criado para vincular a chamada de um serviço de outra classe monitorável;
    c. Contexto de operação criado para vincular a chamada de um outro método da mesma classe.

A seguir, vamos explorar as entidades criadas para representar os diferentes contextos de execução e apresentaremos exemplos de utilização de cada uma delas no código fonte da aplicação.

A Figura 3.6 ilustra essas entidades.

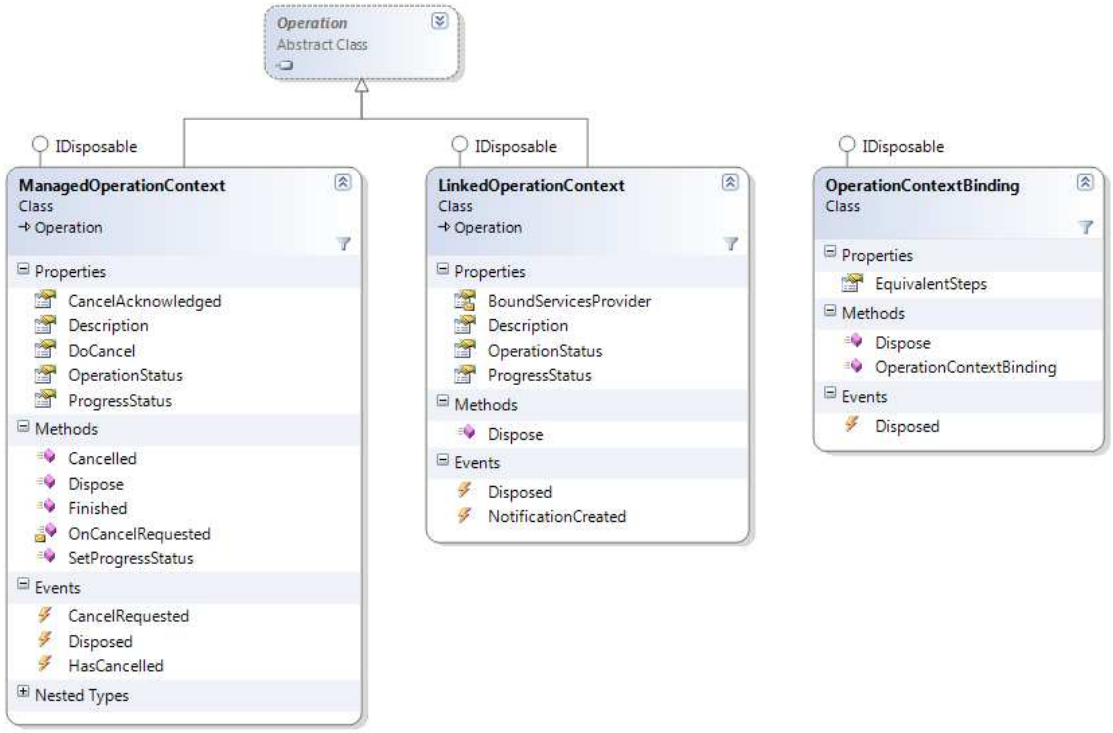


Figura 3.6 – Classes para representação de contextos de operação

As classes que representam os contextos de operação implementam a interface *IDisposable* para permitir o controle sobre o início e o término de execução dos contextos. Assim, a chamada do construtor da classe representa o início da operação e a chamada do método *Dispose(),* obrigatório, em função do uso da construção *using*, representa o fim da operação. Quando o método *Dispose()* é executado sem que estado de execução da operação tenha sido marcado como *Finished*, temos a indicação de que um erro inesperado ocorreu durante a execução da operação e impossibilitou a sua conclusão com sucesso.

3.1.3.1 ManagedOperationContext

Esta classe representa o contexto de operação mais usual, aquele que agrupa um conjunto de instruções no código fonte do serviço e cuja mudança de estado do progresso deve ser gerenciada e controlada explicitamente durante o fluxo de execução.

A classe *ManagedOperationContext* herda da classe abstrata *Operation* e implementa a lógica de disparo de eventos e controle de mudança de estados.

Os principais membros desta classe são:

(1) *CancelAcknowledged (propriedade booleana)* – Indica se o cancelamento da operação foi percebido e está sendo processado ou foi processado pelo provedor de serviços.

(2) *Description (propriedade String)* – Contém a descrição da operação;

(3) *DoCancel (propriedade booleana)* – Indica se o cancelamento da operação foi solicitado pelo consumidor dos serviços. O código fonte do provedor de serviços deve verificar eventualmente o estado desta propriedade a fim de perceber a solicitação de cancelamento e iniciar o procedimento de cancelamento, se aplicável. Quando existe a solicitação de cancelamento e o provedor verifica através desta propriedade, fica preestabelecido pela biblioteca que as providências de cancelamento serão tomadas e a propriedade *CancelAcknowledged* muda para *true*, indicando que o cancelamento foi percebido pelo provedor.

(4) *OperationStatus* – Armazena o estado atual da operação. O estado da operação é controlado automaticamente pela biblioteca em função do início e término do contexto de execução. O bloco de código contemplado pelo contexto de execução também detecta o lançamento de exceções inesperadas que alteram o estado de execução da operação para indicar a ocorrência de erro.

(5) *ProgressStatus* – Armazena o estado de progresso da operação. Esta propriedade é alterada em função das chamadas realizadas ao método *SetProgressStatus()*.

(6) *Cancelled()* – Método executado pelo provedor para indicar que o procedimento de cancelamento da operação foi concluído. A chamada deste método desencadeia o disparo do evento *StatusChanged* e *HasCancelled* da operação.

(7) *Dispose()* – Método decorrente da implementação da interface *IDisposable* que representa o término da execução da operação.

(8) *Finished()* – Método executado pelo provedor para indicar que a operação foi concluída com sucesso.

(9) *SetProgressStatus()* – Método executado pelo provedor para atualizar o estado do progresso da operação à medida que o serviço é processado. A chamada deste método desencadeia o disparo do evento *StatusChanged* da operação.

(10) *CancelRequested* – Evento disparado quando o consumidor solicita o cancelamento da execução da operação. O provedor de serviços deve registrar-se para recebimento deste evento com o intuito de implementar código responsável pelo cancelamento.

(11) *HasCancelled* – Evento disparado quando o procedimento de cancelamento da operação é realizado.

O código fonte a seguir ilustra a criação de uma classe provedora de serviços monitoráveis que utiliza uma instância da classe *ServicesMonitor* para implementar um serviço logicamente divido em três suboperações. Para isso, no exemplo dado, criamos um contexto de operação principal, com custo 100, para representar o serviço como um todo, e mais três suboperações com custos 30, 40, 30, nessa ordem,

```
public void ServiceExample()
{

      using (ManagedOperationContext mainOperation =
            _Monitor.CreateOperationContext("Serviço", 100))

      {
            using (ManagedOperationContext op =
                  _Monitor.CreateOperationContext("Operação 1", 30))
            {
                  // ...
                  op.SetProgressStatus(10);
                  // ...
                  op.Finished();
            }

            using (ManagedOperationContext op =
                  _Monitor.CreateOperationContext("Operação 2", 40))
            {
                  // ...
                  op.SetProgressStatus(20);
                  // ...
                  op.SetProgressStatus(30);
                  //..
                  op.Finished();
            }
```

```
            using (ManagedOperationContext op =
                  _Monitor.CreateOperationContext("Operação 3", 30))
            {
                  // ...
                  op.Finished();
            }

            mainOperation.Finished();
      }
}
```

Figura 3.7 – Exemplo de serviço com operações

O contexto de execução de uma operação fica determinado pelo bloco de instruções criado pela estrutura *using*. No exemplo dado, as três suboperações ficam vinculadas à operação principal do serviço pelo fato de terem sido criadas dentro do seu bloco de código *using*. O vínculo existente entre as operações condiciona a atualização do estado do progresso da operação principal e, portanto, quando o método *UpdateProgressStatus()* é executado para as suboperações, a operação principal também tem o seu estado atualizado.

O exemplo apresentado faz uso de um atributo *_Monitor*, do tipo *ServicesMonitor*, pertencente à classe provedora do serviço. Para controlar a criação e controle da vinculação e encadeamento entre operações, a classe *ServicesMonitor* utiliza um mecanismo de empilhamento das operações criadas.

3.1.3.2 OperationContextBinding

Considere que precisamos criar um novo serviço para a classe provedora. Considere também que este novo serviço tem custo 60, sendo que uma das suboperações deste novo serviço consiste em executar o serviço *ServiceExample* do exemplo anterior e isto representa custo 20 no contexto de execução do novo serviço.

A biblioteca provê uma forma de vincular e atualizar automaticamente o custo da suboperação, no contexto de execução do novo serviço, ao custo utilizado na implementação do serviço que está sendo reusado.

Por exemplo, quando a primeira suboperação fosse concluída na execução *ServiceExample*, representando 30 passos concluídos de um total de 100, teríamos o progresso do novo serviço atualizado proporcionalmente, automaticamente, para $\left(\frac{30}{100}\right) \times 20 = 6$.

Para representar e controlar este vínculo, a biblioteca oferece a classe *OperationContextBinding* que pode ser criada a partir do *ServicesMonitor*. Os principais membros dessa classe são:

(1) *EquivalentSteps* – Propriedade que indica o custo da operação no contexto do serviço principal em execução, que reusa a outra operação.

(2) *Dispose()* – Método que decorrente da implementação de *IDisposable*. A chamada deste método indica que o contexto de vinculação da operação foi concluído.

```
public void OtherServiceExample()
{
      using (ManagedOperationContext mainOperation =
            _Monitor.CreateOperationContext("Outro Serviço", 60))
      {
            using (ManagedOperationContext op =
                  _Monitor.CreateOperationContext("Operação 1", 40))
            {
                  // ...
                  op.Finished();
            }

            using (OperationContextBinding bind =
                  _Monitor.CreateBindingContext(20))
            {
                  ServiceExample();
            }

            mainOperation.Finished();
      }
}
```

Figura 3.8 – Exemplo de chamada a outro serviço da mesma classe

Observe que a chamada do método *ServiceExample()* acontece dentro do contexto de vinculação de execução. Dessa forma, o *ServicesMonitor* fará com que operação principal do serviço *ServiceExemple()* seja uma suboperação de custo 20 no contexto de execução do serviço *OtherServiceExample()*.

3.1.3.3 LinkedOperationContext

Considere agora que estamos desenvolvendo uma aplicação com arquitetura em camadas. Naturalmente, a implementação dos serviços de uma camada intermediária requer a execução de serviços da camada imediatamente inferior. Suponha que todas as camadas da aplicação sejam monitoráveis. Assim, vamos considerar que a fachada da

camada inferior implementa *IMonitorableServicesProvider* e estamos desenvolvendo um serviço de custo 100 onde uma das suboperações deste serviço, com custo 50, consiste em solicitar a execução de um serviço da fachada da camada inferior.

A classe *LinkedOperationContext* representa um contexto de operação vinculado à execução de um serviço em uma outra entidade que implemente a interface *IMonitorableServicesProvider*.

A figura a seguir ilustra um exemplo de uso desta classe.

```
public void ServiceExample()
{
      ILayerFacade nextLayer = GetNextLayer();

      using (ManagedOperationContext mainOperation =
            _Monitor.CreateOperationContext("Outro Serviço", 100))
      {

            // ...

            using (LinkedOperationContext op =
            _Monitor.CreateLinkedOperationContext(nextLayer, 50))
            {
                  nextLayer.MyService();
            }

            // ...

            mainOperation.Finished();
      }
}
```

Figura 3.9 – Exemplo de chamada a outro serviço monitorável em outra entidade

A variável *nextLayer* contém a referência para uma instância da fachada da camada imediatamente inferior. A chamada do método *nextLayer.MyService()* acontece no contexto da operação vinculada. Assim, a operação gerada pela camada inferior é capturada pelo *ServicesMonitor* e vinculada como uma suboperação de custo 50 no contexto do serviço *ServiceExample()*.

Observe que a criação do contexto de vinculação acontece através da chamada do método *_Monitor.CreateLinkedOperationContext(nextLayer, 50)*. Observe que o *_Monitor* recebe uma referência para *nextLayer*, que implementa a interface *IMonitorableServicesProvider* e que será inspecionado pelo *_Monitor* para detectar a execução de operações e vinculá-las automaticamente ao contexto.

Os principais membros da classe *LinkedOperationContext* são:

(1) *BoundServicesProvider* – contém uma referência para a entidade que implementa *IMonitorableServicesProvider* e cujas operações serão inspecionadas e vinculadas ao contexto de execução do serviço principal.

(2) *Description* – Contém a descrição da operação, vinculada à descrição da operação gerada pelo *BoundServicesProvider*.

(3) *OperationStatus* - Contém o *status* da operação, vinculado às mudanças de *status* ocorridas na operação gerada pelo *BoundServicesProvider*.

(4) *ProgressStatus* – Contém o estado do progresso de execução da operação no contexto da camada superior. O valor é atualizado e calculado automaticamente, com custo proporcionalizado, em função das alterações do progresso da operação executada em *BoundServicesProvider*.

(5) *Dispose()* – Método decorrente da implementação da interface *IDisposable*. Este método é executado para indicar o término do contexto de vinculação de operação executada na camada inferior.

3.1.3.4 Operações recursivas

Como vimos, toda operação possui um custo associado, dentro do contexto do serviço em que ela está inserida. É importante perceber que a lógica utilizada para construção dos contextos de operação permite a abstração da natureza e complexidade das instruções executadas dentro do escopo de uma operação. Assim, por mais complexo que for o encadeamento de operações, o programador poderá abstrair e ter a certeza de que, ao terminar a execução de um contexto de operação, o seu custo terá sido atualizado corretamente em função da atualização das operações que ocorrerem em cadeia.

Esta facilidade de abstração é particularmente útil para serviços onde ocorre recursão. Quando o mesmo serviço é executado várias vezes em uma cadeia de execução, o contexto de operação de cada serviço é isolado e a classe *ServicesMonitor* vincula as atualizações de progresso realizadas ao contexto de operação imediatamente superior.

Por exemplo, se tivermos um serviço X que possui custo 100 e realiza uma chamada recursiva de custo 40, vamos analisar o que acontece quando ele é processado.

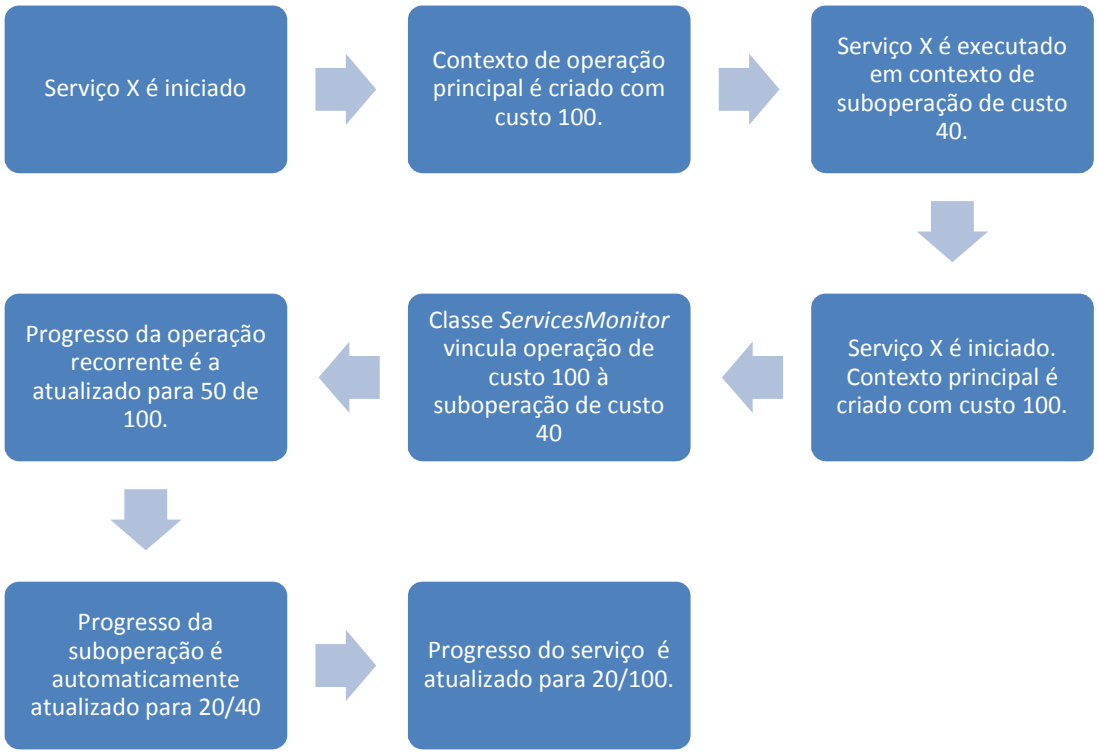


Figura 3.10 – Exemplo de criação de operação recursiva

O controle sobre o encadeamento de operações recursivas é realizado pela classe *ServicesMonitor*. Quando o método é executado pela segunda vez e solicita a criação de um novo contexto de operação, a classe identifica a existência de uma operação não concluída e associa a execução recursiva como uma suboperação do contexto de operação principal. É utilizada uma pilha de objetos *Operation* para controlar a cadeia de execução. O capítulo 4 apresenta o funcionamento de um serviço recursivo em uma aplicação real.

3.1.4 Notificações

Além de automatizar e encapsular os processos de geração e controle de estado das operações, a biblioteca contempla mecanismos que simplificam a geração e encadeamento de notificações.

Conforme vimos no *framework*, existem dois tipos de notificação que precisam ser consideradas: as notificações associadas à execução de uma operação e aquelas geradas arbitrariamente pelo serviço, com informações gerais sobre sua execução.

A Figura 3.11 ilustra as classes da biblioteca utilizadas para representar as notificações.

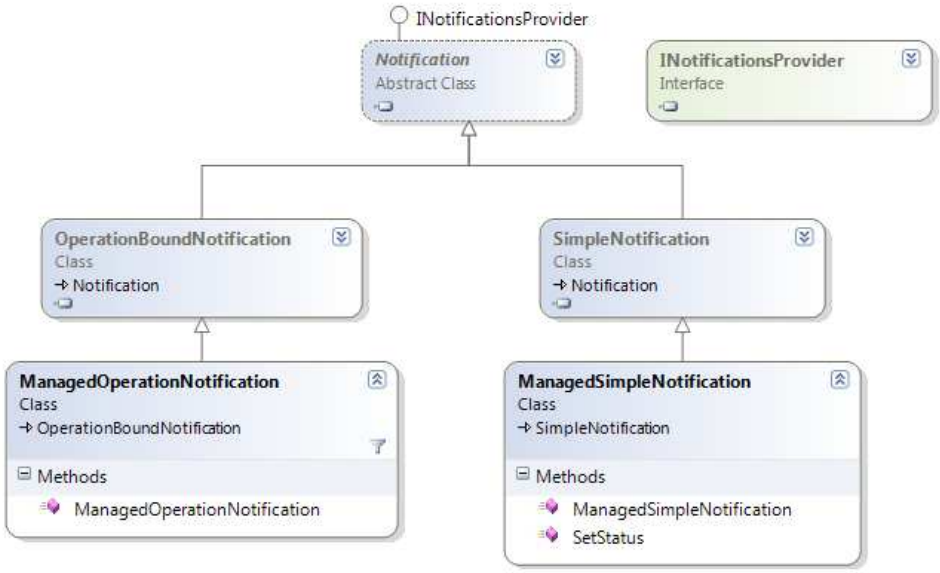


Figura 3.11 – Diagrama de classes que representam notificações

3.1.4.1 ManagedOperationNotification

Quando iniciamos a execução de uma operação, através da criação de contextos de operação, a biblioteca adiciona automaticamente uma notificação referente à execução do serviço, na árvore de notificações. Outras notificações que sejam criadas dentro do mesmo contexto são automaticamente encadeadas e vinculadas como filhas da notificação associada ao contexto corrente. Dessa forma, o encadeamento de notificações pode ser facilmente abstraído do desenvolvedor, pois está implícito na estrutura de criação de contextos de notificação no código fonte.

Ao utilizar a biblioteca, o desenvolvedor pode abstrair completamente a geração das notificações associadas aos contextos de operação, já que o processo é completamente automatizado pelo *ServicesMonitor* e o estado da notificação é atualizado em função das alterações no estado da operação.

A classe *ManagedOperationNotification* representa uma notificação associada a uma operação. Esta classe não apresenta membros públicos porque o desenvolvedor do provedor de serviços não precisa manipular a criação deste tipo de notificação. Este processo é encapsulado pelo *ServicesMonitor*.

3.1.4.2 ManagedSimpleNotification

Além das notificações vinculadas à execução de operações, a biblioteca facilita a criação de notificações genéricas, que podem ser alertas, mensagens de erro ou informações específicas, inerentes ao domínio do serviço executado.

A classe *ManagedSimpleNotification* representa este tipo de notificação. O único membro público desta classe é o método *SetStatus()* que permite ao usuário determinar o *status* da notificação (erro, alerta, mensagem, etc.)

A classe *ServicesMonitor* contém dois métodos que permitem criar notificações do tipo *ManagedSimpleNotification*: *WriteWarning* e *WriteMessage*. A figura abaixo exemplifica a geração de um alerta.

```
using (ManagedOperationContext mainOperation =
       _Monitor.CreateOperationContext("Serviço", 100))
{
       // ...
       _Monitor.WriteWarning("Mensagem de alerta");
       // ...
}
```

Figura 3.12 – Exemplo serviço gerando notificações de alerta

3.1.5 Cancelamento

Para que um serviço dê suporte a cancelamento, é preciso que o desenvolvedor insira no algoritmo código que verifique se o cancelamento foi solicitado pelo usuário. A classe *ManagedOperationContext* oferece suporte a esta verificação. Para que o cancelamento ocorra, os seguintes passos precisam ocorrer:

(1) Consumidor de serviços solicita cancelamento através do método *Cancel()* da classe *ServicesMonitor*;

(2) Em meio à execução, provedor de serviços verifica se há uma solicitação de cancelamento pendente. Para isso, utiliza a propriedade *DoCancel* da operação;

(3) Uma vez detectado, provedor altera o estado de operação para *Cancelling*, indicando que a solicitação foi detectada e o cancelamento está em operação. Esta mudança de estado é realizada automaticamente pela classe *ManagedOperationContext*, na medida em que a propriedade *DoCancel* é acessada;

(4) Provedor de serviços realiza os procedimentos de cancelamento;

(5) Provedor de serviços finaliza a operação executando o método *Cancelled()*, indicando que o cancelamento foi realizado com sucesso;

A figura abaixo ilustra a estrutura de um código fonte típico de um serviço passível de cancelamento.

```
using (ManagedOperationContext mainOperation =
      _Monitor.CreateOperationContext("Serviço", 100))
{
      // ... (instruções iniciais do serviço) ...

      // PRIMEIRO PONTO DE CANCELAMENTO
      if (mainOperation.DoCancel)
      {
            // ... Inicia procedimento de cancelamento da operação
      }
      else
      {
            // ... continua executando instruções ...
      }

      // SEGUNDO PONTO DE CANCELAMENTO
      if (mainOperation.DoCancel)
      {
            // ... Inicia procedimento de cancelamento da operação
      }
      else
      {
            // ... continua executando instruções ...
      }

      if (mainOperation.CancelAcknowledged)
            mainOperation.Cancelled();
      else
            mainOperation.Finished();
}
```

Figura 3.13 – Estrutura de um serviço passível de cancelamento

3.2 Biblioteca ServiceMON.ConsumerToolkit

O objetivo desta biblioteca é fornecer componentes gráficos que tornem fácil a exibição das informações de monitoramento para o usuário. Os componentes visam à exibição de uma barra de progresso e de uma árvore de notificações à medida que o serviço é executado.

Os componentes devem ser capazes de escutar os eventos e utilizar as informações presentes nas interfaces *IMonitorableServicesProvider* e *IServicesMonitor* para exibir na tela o estado da execução do serviço.

Os componentes da biblioteca estão ilustrados no diagrama abaixo.

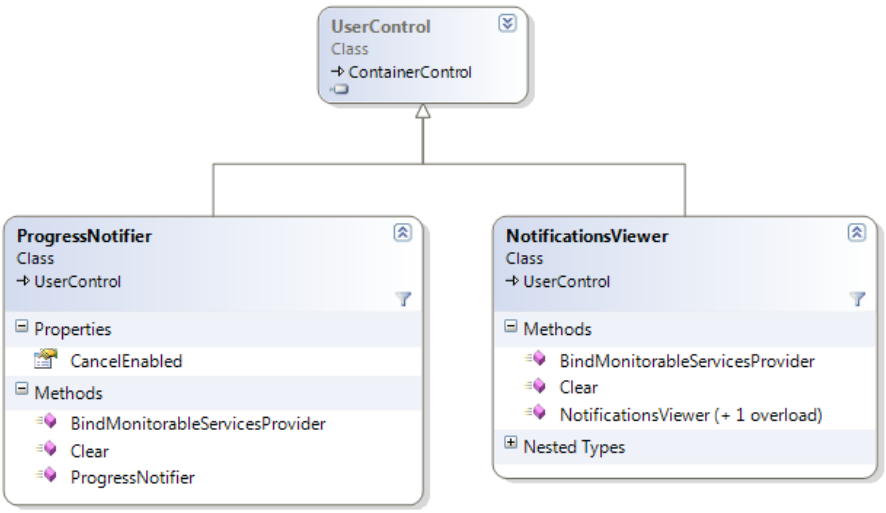


Figura 3.14 – Componentes da biblioteca *ConsumerToolkit*

3.2.1 Componente *ProgressNotifier*

O componente *ProgressNotifier* pode ser facilmente adicionado a uma interface gráfica de um sistema e vinculado a um objeto que implemente a interface *IMonitorableServicesProvider* para exibir na tela uma barra de progresso com o percentual executado pelo serviço, a descrição das operações que se encontram em execução e permitir que o usuário possa cancelar a execução do serviço. A classe herda de *UserControl* – classe da Plataforma .Net da qual devem herdar os componentes de interface com o usuário.

A figura abaixo ilustra o formato de exibição da informação de progresso pelo componente.

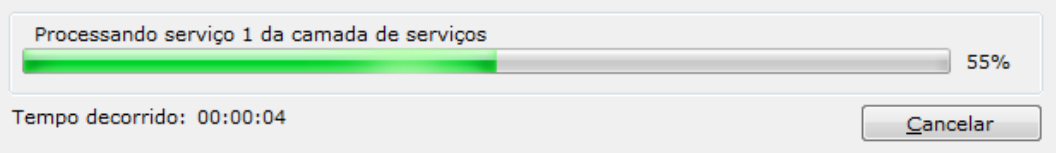


Figura 3.15 – Componente *ProgressNotifier*

Os métodos públicos do componente são:

(1) *BindMonitorableServicesProvider(IMonitorableServicesProvider provider)* – Método que permite a vinculação do controle à entidade provedora de

serviços monitoráveis (que, necessariamente, implementa *IMonitorableServicesProvider)*;

(2) *Clear()* – Restaura o componente para o seu estado inicial, em que a barra de progresso se encontra em 0% e nenhuma operação está sendo executada.

3.2.2 Componente *NotificationsViewer*

O componente *NotificationsViewer* consiste de uma árvore de exibição das notificações geradas pela execução de um serviço. Associado a cada nó da árvore, a depender do *status* da notificação, cores e ícones diferentes são utilizados. Da mesma maneira que o componente *ProgressNotifier*, este componente deve ser vinculado à entidade provedora de serviços que implementa a interface *IMonitorableServicesProvider* através do método *BindMonitorableServicesProvider.*

A figura abaixo ilustra um protótipo de exibição da árvore de notificações geradas por um serviço qualquer de uma aplicação.

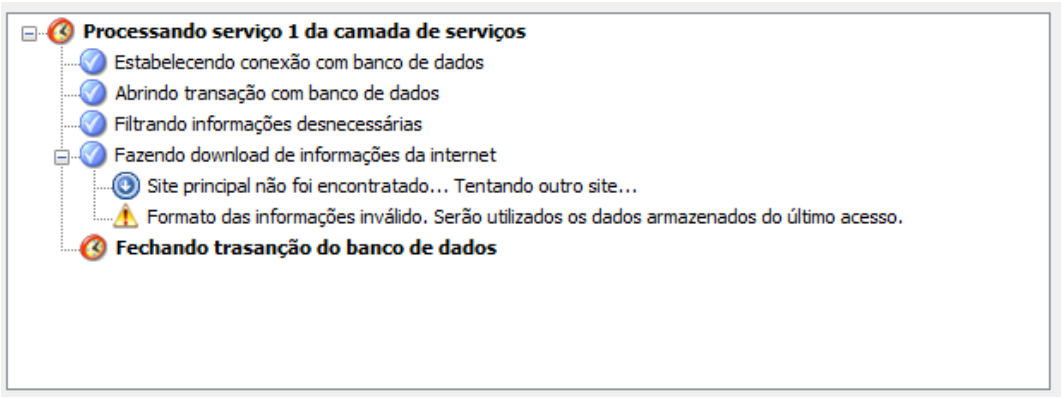


Figura 3.16 – Componente *NotificationsViewer*

CAPÍTULO 4

APLICAÇÃO DAS BIBLIOTECAS

Nos capítulos anteriores, detalhamos a criação de um *framework* e bibliotecas que acreditamos simplificar o desenvolvimento de sistemas com serviços monitoráveis. Neste capítulo, vamos validar os resultados obtidos e verificar o funcionamento das bibliotecas na prática, através da implementação de uma aplicação típica, com arquitetura em camadas e serviços fictícios, que contemple a utilização de suboperações, chamadas de serviços encadeados, chamadas recursivas, suporte a cancelamento, atualização de progresso dinâmica, e todos os casos comuns uma aplicação comercial típica.

4.1 VISÃO GERAL

Criamos uma aplicação com três funcionalidades fictícias disponíveis para o usuário. Cada uma delas contempla situações específicas e serão explicadas nas seções seguintes.

O intuito da aplicação é verificar o funcionamento das bibliotecas. Os serviços que elas apresentam não servem a nenhum propósito real. Foram inseridas instruções temporizadoras no corpo dos métodos, para simular a execução de instruções e algoritmos custosos cuja resposta não é imediata.

Para verificar que as funcionalidades de integração entre diferentes entidades monitoráveis e vinculações de operações funcionam, a aplicação foi organizada em três camadas: (1) interface com o usuário, (2) serviços da aplicação e (3) dados. Assim, métodos da camada de serviços fazem chamadas a métodos da camada dados, como uma suboperação da operação principal.

A Figura 4.1 mostra as entidades envolvidas nas camadas de serviços e dados.

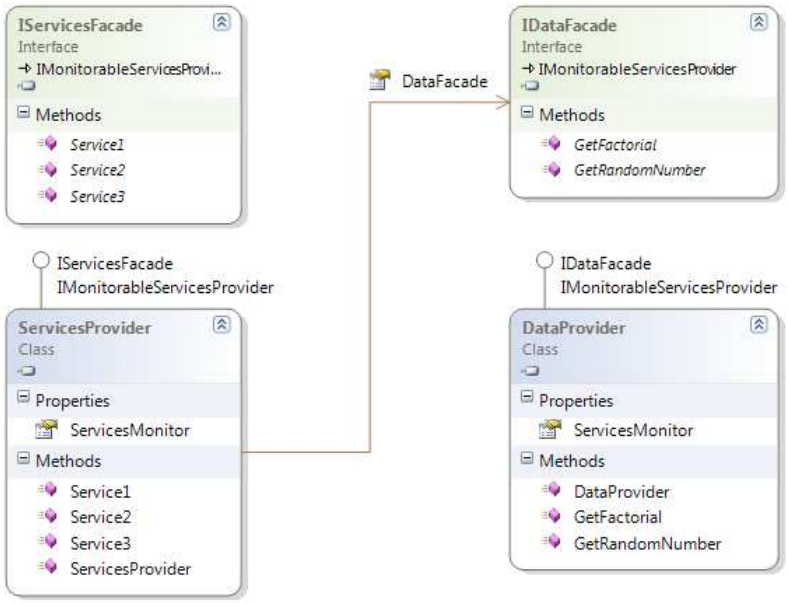


Figura 4.1 – Entidades das camadas de serviços e dados

O modelo utilizado segue o padrão *Facade* para ambas as camadas: temos uma interface de fachada e uma classe com a implementação. Para torná-las monitoráveis, implementamos em cada fachada a interface *IMonitorableServicesProvider* e utilizamos, na implementação, uma instância da classe *ServicesMonitor*, da biblioteca *ProviderToolkit*. A classe *ServicesProvider*, da camada de serviços, possui uma referência para a interface da camada de dados *IDataFacade.*

A interface gráfica faz uso dos componentes da biblioteca *ConsumerToolkit* e permite ao usuário inicializar um dos três serviços suportados. A figura a seguir exibe o protótipo desta interface.

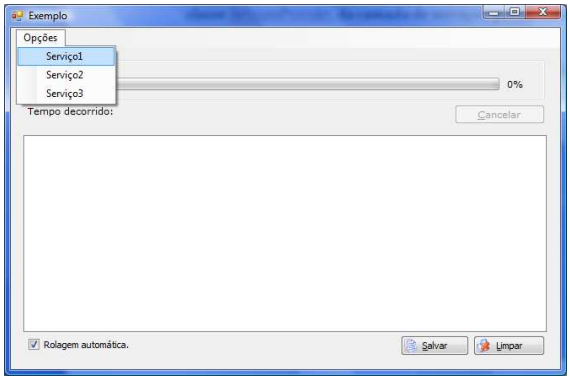


Figura 4.2 – Interface gráfica da aplicação

4.2 Classe *ServicesProvider*

A classe *ServicesProvider* implementa os três serviços de exemplo definidos na interface *IServicesFacade*. A seguir explicaremos as peculiaridades e propósitos de cada um desses serviços.

4.2.1 Método *Service1*

Este método representa um serviço simples da aplicação. O método consiste de algumas suboperações que dividem o algoritmo em etapas de processamento.

Uma das suboperações emite notificações de alerta em um *loop*, conforme código fonte abaixo.

```
_Monitor.WriteMessage("Logo abaixo, serão gerados 5 alertas dinamicamente");
for (int i = 1; i <= 5 && !mainOperation.DoCancel; i++)
{
        Thread.Sleep(200);
        _Monitor.WriteWarning("Mensagem de alerta número " + i.ToString());
}
```

Figura 4.3 – *Loop* gerando notificações

Note que a condição `i <= 5 && !mainOperation.DoCancel` de execução do *loop* leva em consideração a possibilidade de cancelamento por parte do usuário. Quando uma solicitação de cancelamento for realizada pelo usuário, `mainOperation.DoCancel` retornará *true* e o *loop* será terminado.

A instrução `Thread.Sleep(200)` faz com que o processador espere 200 milissegundos antes de continuar. A instrução foi utilizada para simular a execução de algum processamento/algoritmo custoso dentro do serviço.

Ainda no serviço 1, temos um exemplo de suboperação cujo progresso é atualizado dinamicamente em função de um contador do *loop*.

```
if (!mainOperation.DoCancel)
        using (ManagedOperationContext op = _Monitor.CreateOperationContext("Filtrando
                informações desnecessárias", 30))
        {
                for (int i = 1; i <= 15; i++)
                {
                        Thread.Sleep(100);
                        op.SetProgressStatus(30 * i / 15);
                }

                op.Finished();
        }
```

Figura 4.4 – *Loop* com atualização dinâmica do progresso

A linha `op.SetProgressStatus(30 * i / 15)` atualiza o progresso da operação em função da variável incrementada no *loop*.

4.2.2 Método *Service2*

Neste método, exemplificamos a criação de suboperações vinculadas à execução de outros serviços.

A primeira suboperação deste método executa o método *Service1* três vezes, dentro de um *loop*.

```
for (int i = 1; i <= 3 && !mainOperation.DoCancel; i++)
{
        _Monitor.WriteMessage(i.ToString() + "a. execução de serviço em contexto");
        using (OperationContextBinding binding = _Monitor.CreateBindingContext(200))
                Service1();
}
```

Figura 4.5 – Execução de serviços encadeados na mesma classe

Note que, no método *Service2*, a métrica de custos utilizada nas operações é diferente daquela utilizada em *Service1*. Assim, no contexto da execução de *Service2*, o custo de execução de *Service1* é de 200 passos. À medida que o *Service1* é executado, a biblioteca se responsabiliza por proporcionalizar os valores e atualizar o estado de progresso da execução de *Service2*. A Figura 4.6 mostra como os serviços são encadeados na árvore de execução.

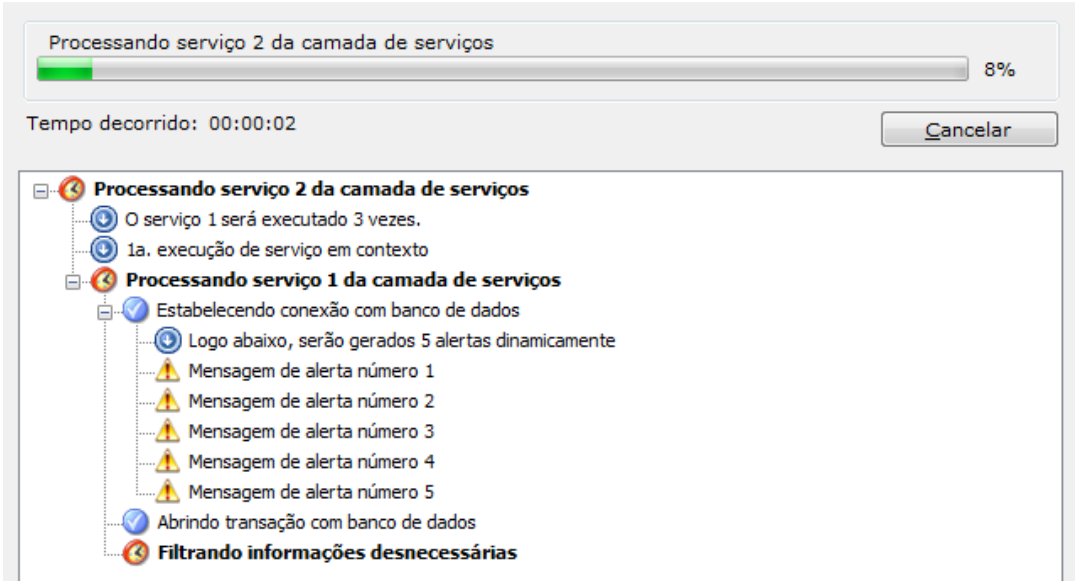


Figura 4.6 – Encadeamento do serviço 1 dentro do serviço 2

Ainda neste método, temos a execução de um serviço da camada de dados, o método *GetRandomNumber* da interface *IDataFacade*.

```
using(LinkedOperationContext linkedOperation =
        _Monitor.CreateLinkedOperationContext(_DataFacade, 400))
{
        randomNumber = _DataFacade.GetRandomNumber();
}
```

Figura 4.7 – Execução de serviço em uma outra camada

Neste caso, o método `_Monitor.CreateLinkedOperationContext(_DataFacade, 400)` faz com que a classe *ServicesMonitor* aguarde a execução de um serviço na fachada, referenciada através da variável *_DataFacade*, com custo 400 no contexto de execução do *Service2*.

4.2.3 Método *Service3*

O método *Service3* exemplifica a execução de uma suboperação cujo custo é calculado dinamicamente em função de um argumento recebido e executa um método recursivo da camada de dados para cálculo de um número fatorial.

```
using (LinkedOperationContext binding =
        _Monitor.CreateLinkedOperationContext(_DataFacade, n * 1000))
{
        result = _DataFacade.GetFactorial(n);
}
```

Figura 4.8 – Criação de suboperação com custo dinâmico

Quanto maior o valor do argumento *n* recebido pela entrada, maior será o custo da operação. Assim, podemos definir o custo facilmente em função de quaisquer variáveis existentes, em tempo de execução.

O código abaixo mostra como o método *GetFactorial* foi implementado na camada de dados.

```
using (ManagedOperationContext mainOperation=_Monitor.CreateOperationContext("", n+1))
{

if (n == 0)
{
        _Monitor.WriteMessage("Caso base da recursão fatorial!");
        value = 1;
        _Monitor.WriteMessage("Fatorial de 0 é 1");
}
else
{
        _Monitor.WriteMessage("Será calculado fatorial de " + (n - 1).ToString());

        int fact_low = 0;
        using (OperationContextBinding binding = _Monitor.CreateBindingContext(n))
                fact_low = GetFactorial(n - 1);

        value = n * fact_low;

        if(!mainOperation.DoCancel)
                _Monitor.WriteMessage("Fatorial de " + n.ToString() + " é " + value);
}

}
```

Figura 4.9 – Exemplo de método recursivo, fatorial

O código fonte aparentemente grande para implementar uma função simples como o fatorial é devido às emissões de notificação que foram inseridas no código fonte. O custo total definido para a operação fatorial(n) é n+1, já que serão processados n+1 passos (considerando que o caso base é zero). Assim, antes de fazer a chamada recursiva fatorial(n-1), cria-se uma operação vinculada com custo n. O resultado da execução do algoritmo é representado pela árvore de notificações da seguinte forma:

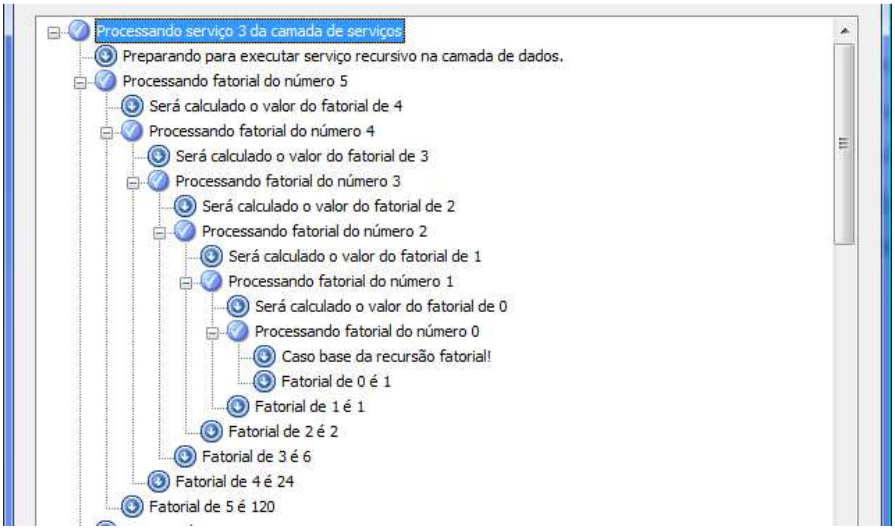


Figura 4.10 – Árvore de notificações do método recursivo

CAPÍTULO 5

CONCLUSÃO

Neste capítulo, apresentamos uma avaliação objetiva dos resultados positivos e negativos do trabalho. São apresentadas as principais contribuições resultantes, dificuldades encontradas, limitações, possibilidades de melhoria e trabalhos futuros.

5.1 RESULTADOS OBTIDOS E CONTRIBUIÇÕES

Como resultado das pesquisas e do esforço de implementação realizado ao longo deste trabalho, obtivemos:

(1) Levantamento das principais necessidades e problemas envolvidos no processo de monitoramento de execução dos serviços pelas aplicações;

(2) Definição de conceitos e abstrações necessários para a sistematização e tratamento metódico do problema de monitoramento;

(3) Concepção e implementação de um *framework*, *ServiceMon Fundamentals*, baseado na *Plataforma .Net* da *Microsoft*, capaz de representar os conceitos definidos, estabelecer as principais estruturas de controle de monitoramento e definir as interfaces de comunicação que permitem o compartilhamento de informações de monitoramento entre provedor e consumidor;

(4) Concepção e implementação de *bibliotecas* baseadas no *framework, ServiceMon ProviderToolkit e ConsumerToolkit,* com implementações específicas e concretas para facilitar o desenvolvimento das aplicações;

(5) Implementação de uma aplicação que ilustra o funcionamento das bibliotecas, utiliza e verifica a todos os requisitos de monitoramento previstos.

É importante perceber que, caso se considere que as bibliotecas implementadas não atendem bem a um determinado tipo específico de aplicação, ainda assim é possível utilizar a *framework* para criar uma nova biblioteca para resolver uma situação de domínio específico.

A Plataforma .Net e a linguagem C#, juntas, oferecem muitos recursos úteis ao desenvolvimento deste trabalho, como a interface *IDisposable*, a instrução *using*, eventos, *structs* e enumerações. A utilização destes recursos permitiu a construção de um *framework* que pode ser utilizado facilmente pelo desenvolvedor, sem aumento significativo da complexidade do código. Além disso, isto tornou possível a redução da quantidade de esforço de implementação deste trabalho.

5.2 DIFICULDADES E LIMITAÇÕES

Ao longo da fase de análise e concepção do problema envolvido neste trabalho, algumas limitações foram percebidas. Algumas destas limitações não foram resolvidas porque (1) aumentariam significativamente a complexidade do projeto, de maneira que o esforço e tempo de pesquisa requerido extrapolariam o escopo do trabalho ou (2) as limitações não eram suficientemente críticas por representar casos particulares e bastante incomuns em aplicações comerciais típicas.

As principais dificuldades e limitações encontradas foram:

(1) Legibilidade e minimização do impacto no código fonte – Apesar de termos utilizado estruturas de linguagem específicas e um modelo enxuto e objetivo para a utilização das bibliotecas, encapsulando todo o controle de monitoramento nas bibliotecas, ainda assim, o código fonte de um serviço monitorável sofre algum impacto e precisa ser modificado para fazer uso dos recursos da biblioteca. As instruções do código fonte que visam à implementação do real propósito do serviço não fica completamente dissociada das instruções relacionadas com o controle de monitoramento. O desenvolvedor do serviço precisa conhecer e utilizar as bibliotecas adequadamente ao implementar o serviço monitorável. Na seção que fala sobre trabalhos futuros, mais adiante, sugerimos meios de minimizar este problema.

(2) Suporte à execução de serviços concorrentes – Embora o *framework* desenvolvido seja suficientemente genérico para representar todos os tipos de monitoramento, com ou sem concorrência, as bibliotecas criadas foram projetadas para utilização em aplicações com a limitação de que uma entidade que possua serviços monitoráveis não pode ter mais de um serviço

executando concorrentemente. Isto acontece porque o controle de monitoramento é realizado com base na alteração de atributos da entidade e empilhamento de métodos. Escolhemos esta abordagem porque ela permite que a biblioteca controle todo o processo de encadeamento implicitamente, requerendo menos controle explícito por parte do desenvolvedor, o que simplifica o trabalho, torna o uso da biblioteca mais simples e o código fonte menos acoplado, o que ameniza a limitação anterior (1).

(3) Determinação arbitrária dos custos de execução das operações – Neste trabalho, permitimos que o desenvolvedor defina, de acordo com o seu conhecimento, quais devem ser os custos de cada operação realizada pelo serviço ao longo de sua execução. Se, por um lado, damos a flexibilidade para o desenvolvedor utilizar quaisquer critérios que considerar mais convenientes para cada caso, criamos para ele a dificuldade de estabelecer precisamente e coerentemente os custos das operações realizadas. Estes custos podem depender do tamanho das entradas e de informações que só existem no contexto da execução do serviço. Na seção que fala de trabalhos futuros e melhorias, mais adiante, falaremos sobre uma técnica que pode ajudar a solucionar este problema.

(4) Dependência da regra de negócio – a flexibilização da determinação dos custos das operações, através do uso de variáveis da aplicação no cálculo dos custos, gera uma independência indesejável entre as regras de negócio e o monitoramento. Isto pode gerar problemas de manutenção futuros, quando as variáveis da regra de negócio precisarem ser alteradas.

As limitações apresentadas não neutralizam as contribuições e facilidades que o projeto traz para o desenvolvimento de aplicações típicas. Todas elas podem ser resolvidas e/ou significativamente minimizadas através da realização de melhorias, extensões ou derivações deste trabalho, conforme explicado na próxima seção.

5.3 TRABALHOS FUTUROS

Identificamos algumas formas de melhorar este trabalho através da resolução das limitações apresentadas ou da criação de abordagens mais apropriadas para serem utilizadas em aplicações específicas:

(1) Criação de uma linguagem de programação com estruturas especificamente voltadas para dar suporte ao monitoramento da execução dos métodos. A linguagem daria suporte nativo aos conceitos de monitoramento aqui definidos. O *framework* seria utilizado internamente pelo compilador para transformar código fonte do desenvolvedor em código que utiliza o framework para resolver o problema. Isto poderia aumentar a eficiência e diminuir o grau de complexidade do código fonte da aplicação;

Uma análise imediata leva a crer que a utilização do paradigma orientado a aspectos [19] pode ser útil para este trabalho, uma vez que, para conseguir obter controle sobre a execução dos serviços, precisamos executar rotinas de controle nos inícios e términos de execução dos métodos envolvidos nos serviços da aplicação. É importante perceber que, além dessas rotinas, várias outras precisam ser executadas no corpo de cada método, em pontos específicos definidos explicitamente pelo programador, que não podem ser automatizadas porque são inerentes à semântica do algoritmo e ao domínio da aplicação. Uma linguagem que dê suporte a esse monitoramento precisa ter estruturas específicas que permitam a demarcação explícita das operações realizadas dentro dos métodos.

(2) Utilização de um modelo estatístico que permita criar um indicador do tempo restante previsto para a conclusão das operações, em função do tempo realizado e dos custos das operações realizadas e com execução pendente;

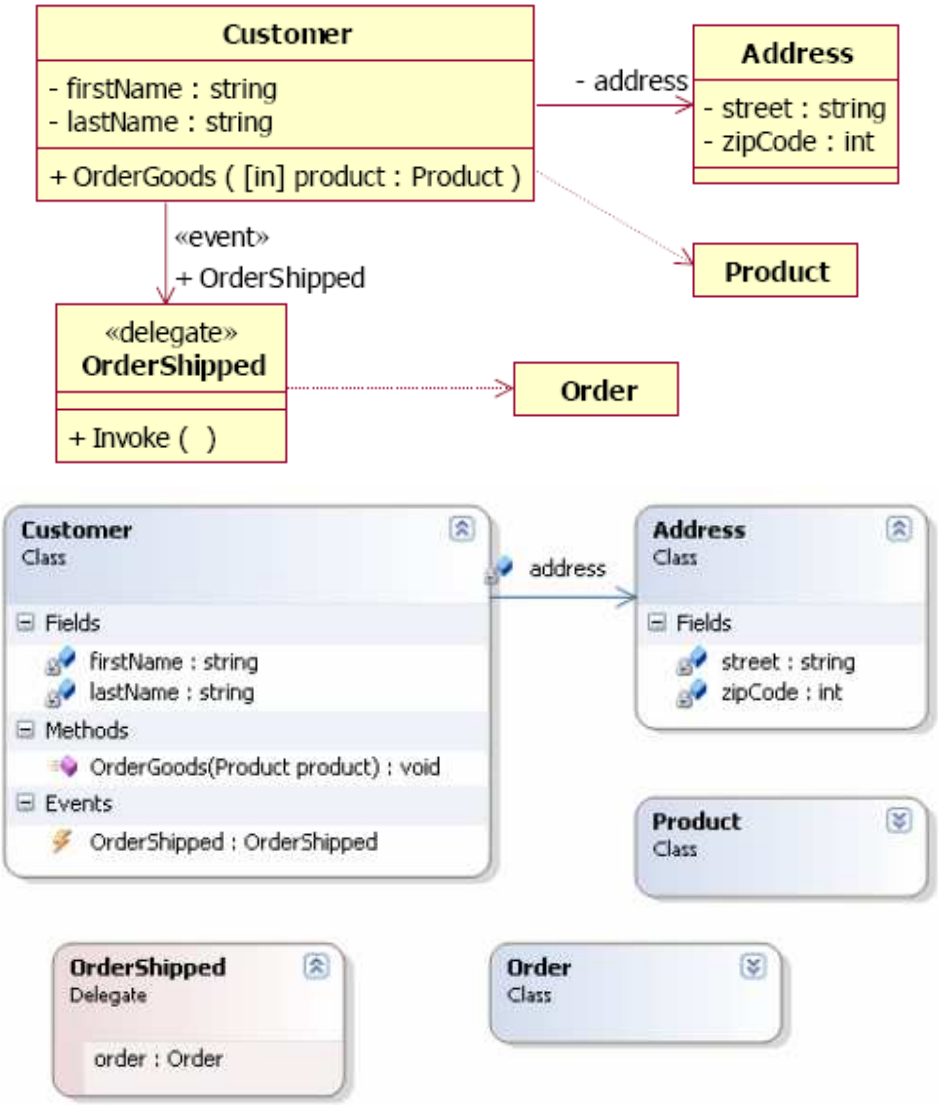
(3) Integração do modelo com ferramentas de *Profiling* [11] [9] [10] para permitir a análise automática de todas as operações realizadas pelo serviço e a utilização de uma heurística para determinação dos custos das operações em função do tempo médio de execução.

(4) Criação de uma biblioteca equivalente à *ServiceMon ProviderToolkit* que utilize um formato de controle de monitoramento que não esteja vinculado ao estado da entidade provedora de serviços e possibilite o desenvolvimento de entidades cujos serviços possam ser executados concorrentemente.

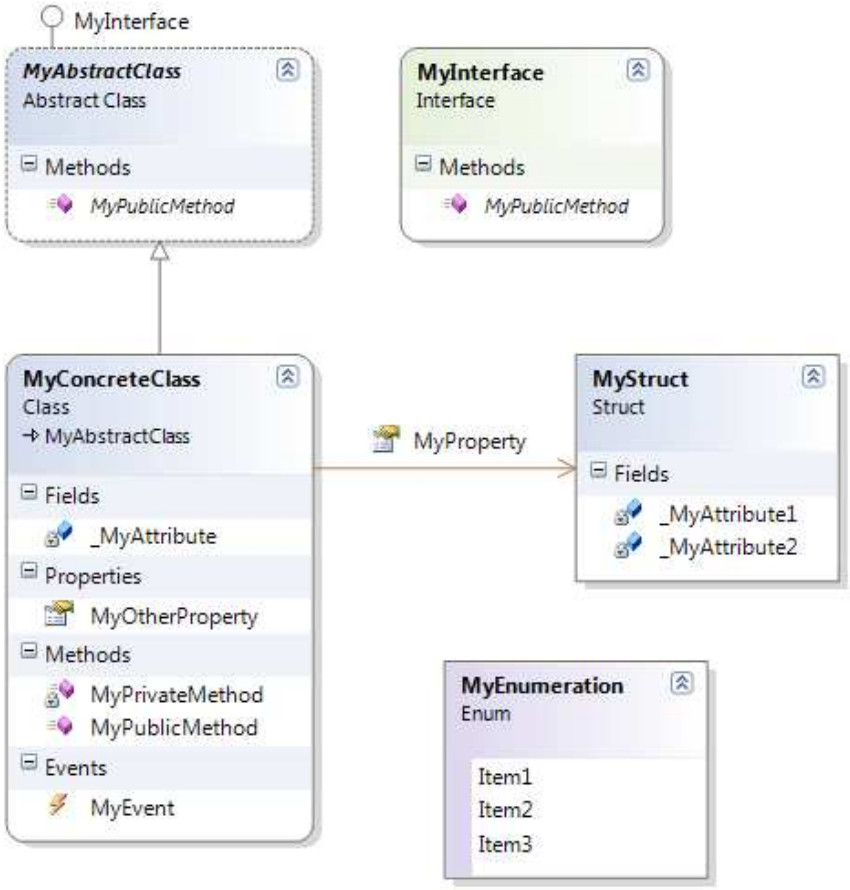
APÊNDICE A – NOTAÇÃO DOS DIAGRAMAS

A notação utilizada neste trabalho é a do *Microsoft Visual Studio Class Designer*, similar à notação UML. Por isso, não será difícil incorporar o formato utilizado. Além disso, a notação é bastante intuitiva se já houver familiaridade com as estruturas da *Plataforma .Net* [20]. Este apêndice apresenta as principais representações gráficas utilizadas nos diagramas de classes deste trabalho.

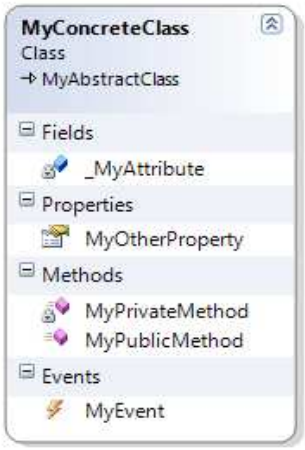
Primeiro, para mostrar o grau de similaridade entre a notação utilizada e UML, considere as duas figuras a seguir (ambas representam o mesmo modelo – a primeira em notação UML, a segunda em notação do *Microsoft Visual Studio Class Designer*).

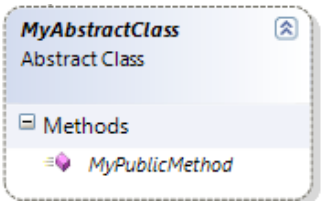
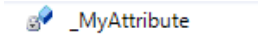
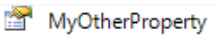
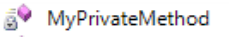
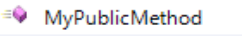
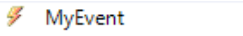
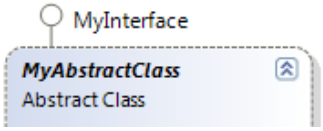
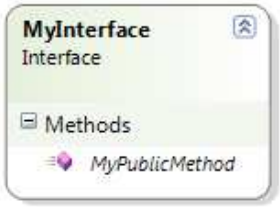
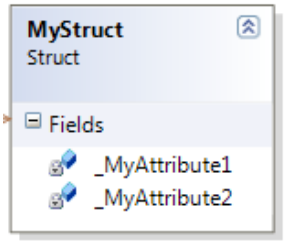
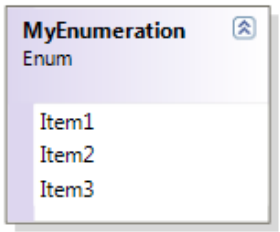
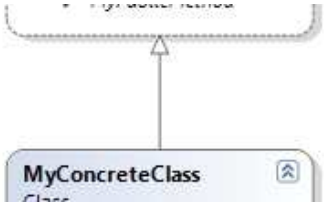


Por analogia, analisando as figuras apresentadas, já é possível inferir as diferenças existentes entre as duas representações. A figura a seguir reúne todas as representações utilizadas no trabalho.



A tabela abaixo descreve cada parte do diagrama.

	Retângulo azulado com extremidades abauladas com borda contínua é utilizado para representar classes concretas. Os membros da classe aparecem organizados 4 compartimentos: *Fields, Properties, Methods* e *Events*, conforme o tipo.

	O retângulo azulado com extremidades abauladas com borda pontilhada é utilizado para representar classes abstratas.
	Atributo privado.
	Propriedade pública.
	Método privado.
	Método público.
	Evento público.
	Indica que a classe *MyAbstractClass* implementa a interface *MyInterface*.
	O retângulo esverdeado é utilizado para representar intefaces.
	O retângulo azulado é utilizado para representar *structs*.
	O retângulo arroxeado é utilizado para representar enumerações. O interior do retângulo lista os itens da enumeração.
	A seta com extremidade cheia é utilizada para representar herança entre classes.

	Representação de uma propriedade por meio de uma seta de associação.

A representação utilizada é simples. Para obter mais detalhes sobre esta notação, consulte a especificação técnica [15].

REFERÊNCIAS BIBLIOGRÁFICAS

1. **Rao, Ramana B. e Rajaraman, Anand.** *Method and Apparatus for Time Estimation and Progress Feedback on Distal Access Operations.* s.l. : Xerox Corporation, 1995.

2. **Kogut, Paul.** The Software Architecture Renaissance. *STSC's Crosstalk newsletter.* 1994.

3. **Gama, Erich.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1994.

4. *Effects of Reuse on Quality, Productivity and Economics.* **Lim, W. C.** s.l. : IEEE Software, 1994.

5. *The Mythical Man-Month: Essays on Software Engineering.* **Brooks, F. P. J.** 20th anniversary edition, s.l. : Addison-Wesley, 1995.

6. Framework. *Wikipédia.* [Online] Wikipédia, 24 de Junho de 2008. [Citado em: 15 de Junho de 2008.]

7. *Service-Oriented Computing: State of the Art and Research Challenges.* **Papazoglou, Michael, et al.** s.l. : IEEE Computer Society, 2007.

8. **Papazoglou, Michael, et al.** *Service-Oriented Computing Research Roadmap.* 2006.

9. **JetBrains.** dotTrace. [Online] JetBrains, 2008. http://www.jetbrains.com/profiler/?gclid=CMqzt6ePlJQCFQEnGgodVjl7uQ.

10. **SciTech Software.** Find Memory Leaks and Optimize Memory Usage in any .NET Program. *.Net Memory Profiler.* [Online] SciTech Software, 2008. [Citado em: 15 de Junho de 2008.] http://memprofiler.com/.

11. **Embarcadero Technologies.** *Why you need a code profiler.* San Francisco : s.n., 2004.

12. **scott.** Diagnostics and Logging In ASP.NET. *OdeToCode.com.* [Online] 20 de Novembro de 2004. [Citado em: 15 de Junho de 2008.] http://www.odetocode.com/Articles/294.aspx.

13. **.NET Framework Class Library.** EventLog Class. *MSDN Library.* [Online] Microsoft, 2008. http://msdn.microsoft.com/en-us/library/system.diagnostics.eventlog.aspx.

14. **Okapi Framework.** Log Interface Specification. *Okapi Framework.* [Online] 2008. http://okapi.sourceforge.net/ILog.html.

15. **Stoecker, Matthew A.** Visual Studio 2005 Class Designer. *MSDN Library.* [Online] Microsoft, Maio de 2004. http://msdn.microsoft.com/en-us/library/aa288743(VS.71).aspx.

16. **MSDN Library.** Events Tutorial. *C# Programmer's Reference.* [Online] Microsoft, 2008. [Citado em: 15 de Março de 2008.] http://msdn.microsoft.com/en-us/library/aa645739(VS.71).aspx.

17. **ECMA International.** *C# Language Specification (ECMA-334).* 2006.

18. IDisposable Interface. *MSDN Library.* [Online] Microsoft. [Citado em: 10 de Junho de 2008.] http://www.msdn.com.

19. **Sarathy, Viji.** Aspect-oriented programming (AOP) and the .NET Framework. *WinDevelopment.com.* [Online] 04 de 13 de 2005. [Citado em: 25 de Março de 2008.] http://searchwindevelopment.techtarget.com/tip/0,289483,sid8_gci1278201,00.html.

20. *Visual Studio Class Designer and Unified Modelling Language.* **Katona, Krisztina.** Budapest : Hungarian Joint Symposium on Intelligent Systems, 2006.

21. **Sommerville, Ian.** *Software Engineering, 7th Edition.* s.l. : Addison-Wesley, 2004.

22. *Common Misconceptions about Software Architecture.* **Kruchten, Philippe.** s.l. : The Rational Edge, 2001.

23. **Martin, Robert C.** Design Principles And Design Patterns. *ObjectMentor.* 2000.

24. **Budd, Timothy.** *An Introduction to Object-Oriented Programming, 3rd Edition.* 2002.