FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Reengenharia de aplicações baseadas em componentes: Aplicação ao Processo Clínico

Luís Miguel Alves Moreira da Ponte

VERSÃO FINAL

Relatório de Projecto
Mestrado Integrado em Engenharia Informática e Computação

Orientador: Ademar Manuel Teixeira de Aguiar (Professor Auxiliar)

Julho de 2009

Reengenharia de aplicações baseadas em componentes: Aplicação ao Processo Clínico

Luís Miguel Alves Moreira da Ponte

Relatório de Projecto
Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: João Francisco de Sousa Cardoso (Professor Auxiliar Convidado)

_____

Arguente: José Maria Fernandes (Professor Auxiliar Convidado — Universidade de Aveiro)

Vogal: Ademar Manuel Teixeira de Aguiar (Professor Auxiliar)

15 de Julho de 2009

Resumo

O projecto que aqui se apresenta tem como foco principal o estudo do processo de migração entre duas tecnologias de suporte ao desenvolvimento de aplicações baseadas em componentes. Especificamente, foi analisada a problemática de migração da aplicação de Processo Clínico, desenvolvida pela *GlinttHS*.

O Processo Clínico é um sistema de informação que suporta a actividade clínica dos profissionais das várias áreas funcionais de um estabelecimento de saúde. Foi originalmente desenvolvido utilizando a *Composite UI Application Block* (CAB), que tem o objectivo de facilitar a composição de vários elementos, cada um com a sua interface com o utilizador, integrando-os numa aplicação única. CAB apenas suporta componentes criados em *Windows Forms*, pelo que, de forma a tirar o melhor partido de novas tecnologias como a *Windows Presentation Foundation* (WPF), seria necessário migrar as soluções desenvolvidas em CAB para a sua substituta, a *Composite Application Library* (mais conhecida por *Prism*).

A migração entre as duas tecnologias não poderia ser realizada num só passo, já que isso implicaria uma reestruturação profunda de todo o sistema e consequente paragem no desenvolvimento por um período demasiado longo. Assim, foi desenvolvida uma infra-estrutura de migração que encapsula as funcionalidades das duas tecnologias e possibilita um processo de migração gradual e incremental, sem implicar um custo significativo de desempenho.

Adicionalmente, foi desenvolvido um módulo de definição de protocolos de tratamento oncológico, integrado no Processo Clínico, que utiliza a infra-estrutura desenvolvida e confirma a sua validade e aplicabilidade.

Abstract

The project presented in this report is mainly focused on the study of the migration process between two technologies supporting the development of component-based applications. The migration of the Clinical Process application, developed by *GlinttHS*, was specifically analysed.

The Clinical Process is an information system supporting the clinical activities of all professionals in the various functional areas of a healthcare institution. It was originally developed using the *Composite UI Application Block* (CAB), which enables the composition of many user interface elements, integrating them in a single application. Because CAB only supports components created in *Windows Forms*, to get the most out of new technologies like *Windows Presentation Foundation* (WPF) it would be necessary to migrate the existing solutions, developed in CAB, to its replacement, the *Composite Application Library* (also known as *Prism*).

The migration between the two technologies could not be done in a single step. That would entail a deep restructuring of the whole system, forcing the development process to stop for too long. As such, a migration infrastructure was developed, encapsulating the features of both technologies and allowing the migration process to be gradual and incremental, without introducing a significant performance loss.

Furthermore, an oncologic treatment protocol definition module was developed. The module is integrated in the Clinical Process application, using the developed migration infrastructure and confirming its validity and applicability.

Agradecimentos

Deixo os meus agradecimentos à *GlinttHS*, por me ter proporcionado a oportunidade de realizar este projecto, e em particular ao Eng. Nuno Ribeiro, meu orientador na empresa, que me prestou sempre toda a ajuda necessária à realização deste trabalho.

Da empresa, os meus colegas de sala, Daniel Alves, Rui Pinto, Vítor Videira e Francisco Correia, também não podem ser esquecidos. Durante todo o projecto, mostraram-se sempre disponíveis para me esclarecer todas as dúvidas e foram preponderantes para a minha integração na empresa.

Apesar de também ser meu colega de sala, o José Carvalho merece um agradecimento em separado, já que foi com ele que trabalhei durante as primeiras fases do projecto, tendo sido importante para a concretização dos objectivos.

Da FEUP, começo por agradecer ao meu orientador, Prof. Ademar Aguiar, por todo o apoio prestado durante o projecto.

Ao Prof. António Augusto Sousa, director de curso, pelo tempo que dispensou para me esclarecer algumas dúvidas sempre que precisei, apesar da sua agenda preenchida.

Agradeço também à minha família, por sempre me ter apoiado durante todos estes anos de estudos.

E finalmente, como não podia deixar de ser, à Sara Sena Esteves, por estar *sempre* presente.

Obrigado,

Luís Ponte

Índice

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

API	*Application Programming Interface*
CAB	*Composite UI Application Block*
CBSE	*Component-based Software Engineering*
CORBA	*Common Object Request Broker Architecture*
CSS	*Cascading Style Sheet*
GlinttHS	*Global Intelligent Technologies – Healthcare Solutions*
HTML	*HyperText Markup Language*
MCDT	Métodos Complementares de Diagnóstico e Tratamento
SOA	*Service-Oriented Architecture*
SOAP	*Simple Object Access Protocol*
WPF	*Windows Presentation Foundation*
XAML	*eXtensible Application Markup Language*
XML	*eXtensible Markup Language*

Capítulo 1

Introdução

Este documento tem como objectivo principal descrever o trabalho realizado no âmbito do projecto de fim de curso do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto.

Neste capítulo será feita uma introdução geral ao tema do projecto, apresentando o contexto e enquadramento do mesmo, assim como uma breve descrição da motivação e dos principais objectivos. Será também dada uma perspectiva da organização e conteúdo do restante documento.

1.1 Contexto/Enquadramento

A rápida evolução tecnológica a que se assiste actualmente no mundo das tecnologias de informação, aliada à importância capital que estas assumem no mundo empresarial, leva ao aumento do nível de exigência das organizações relativamente aos sistemas informáticos que utilizam na sua actividade diária. Particularmente, o aparecimento de novas tecnologias de apresentação, que possibilitam o desenvolvimento de interfaces gráficas muito mais ricas e interactivas, permite oferecer aos utilizadores finais uma experiência de utilização muito mais agradável e simples, tornando a usabilidade num dos factores primordiais para o sucesso de qualquer solução informática. Assim, o constante acompanhamento dessas evoluções é essencial para garantir a competitividade de uma empresa que desenvolva aplicações de software.

No entanto, isso constitui um desafio, uma vez que implica frequentemente migrações para novas tecnologias. Quando estão em causa soluções complexas, o problema agrava-se, uma vez que não é possível migrar todo o sistema de uma só vez, substituindo-o por uma nova versão. Assim, o problema das migrações para novas tecnologias assume-se como um problema de reengenharia de software e é precisamente nesse contexto que se enquadra este projecto.

O trabalho foi desenvolvido na *Global Intelligent Technologies – Healthcare Solutions* (*GlinttHS*), uma empresa de soluções informáticas para a área da saúde. A *GlinttHS* fornece aplicações que cobrem todos os processos relativos aos cuidados de saúde, desde a vertente clínica à gestão hospitalar. Este projecto em particular foca-se na aplicação de Processo Clínico. Esta aplicação constitui um Sistema de Informação integrado que dá suporte a todos os departamentos clínicos e áreas funcionais de um estabelecimento de saúde. A informação é centrada no doente, sendo sempre apresentada de forma simples e clara, de modo a facilitar o acesso à mesma por parte de todos os profissionais de saúde. Assim, os principais objectivos do Processo Clínico são os seguintes:

- Melhorar os cuidados de saúde prestados ao doente;
- Partilhar informação clínica entre os profissionais de saúde;
- Diminuir o erro;
- Melhorar a forma como a informação é obtida, registada e disponibilizada;
- Garantir a mobilidade e acesso remoto;
- Melhorar o suporte à decisão clínica;
- Acesso fácil a standards terapêuticos;
- Racionalização de recursos.

A empresa é actualmente um dos principais parceiros da *Microsoft* a nível nacional, desenvolvendo a grande maioria das suas aplicações sob a plataforma *.NET*. A aplicação de Processo Clínico é presentemente baseada na *Composite UI Application Block* (CAB), uma *framework* que facilita o desenvolvimento de aplicações compostas por vários módulos, cada um com a sua própria interface com o utilizador, desenvolvida em *Windows Forms*. No entanto, essas duas tecnologias já estão desactualizadas. CAB foi substituído por *Prism*, que também tem como objectivo auxiliar o desenvolvimento de aplicações compostas, mas agora com interfaces em WPF (*Windows Presentation Foundation*), considerada pela *Microsoft* como a tecnologia de eleição para interfaces gráficas em ambientes *Windows*.

Assim, a empresa tem como objectivo a médio prazo, a migração de todas as suas soluções para *Prism* e WPF.

1.2 Projecto

No contexto acima apresentado, o projecto pode ser dividido em duas componentes distintas. A primeira componente consiste num estudo e desenvolvimento de um processo e infraestrutura de migração que encapsule as diferenças entre as duas tecnologias e possibilite uma transição gradual entre elas. O projecto inclui também uma segunda componente, que prevê a implementação de um módulo de definição de protocolos de tratamento oncológico. Esse módu-

lo, para além de responder a uma necessidade real da empresa, tem como objectivo adicional confirmar a viabilidade do processo de migração desenvolvido na primeira componente.

Os protocolos de tratamento oncológico podem ser compostos por vários tipos de actividades, nomeadamente, exames, medicação ou tratamentos de radioterapia. São geralmente periódicos, sendo divididos em vários ciclos compostos por várias sessões de tratamento, e correspondem a normas internacionais de tratamentos oncológicos.

A *GlinttHS* já possuía um leque de soluções que permitiam definir protocolos em cada um dos tipos de actividades, ou seja, numa aplicação era definido o protocolo de medicação, enquanto os exames eram definidos noutra aplicação, sem que existisse qualquer integração entre elas. O foco principal do módulo de definição de protocolos era a criação de um sistema que permitisse definir um protocolo de tratamento como uma entidade que agregasse os diferentes sub-protocolos relativos a cada tipo de actividades.

1.3 Motivação e Objectivos

Dado o contexto em que se realizou o projecto, a principal motivação era encontrar a melhor forma de realizar a migração entre duas *frameworks* de suporte à composição de elementos visuais. A migração deveria ser realizada de forma suave, de forma a não implicar a paragem do desenvolvimento das soluções durante o período de tempo necessário a uma migração completa, o que seria incomportável para a empresa.

Assim, o projecto tinha dois objectivos principais: (1) criar uma infra-estrutura de suporte à migração e (2) criar um módulo de definição de protocolos de tratamento que, para além de responder a uma necessidade da empresa, pudesse servir como prova de conceito, exemplificando e validando a infra-estrutura de migração desenvolvida.

Para isso, o projecto foi dividido em três fases:

1. **Estudo de tecnologias –** na primeira fase do projecto foram estudadas as tecnologias envolvidas e as soluções já desenvolvidas;
2. **Desenvolvimento da infra-estrutura de migração –** na segunda fase do projecto foi desenvolvida a infra-estrutura de migração que permite migrar as soluções já desenvolvidas de CAB para *Prism*;
3. **Desenvolvimento do módulo de definição de protocolos** – na última fase do projecto, como prova de conceito e demonstração da viabilidade da infra-estrutura de migração, foi desenvolvido o módulo de definição de protocolos de tratamento.

Principalmente durante a segunda fase, o trabalho foi realizado em colaboração com José Carvalho, também aluno do mesmo curso.

1.4 Estrutura do relatório

Para além deste capítulo introdutório, este relatório contém mais cinco capítulos. No capítulo 2 é feita uma descrição mais pormenorizada do problema e do projecto proposto pela *GlinttHS*. No capítulo 3 é apresentada uma revisão teórica sobre os conceitos principais relacionados com o projecto em causa, assim como uma breve apresentação das tecnologias envolvidas. O quarto capítulo é dedicado à descrição detalhada da infra-estrutura de migração desenvolvida, sendo seguido pelo capítulo 5, que apresenta os detalhes de arquitectura e implementação do módulo de definição de protocolos que foi desenvolvido. Finalmente, no sexto e último capítulo, são apresentadas as conclusões relevantes, assim como uma análise crítica do trabalho desenvolvido.

Capítulo 2

Descrição do Problema

O projecto proposto pela *GlinttHS* tinha dois objectivos principais:

1. Criar uma infra-estrutura de suporte à migração entre CAB e *Prism*;
2. Criar um módulo de definição de protocolos de tratamento oncológico.

Este capítulo irá descrever detalhadamente cada um desses objectivos, o seu contexto e motivação.

2.1 Criação de uma infra-estrutura de migração

A empresa em que foi desenvolvido o projecto, a *GlinttHS*, possui um leque alargado de soluções na área de software clínico, procurando cobrir todas as necessidades dos profissionais da área. A grande maioria dessas soluções (todas, com excepção da Gestão Hospitalar) são desenvolvidas sob a plataforma *.NET* da *Microsoft*. As aplicações *desktop* em particular, como é o caso do Processo Clínico, são na sua maioria desenvolvidas em *Windows Forms*.

No entanto, a *Microsoft* considera WPF como a tecnologia de eleição para o desenvolvimento de interfaces com o utilizador. Assim, como parceiro estratégico da *Microsoft* em Portugal, a *Glintt* tem como objectivo migrar todas as suas aplicações para a nova tecnologia.

A aplicação de Processo Clínico em particular, é desenvolvida sobre a *framework* CAB, que foi pensada para ser utilizada com *Windows Forms*, o que dificulta a migração para WPF. Além disso, CAB também é uma tecnologia já desactualizada, tendo sido substituída por *Prism*, essa sim, direccionada ao desenvolvimento em WPF.

Apesar de já terem sido realizados alguns trabalhos com o objectivo de estudar a interoperabilidade entre CAB e WPF [Gom08], a empresa considera que a melhor opção é migrar todas as soluções de CAB para *Prism*, substituindo as interfaces com o utilizador desenhadas em *Windows Forms* por alternativas em WPF.

Dada a complexidade e dimensão das soluções envolvidas, e especificamente do Processo Clínico, em que este projecto é focado, não é viável realizar a migração entre essas tecnologias de uma vez só, seguindo uma abordagem do tipo *big-bang*. CAB, sendo uma *framework* de suporte ao desenvolvimento de aplicações baseadas em componentes, assume um papel fulcral na arquitectura do sistema, e a sua substituição implicaria uma profunda reestruturação que poderia demorar vários meses. Durante esse tempo, não poderia ser desenvolvida qualquer nova funcionalidade na aplicação ou corrigido qualquer problema, criando potenciais problemas no suporte aos clientes. Além disso, a migração de *Windows Forms* para WPF implicaria que todos os elementos de interface com o utilizador fossem redesenhados. Em suma, a migração, a ser realizada dessa forma, implicaria a substituição de praticamente toda a camada de apresentação do sistema. Tal abordagem constitui um risco demasiado elevado para a empresa.

Perante este problema, a solução é adoptar uma abordagem incremental, em que a migração vai sendo feita de forma gradual, mantendo sempre o sistema em funcionamento e podendo ser realizada ao ritmo mais adequado para a empresa. Nasce assim a ideia de criar aquilo que seria denominado uma *infra-estrutura de migração*, fazendo a ponte entre a aplicação final e a *framework* que está a ser utilizada, seja ela CAB ou *Prism*. Os requisitos principais dessa infra-estrutura seriam os seguintes:

- Encapsular as principais funcionalidades de CAB e de *Prism* perante as aplicações finais. Dessa forma, as aplicações podem ser desenvolvidas sem dependerem explicitamente de CAB ou de *Prism*;
- Garantir que a introdução da infra-estrutura na solução possa ser realizada de forma gradual;
- Garantir que todas as operações realizadas através da infra-estrutura tenham exactamente o mesmo resultado do que se fosse utilizada cada uma das *frameworks* directamente;
- Permitir o desenvolvimento de interfaces com o utilizador em WPF e em *Windows Forms*, integradas na mesma aplicação;
- Não introduzir uma penalização de desempenho significativa.

Caso estes objectivos fossem cumpridos, a infra-estrutura possibilitaria a migração entre CAB e *Prism* e entre *Windows Forms* e WPF de forma suave, não interrompendo o desenvolvimento.

2.2 Definição de protocolos de tratamento oncológico

O segundo objectivo o projecto consistia na criação de um módulo de definição de protocolos de tratamento oncológico a ser integrado na aplicação de Processo Clínico. Este objectivo surge com duas motivações distintas. Por um lado, o módulo responde a uma necessidade já identificada na empresa, colmatando uma lacuna nas soluções clínicas existentes para tratamen-

tos oncológicos. Por outro lado, aliado ao desenvolvimento da infra-estrutura de migração, este módulo serviria de prova de conceito, constituindo o primeiro módulo do Processo Clínico totalmente desenvolvido em WPF e utilizando a infra-estrutura de migração, sem que exista qualquer dependência explícita de CAB ou de *Prism*.

Os protocolos de tratamento oncológico são constituídos por exames[1], medicação e outros actos médicos, organizados em ciclos constituídos por várias sessões. O módulo de definição de protocolos tem como objectivo concentrar num só local a definição dos vários elementos do protocolo, facilitando assim a prescrição de um dado protocolo a um doente.

Para isso, são também guardadas informações sobre a aplicabilidade do protocolo. Cada protocolo deve ter uma lista de diagnósticos a que é aplicável, assim como uma lista de critérios de inclusão e exclusão, que especificam em que situações o protocolo pode ser ou não prescrito. Esses critérios funcionarão como uma *checklist*, auxiliando o médico na prescrição do mesmo.

Apesar de numa primeira fase o módulo de definição de protocolos ser mais direccionado à aplicação a protocolos de tratamento oncológico, considera-se importante que este seja desenvolvido tendo em conta a possível aplicabilidade a outro tipo de protocolos de tratamento. O módulo deveria ser, portanto, o mais genérico possível.

Nas soluções que a *GlinttHS* oferecia aos seus clientes no início destes projecto já era possível definir protocolos de tratamento. No entanto, existia uma separação fundamental entre os protocolos definidos na aplicação de Gestão Hospitalar, que apenas contemplam actos médicos e exames, e os protocolos definidos na aplicação de Farmácia, que se cingem a medicação. Não existia qualquer tipo de integração entre eles, e essa era a lacuna que este módulo definição de protocolos pretendia colmatar.

O módulo de definição de protocolos é complementado por um módulo de prescrição e agendamento, desenvolvido num projecto que decorreu em paralelo.

[1] Segundo a terminologia da área, normalmente designados MCDT – Métodos Complementares de Diagnóstico e Tratamento.

Capítulo 3

Revisão bibliográfica e tecnológica

Neste capítulo serão abordados dois conceitos essenciais para o projecto desenvolvido: **Reengenharia de Software** (*Software Reengineering*) e **Engenharia de Software Baseada em Componentes** (*Component-based software engineering* - CBSE). Foram dois temas omnipresentes desde o início do projecto, pelo que foram alvo de um estudo aprofundado. Serão aqui apresentados os principais conceitos e fundamentos teóricos em cada uma das duas temáticas.

O estudo da reengenharia de software é justificado pela necessidade de migração entre as duas tecnologias, que foi abordada na parte inicial do projecto. O estudo será focado nos processos, padrões e boas práticas recomendadas pela literatura.

Relativamente ao segundo tema, o estudo de Engenharia de Software baseada em Componentes foi importante, uma vez que o processo de reengenharia teria de ser aplicado num contexto de soluções compostas por vários componentes, implicando, portanto, o estudo desses paradigmas. Serão identificados os vários tipos de integração de componentes, como parte de uma análise de âmbito mais alargado, seguida por uma descrição mais pormenorizada da integração de componentes ao nível da interface com o utilizador. Serão abordadas as duas principais tecnologias que foram alvo de estudo, CAB e *Prism*.

Adicionalmente, para além das tecnologias de composição, será também feita uma breve análise de outras tecnologias em que assentam as soluções que foram desenvolvidas.

3.1 Reengenharia de software

Qualquer sistema de software terá que ser alvo de alterações e modificações durante a sua vida útil. Essas alterações podem ser devidas a várias razões, sejam elas de ordem técnica (como a migração para novas plataformas) ou de negócio (como o surgimento de novos requisitos) [Som06]. Além disso, um sistema de software torna-se naturalmente obsoleto com o passar do

tempo, devido ao aparecimento de novas tecnologias e novas tendências em Engenharia de Software [Cle02].

Assim, a reengenharia de software pode ser definida como o processo de estudo, análise e alteração de um sistema de software existente, reconstituindo-o numa nova forma [Ros98].

Apesar de cada processo de reengenharia ser único, podem ser identificados alguns factores que motivam um processo de reengenharia sobre um sistema pré-existente [Dem08]:

- **Modularização** – ao dividir um sistema monolítico e complexo e partes mais pequenas, pode ser mais fácil combiná-las ou comercializá-las em separado;
- **Performance** – um sistema pode ter que sofrer modificações para melhorar a sua performance;
- **Migração para uma nova plataforma** – a arquitectura do sistema pode ter que ser alterada de forma a separar claramente o código que é directamente dependente da plataforma em utilização, facilitando processos de migração;
- **Exploração de novas tecnologias –** a necessidade de adaptação a novos desenvolvimentos tecnológicos motiva frequentemente um processo de reengenharia;
- **Redução das dependências humanas –** um processo de reengenharia pode ter como objectivo a documentação de um sistema, sistematizando o conhecimento sobre o seu funcionamento e simplificando a sua manutenção.

Uma vez que as soluções já existentes são frequentemente fundamentais para o negócio, é impossível abandoná-las abruptamente e substituí-las por um sistema mais actual, pelo que é importante adoptar um processo que permita uma transição gradual [Cle02]. Por isto, torna-se importante o estudo das boas práticas de um processo de reengenharia, para que este possa ser bem sucedido e tenha os resultados desejados.

3.1.1 Processos de reengenharia

Tal como em todas as actividades relacionadas com engenharia de software, é importante modelar os processos envolvidos, permitindo às equipas de desenvolvimento seguir uma aproximação organizada e sistemática [Som06]. Assim, na secção seguinte será apresentado um modelo genérico que procura descrever as actividades relacionadas com os processos de reengenharia de forma genérica. Serão depois analisadas três abordagens possíveis para a reengenharia, terminando com a apresentação de um possível modelo concreto para um processo de reengenharia proposto por Rosenberg.

3.1.1.1 Horseshoe Model

O conceito de reengenharia de software não inclui apenas a transformação do código-fonte. Com efeito, um processo de reengenharia envolve normalmente modificações de um sistema a

todos os níveis [Dem08]. Assim, de acordo com essa filosofia, Kazman, Woods e Carrière [Kaz98] definiram o *Horsehoe Model*.

O modelo, tal como o nome indica, assume a metáfora visual de uma ferradura, sendo apresentado na Figura 1.

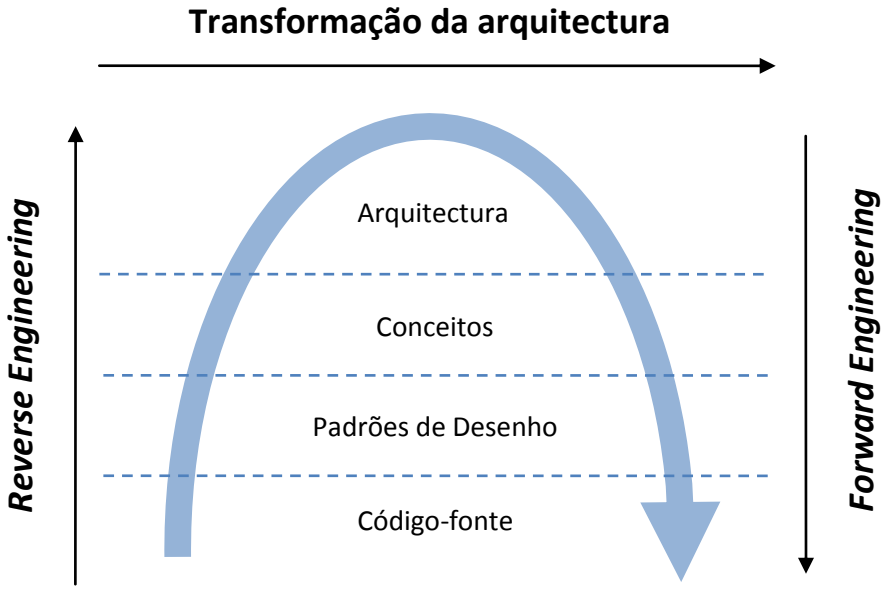


Figura 1 – Representação visual do *Horseshoe Model.*

Segundo a figura, o processo de reengenharia pode ser dividido em três grandes partes: *reverse engineering*[2], transformação da arquitectura e *forward engineering*.

Chikofsky e Cross [Chi92] definem *reverse engineering* como o processo de análise de um sistema, identificando os seus componentes e inter-relações, com o objectivo de criar uma representação do mesmo sistema noutra forma, a um nível mais alto de abstracção. Assim, as actividades de *reverse engineering* estão relacionadas com a compreensão do sistema, assumindo-se como requisito fundamental para qualquer processo de reengenharia. Durante essa fase, a arquitectura actual do sistema é comparada com a arquitectura apresentada na documentação, caso ela exista, e podem ser identificados estilos e padrões de arquitectura já conhecidos que ajudem na análise do sistema. [Kaz98]

A fase de transformação da arquitectura representa a etapa em que a arquitectura do sistema actual, identificada na fase de *reverse engineering*, é alterada e adaptada às novas necessidades. A arquitectura que é definida nesta fase depende muito dos objectivos e requisitos do processo de reengenharia [Kaz98].

O terceiro processo no modelo, *forward engineering*, implica a implementação do sistema segundo a nova arquitectura, partindo dos conceitos de alto nível que resultaram dos processos

[2] O termo pode ser traduzido para Engenharia Reversa. No entanto, uma vez que a designação anglo-saxónica é amplamente usada na literatura, será mantida neste documento.

anteriores. O objectivo final é a implementação concreta do sistema segundo as orientações da nova arquitectura definida no segundo processo.

Tal como é visível na Figura 1, o modelo encontra-se também dividido em quatro níveis verticais que representam diferentes níveis de abstracção, reflectindo a ideia de que o processo de reengenharia não se reduz apenas a modificações de código-fonte. A fase de *reverse engineering* traduz-se por um percurso ascendente através das quatro camadas, enquanto a fase de *forward engineering* realiza o percurso inverso, partindo das abstracções de alto nível para a implementação em código propriamente dito.

3.1.1.2 Abordagens de reengenharia

Rosenberg [Ros98] descreve três abordagens distintas num processo de reengenharia: *big-bang*, incremental e evolutiva.

A abordagem ***big-bang*** consiste apenas na substituição de todo o sistema de uma só vez, tal como é esquematizado na Figura 2.

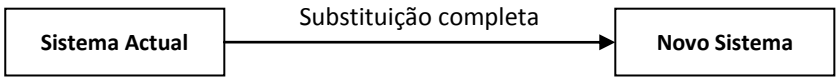


Figura 2 – Esquema da abordagem *big-bang*

É uma abordagem bastante utilizada em migrações para novas tecnologias ou arquitecturas, já que não requer o desenvolvimento de qualquer tipo de interfaces entre o sistema novo e o sistema antigo. Ao descartar completamente o sistema que se deseja substituir, o processo de migração torna-se bastante mais simples, sendo essa a principal vantagem. Por outro lado, este método acarreta riscos bastante elevados, uma vez que se torna complicado garantir que o novo sistema tem o funcionamento desejado. Finalmente, existe uma dificuldade importante: a substituição completa de um sistema implica que exista um período de tempo em que este esteja indisponível, o que nem sempre é possível [Som06].

Na abordagem **incremental** (Figura 3), os vários componentes de um sistema vão sendo substituídos ao longo do tempo, a cada nova versão do sistema.

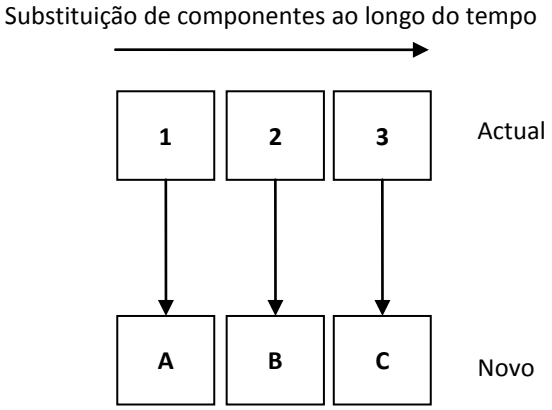


Figura 3 – Esquema da abordagem incremental

Os riscos desta abordagem são menores do que os da abordagem *big-bang*, já que são produzidas versões intermédias que permitem aos clientes identificar potenciais problemas ou falhas no funcionamento do novo sistema. Além disso, a substituição incremental dos componentes facilita o processo de reengenharia, uma vez que este passa a ser focado em componentes específicos e não em todo o sistema em simultâneo. No entanto, existem duas desvantagens dignas de nota: a impossibilidade de alterar a arquitectura básica do sistema (uma vez que apenas os seus componentes são modificados) e o aumento do tempo necessário para completar o processo de reengenharia, devido à produção de versões intermédias.

Finalmente, a abordagem **evolutiva** (Figura 4) é semelhante à iterativa, mas centra-se nas funcionalidades e não na estrutura do sistema.

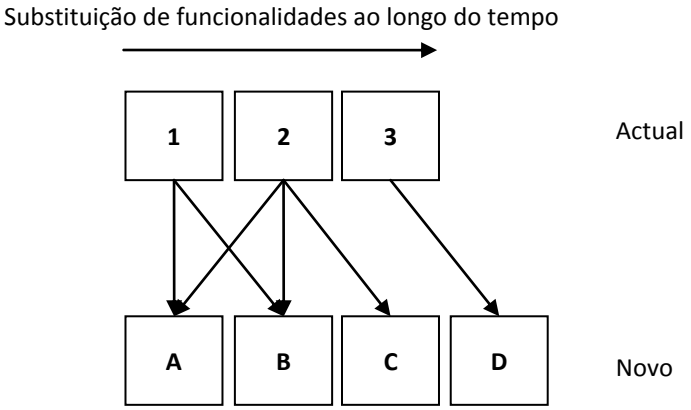


Figura 4 – Esquema da abordagem evolutiva

As partes do sistema original que serão alvo de transformação em cada versão são escolhidas de acordo com as funcionalidades que disponibilizam. Isto permite reduzir as interacções entre os componentes do sistema e resulta, geralmente, em desenhos mais modulares. No entanto, esta abordagem implica um esforço de identificação de funcionalidades relacionadas em todo o sistema.

3.1.1.3 Processo de reengenharia proposto por Rosenberg

Um processo de reengenharia genérico pode ser divido em cinco fases fundamentais [Ros98]:

1. **Constituição da equipa –** a equipa responsável por um projecto de reengenharia será responsável por gerir todo o processo. Deverá ter boas capacidades técnicas e de análise, mas também inter-pessoais, uma vez que será responsável manter os restantes colaboradores informados sobre os novos conceitos introduzidos e, eventualmente, gerir conflitos que surjam da rejeição dos mesmos;
2. **Análise de viabilidade do projecto –** a equipa de reengenharia deve identificar as necessidades que o sistema actual satisfaz e quais os benefícios previstos para o projecto de reengenharia. Os custos do próprio projecto de reengenharia também devem ser

explicitados, de forma a realizar uma análise custo-benefício e determinar se o projecto é viável;

3. **Análise e planeamento –** esta fase pode ser dividida em três actividades distintas: análise do sistema actual (através do estudo da documentação e código fonte), especificação das características do sistema-alvo e criação de uma plataforma de testes e validação;
4. **Implementação e reengenharia –** depois de identificados os objectivos e os resultados esperados, dá-se início às fases descritas no *Horseshoe Model*: *reverse engineering*, transformação da arquitectura e *forward engineering*;
5. **Teste e transição –** à medida que o sistema vai evoluindo, é importante realizar testes que garantam que não foram perdidas funcionalidades durante a reengenharia. Caso os requisitos do novo sistema sejam os mesmos do sistema actual, podem ser utilizados os mesmos casos de teste para validar o novo sistema. A documentação do sistema deve ser actualizada de forma a incluir a informação necessária para a sua operação e manutenção.

3.1.2 Padrões e boas práticas

Em Engenharia de Software, os padrões surgem como uma forma de documentar soluções para problemas recorrentes. Como tal, da mesma forma que os *design patterns* apresentam soluções para problemas de desenho [Gam95], os padrões de reengenharia apresentados por Demeyer, Ducasse e Nierstrasz no livro *Object-Oriented Reengineering Patterns* [Dem08] têm como objectivo capturar conhecimentos sobre as metodologias de modificação de software. São padrões de índole mais abstracta do que os *design patterns*, mas ainda assim oferecem indicações válidas para qualquer processo de reengenharia.

Serão aqui apresentados alguns dos padrões considerados mais relevantes e que foram mais directamente aplicados no projecto. Será seguida a ordem natural de um processo de reengenharia, de acordo com o modelo já apresentado. Assim, os primeiros padrões dizem respeito à familiarização com o sistema alvo e *reverse engineering*, enquanto os últimos já se focam nas problemáticas de implementação e reengenharia do novo sistema.

3.1.2.1 Keep It Simple

Este padrão endereça principalmente a dicotomia entre a flexibilidade e a complexidade de um sistema. Existe frequentemente a tendência para procurar desenhar soluções altamente flexíveis, reutilizáveis e facilmente modificáveis. No entanto, esse esforço implica que se preveja quem utilizará o software e para que propósito, o que nem sempre é possível, fazendo com que o aumento de complexidade traga poucos ou nenhuns benefícios concretos. Além disso, terá como consequência inevitável o aumento da complexidade, o que pode mesmo dificultar a manutenção e as alterações que possam vir a ser necessárias no futuro.

A solução para o dilema, segundo os autores, é implementar sempre a solução mais simples que resolva o problema em questão. Desse modo, a solução será implementada mais rapidamente, permitindo obter *feedback* com a menor brevidade possível.

3.1.2.2 Read All The Code In One Hour

O processo de compreensão do código-fonte de uma aplicação complexa é uma problemática recorrente em praticamente todos os projectos de reengenharia. Para além da dimensão do sistema, a falta de familiaridade com a sua estrutura também é uma dificuldade, já que se torna quase impossível filtrar aquilo que é mais relevante.

O padrão propõe uma sessão de estudo intensivo do código do sistema com a duração aproximada de uma hora, sem interrupções. Recomenda que sejam tomadas notas durante esse intervalo de tempo e que seja produzido um pequeno relatório, informal, no final do processo, incluindo:

- Uma análise geral sobre a viabilidade do projecto de reengenharia;
- As entidades que parecem ser mais importantes;
- Práticas de programação "suspeitas", ou *code smells* [Fow99];
- Partes a investigar posteriormente com mais detalhe.

Os autores também recomendam que dispendido algum tempo na preparação da sessão de estudo intensivo, identificando alguns pontos que pode ser interessante analisar.

A principal vantagem que decorre da utilização deste padrão é a familiarização com os conceitos gerais da solução e da sua arquitectura. Espera-se que uma análise rápida permita absorver os conceitos de alto nível principais, deixando os detalhes de implementação para análises futuras e mais aprofundadas.

3.1.2.3 Step Through The Execution

A simples análise do código fonte de um sistema não é, frequentemente, suficiente para perceber realmente quais os objectos que são instanciados em *run-time* e quais são as interacções entre eles.

A utilização de ferramentas de depuração[3] permite acompanhar a execução de um programa passo a passo, identificando quais os objectos instanciados, quais as operações que são realizadas e qual o estado de cada um em cada momento.

Uma dificuldade importante desta sugestão é a dependência de ferramentas de depuração, já que as funcionalidades que oferecem têm um impacto directo na forma como o sistema pode ser analisado.

[3] *Debuggers*, em inglês.

3.1.2.4 Write Tests To Enable Evolution

Garantir que um novo sistema, obtido como resultado de um processo de reengenharia, mantém a funcionalidade do sistema antigo constitui um dos maiores desafios deste tipo de projectos. O facto de o sistema não ser correcta e completamente compreendido, aliado à falta de documentação e informação sobre as suas dependências, arquitectura e implementação, pode fazer com que sejam introduzidos erros inadvertidamente.

Este padrão apresenta uma solução para este problema, baseada numa prática comum em Engenharia de Software: a realização de testes automatizados e passíveis de serem repetidos. Após cada alteração no sistema, os testes devem ser executados, garantindo que as funcionalidades existentes continuam a funcionar como esperado. O facto de os testes serem automatizados é particularmente relevante, uma vez que é importante que estes possam ser realizados com o mínimo de esforço, sob pena de os programadores hesitarem em utilizá-los como ferramenta de validação.

3.1.2.5 Involve The Users

Os projectos de reengenharia incidem normalmente sobre sistemas que funcionam e realizam a sua função adequadamente. Apesar de serem de difícil manutenção ou tecnologicamente desactualizados, respondem às necessidades dos utilizadores. Os eventuais problemas que possam ter já são conhecidos e os utilizadores sabem como lidar com eles e como contorná-los. Assim, existe frequentemente uma relutância em aceitar um novo sistema, excepto se este lhes facilitar as suas tarefas consideravelmente.

Assim, os autores afirmam que é importante envolver os utilizadores na concepção do sistema, mantendo um contacto próximo com eles durante o desenvolvimento. Isso permitirá perceber as suas necessidades e construir o novo sistema com base nessas informações. Espera-se que isso faça com que o sistema responda às necessidades reais e que seja, por isso, mais facilmente aceite junto dos seus utilizadores finais.

3.1.2.6 Migrate Systems Incrementally

Este padrão recomenda que um projecto de reengenharia siga uma abordagem incremental, evitando os problemas e a complexidade que uma abordagem do tipo *big-bang* acarretaria. O sistema sobre o qual incide o projecto de reengenharia deve ser alterado iterativamente, introduzindo progressivamente as alterações desejadas e mantendo sempre o sistema funcional em cada iteração.

Aliado ao padrão *Involve The Users*, isto permite que seja criado um ambiente de confiança. Os programadores ficam mais confiantes que estão a desenvolver uma solução com valor real e que satisfaz os requisitos, enquanto os utilizadores finais ficam com certezas que o sistema responderá às suas necessidades.

O principal problema desta abordagem reside no facto de nem sempre ser fácil dividir um projecto complexo em partes menores, especialmente se este estiver relacionado com alterações profundas na arquitectura de um sistema.

3.1.2.7 Always Have a Running Version

A reengenharia de um sistema implica muitas vezes a redefinição da sua arquitectura. Esse tipo de mudanças estruturais pode implicar um período de tempo bastante alargado em que não existe uma versão estável e compilada do sistema, sendo essa uma das principais dificuldades da aplicação do padrão anterior.

O padrão recomenda que seja instituída uma regra que obrigue a que todas as alterações sejam integradas diariamente, garantindo que existe sempre uma versão estável e que pode ser utilizada para demonstrar o estado actual do sistema aos utilizadores, facilitando a aplicação do padrão *Involve The Users*.

3.1.2.8 Make a Bridge To The New Town

Por vezes, a migração de um sistema antigo para a sua nova versão implica um período de tempo em que ambos funcionam em simultâneo. Assim, coloca-se o problema de migrar a totalidade dos dados para o novo sistema.

A solução consiste na criação de uma "ponte" entre as duas versões do sistema, responsável por redireccionar os pedidos de acesso a dados para o sistema antigo, realizando todas as conversões necessárias. Esse componente deve ser totalmente transparente, para que o novo sistema não tenha qualquer conhecimento da sua existência e permitindo que deixe de ser utilizado assim que todos os dados forem migrados para o novo sistema. A Figura 5 apresenta um esquema do conceito.

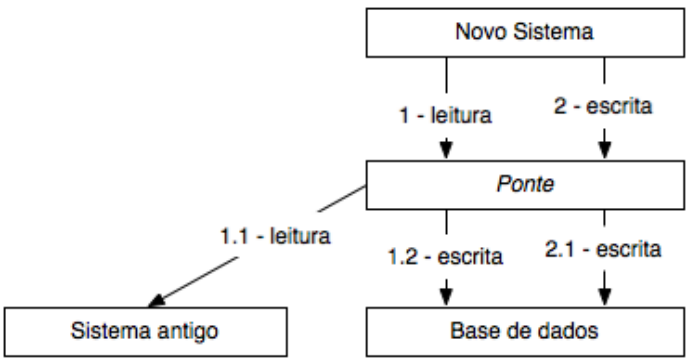


Figura 5 – Estrutura do padrão *Make a Bridge To The New Town*

Tal como é possível ver na figura, durante o processo de migração a "ponte" redirecciona as leituras para o sistema antigo, realizando também uma operação de escrita do valor lido na nova base de dados. Isto garante que os dados vão sendo migrados para o novo sistema. Quando

é feita uma operação de escrita, ela apenas é reflectida na nova versão do sistema, já que é essa que deve ser mantida actualizada.

3.2 Engenharia de Software baseada em Componentes

Após o aparecimento do paradigma de programação orientada a objectos, a maioria dos especialistas consideravam que os objectos passariam a ser as unidades básicas de reutilização de software, chegando mesmo a ser previsto o surgimento de um mercado de objectos reutilizáveis. No entanto, verificou-se que as classes são geralmente demasiado específicas e detalhadas, o que dificulta a sua reutilização. A ideia de comercialização de objectos como unidades de software reutilizável nunca foi adoptada em grande escala [Som06].

Surgiu então o conceito de Engenharia de Software Baseada em Componentes. O tema tem sido extensamente estudado na comunidade científica e tem recebido bastante atenção, principalmente devido ao crescimento em tamanho e complexidade do software. A fácil reutilização de componentes de software complexos permite desenvolver aplicações mais rapidamente e de forma robusta.

Para além dos componentes em si e das tecnologias que permitem construir aplicações com base em componentes reutilizáveis, a CBSE também inclui o estudo dos processos de desenvolvimento recomendados para aplicações desse tipo.

3.2.1 Definições de componente

Vários autores definem um componente de diferentes formas, não existindo uma definição consensual do conceito. Serão aqui apresentadas as várias definições encontradas na literatura.

3.2.1.1 Especificação de UML 1.3

> *"Component: A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files." [Obj99].*

Esta definição foca bastante o conceito de "unidade física", algo que não é referido directamente por outros autores. A definição menciona também as interfaces implementadas pelo componente, mas não faz qualquer referência a dependências.

Além disso, a inclusão de elementos não executáveis é um aspecto que também não é consensual.

3.2.1.2 Grady Booch

> *"A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction." [Gra87]*

A definição de Grady Booch é apresentada num livro sobre a linguagem Ada, mas não deixa de ser uma visão válida e aplicável ao conceito de componentes de software em geral. Não é uma definição muito específica no que diz respeito a dependências externas, focando-se mais no aspecto semântico.

3.2.1.3 Heinemann e Council

> *"A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." [Hei01]*

A definição introduz o conceito de modelo de composição (*component model*), que é clarificado pelos mesmos autores da seguinte forma:

> *"A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model." [Hei01]*

Para além da noção de modelo de composição, a definição de Heinemann e Council refere uma característica importante: a possibilidade de utilização de um componente sem que seja necessária qualquer modificação.

3.2.1.4 Szypersky

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [Cle02]*

Szypersky afirma que a sua definição pode ser dividida em duas partes: uma primeira parte mais técnica, referindo a ideia de interfaces especificadas contratualmente e dependências explícitas, e a segunda parte relacionada com o mercado, abordando a distribuição e utilização independente dos componentes.

3.2.1.5 Sametinger

> *"Reusable software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status." [Sam97]*

A definição de Sametinger tem uma particularidade interessante, ao incluir explicitamente componentes que apenas descrevem funcionalidades. O autor explica o seguinte:

> *"(...) components can have a variety of forms, for example source code, documentation, executable code."[Sam97]*

3.2.1.6 D'Souza e Wills

> *"Component (general) – A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger.*
>
> *Component (in code) – A coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the services it requires from others, and (d) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves." [DSo99]*

D'Souza e Wills apresentam duas definições: a primeira, mais geral, centra-se no desenvolvimento e distribuição de componentes, enquanto a segunda, bastante mais restritiva, se foca em aspecto mais técnicos.

3.2.1.7 Brown e Wallnau

> *"A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces." [Bro96]*

Mais uma vez, a definição refere a independência como uma característica fundamental de um componente.

Das várias definições apresentadas, é possível retirar um conjunto de características básicas que um componente deve ter:

- **Reutilizável** - deve ser possível utilizar um componente em diferentes contextos sem que sejam necessárias modificações;
- **Compositivo** - o componente deve estar de acordo com um modelo de composição, o que possibilita a sua composição com outros componentes que satisfaçam o mesmo modelo;
- **Independente** - o componente deve conter todas as funcionalidades e recursos necessários para ser utilizado sem quaisquer dependências de outros componentes específicos;
- **Documentado** - caso existam dependências específicas, estas devem ser claramente explicitadas. Todas as interfaces externas devem ser especificadas, sintáctica e semanticamente.

3.2.2 Vantagens na reutilização de componentes

A reutilização de componentes apresenta as vantagens que são normalmente associadas à reutilização de software em geral. Sommerville [Som06] identifica as seguintes:

- **Confiabilidade** – A utilização de componentes que já foram testados e utilizados em diversas situações oferece mais garantias de funcionamento de acordo com as especificações, uma vez que os problemas que existiam já foram corrigidos;
- **Redução de riscos** – A implementação de soluções de raiz acarreta sempre custos e riscos que nem sempre são facilmente previsíveis. Por outro lado, a reutilização de componentes já desenvolvidos tem um custo fixo e previsível;
- **Melhor utilização de especialistas** – Em vez de repetirem o mesmo trabalho várias vezes, os especialistas podem desenvolver software reutilizável que encapsule o seu conhecimento;
- **Aplicação de *standards* –** Alguns *standards* da indústria podem ser implementados sob a forma de componentes de software. Assim, a reutilização desses componentes facilita a implementação de acordo com os *standards*;
- **Rapidez de desenvolvimento –** A reutilização de componentes já desenvolvidos permite reduzir os custos e o tempo de desenvolvimento.

3.2.3 Problemas na reutilização de componentes

Apesar de facilitar o desenvolvimento de aplicações complexas, a aplicação dos princípios propostos pela CBSE apresenta ainda alguns problemas. Sommerville [Som06] identificou os seguintes:

- **Credibilidade dos componentes** – Uma vez que os componentes podem ser utilizados de forma independente, são frequentemente distribuídos em formato binário. Não tendo acesso ao código-fonte do componente, os utilizadores não têm forma de garantir que o componente se comporta da forma esperada em todas as situações;
- **Certificação de componentes** – Uma vez que o próprio utilizador não tem possibilidades de verificar o comportamento dos componentes, levanta-se a questão da certificação. Seria necessário que existisse uma entidade independente que pudesse verificar a operação e a segurança dos componentes. No entanto, a forma de implementar um modelo de certificação na prática não é clara. Segundo Sommerville, a única forma viável seria obrigar os componentes a estarem de acordo com uma especificação formal, o que implicaria custos que a indústria não parece estar disposta a suportar;
- **Comportamentos emergentes** – Uma vez que os componentes podem ser distribuídos como "caixas negras", é particularmente difícil prever o comportamento dos mesmos quando integrados num sistema complexo;
- **Compromissos de requisitos** – Os componentes disponíveis não correspondem, normalmente, aos requisitos específicos de uma determinada aplicação. Isso faz com que seja necessário seleccionar os componentes mais adequados a partir de um conjunto limitado. Actualmente, esse é um processo que é feito intuitivamente. Sommerville sugere que é necessário estabelecer um processo mais formal para a escolha de componentes.

3.2.4 Tipos de integração de componentes

A integração de componentes num sistema complexo pode ocorrer a três níveis, que genericamente correspondem às camadas de uma arquitectura comum em três camadas [Dan07]:

- Acesso a dados
- Aplicação (lógica de negócio)
- Apresentação (interface com o utilizador)

3.2.4.1 Integração de dados

A integração de dados provenientes de diferentes fontes pode ser considerada uma forma de integração de componentes. Segundo esta abordagem, uma aplicação composta tem camadas de apresentação e lógica de negócio próprias. A integração dos vários componentes é feita ao nível da camada de acesso a dados. Os diversos repositórios de dados são geridos independentemente por cada componente [Dan07]. Esta arquitectura é esquematizada na Figura 6.

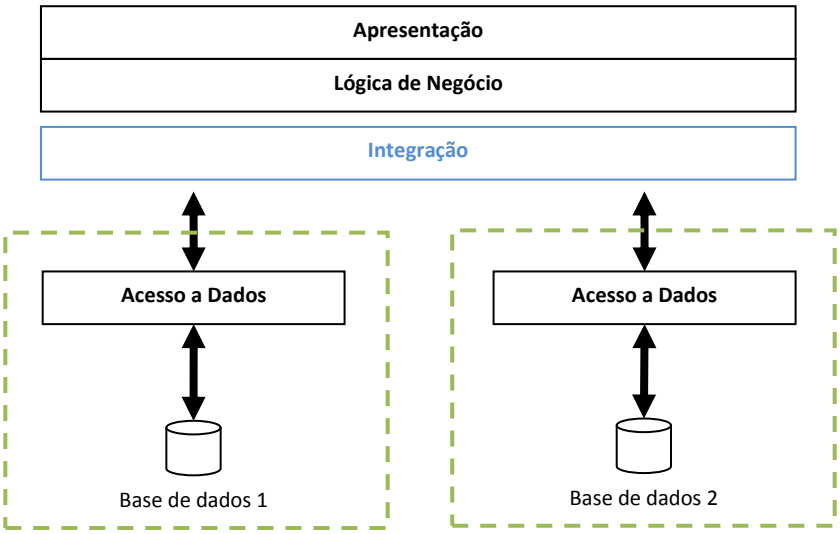


Figura 6 – Arquitectura lógica de uma solução integrando componentes ao nível de dados

A integração de dados é bastante usada porque não requer muita cooperação por parte dos sistemas constituintes. No entanto, esse aspecto traz problemas a vários níveis, uma vez que se torna necessário conhecer profundamente os esquemas de dados dos constituintes e analisar incoerências semânticas [Dan07].

3.2.4.2 Integração ao nível da lógica de negócio

A integração ao nível da lógica de negócio é provavelmente a mais estudada. Neste caso, a aplicação composta tem a sua própria camada de apresentação. A camada de lógica de negócio é desenvolvida, completa ou parcialmente, com base em funcionalidades expostas pelos componentes [Dan07]. A Figura 7 apresenta esta arquitectura.

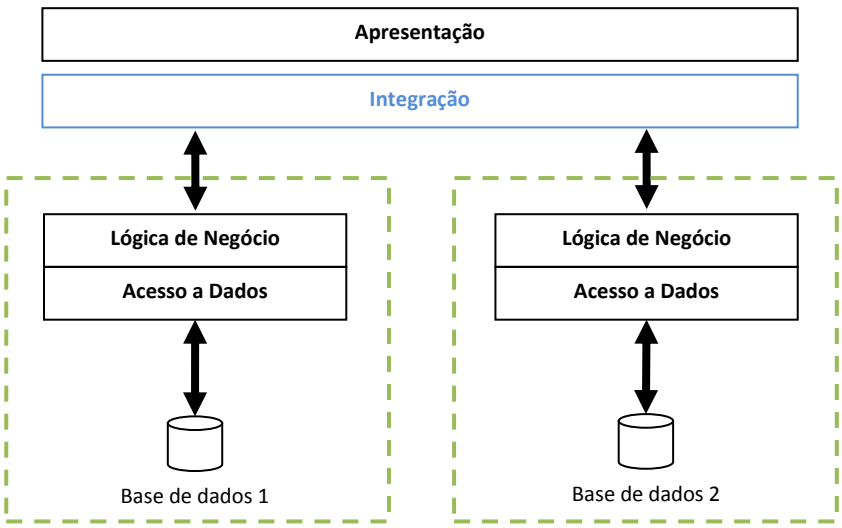


Figura 7 – Arquitectura lógica de uma solução integrando componentes ao nível da lógica de negócio

Ao nível da lógica de negócio, os vários componentes são integrados com recurso a tecnologias como CORBA ou *Web Services*. A arquitectura SOA (*Service-Oriented Architecture*), actualmente bastante divulgada, pode ser considerada como uma arquitectura de componentes distribuídos [Fer05]. Cada serviço expõe uma interface e pode ser utilizado como um componente em qualquer aplicação que assim o deseje, sendo por isso reutilizável. Uma vez que obedecem ao protocolo SOAP para troca de informação, satisfazem um modelo de composição que permite a integração com outros componentes.

3.2.4.3 Integração de componentes de interface com o utilizador

A integração de componentes também pode ser realizada ao nível da interface com o utilizador. Neste caso, a aplicação é composta, pelo menos parcialmente, pelas camadas de apresentação de cada um dos seus componentes [Dan07]. A Figura 8 ilustra este conceito.

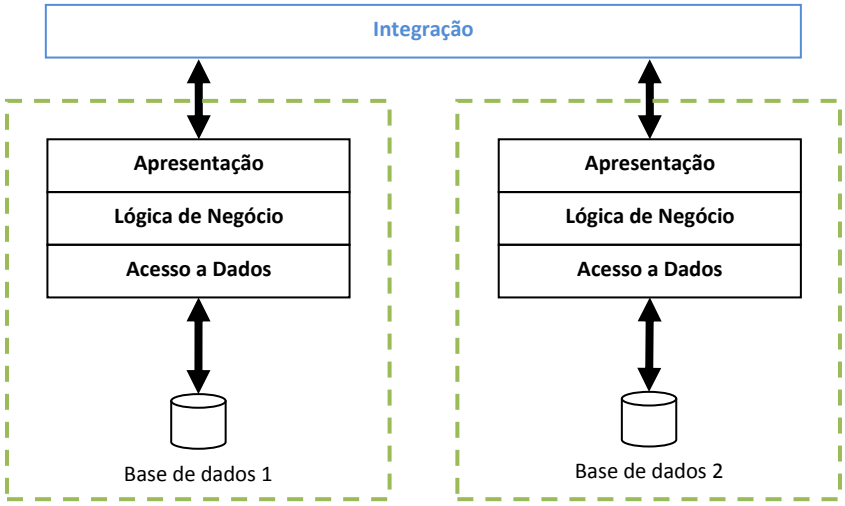


Figura 8 – Arquitectura lógica de uma solução integrando componentes ao nível da interface com o utilizador

No capítulo seguinte serão abordadas várias tecnologias estudadas para a composição de elementos de interface com o utilizador.

3.3 Integração de Componentes Visuais

Existem várias tecnologias que permitem a integração de componentes ao nível da interface com o utilizador, podendo ser divididas em dois grandes grupos: as que permitem a integração de componentes em aplicações *desktop*, e as que estão relacionadas com tecnologias Web.

No que toca a composição de componentes em ambiente Web, a tecnologia *JavaServer Faces* (JSF) é uma das mais reconhecidas, tendo sido inicialmente especificada em 2004 com o objectivo de facilitar o desenvolvimento de interfaces com o utilizador em ambientes J2EE. A

framework inclui mecanismos para a representação e gestão de componentes visuais, incluindo manutenção de estado e tratamento de eventos [Sun09].

Sendo talvez uma das tecnologias de composição mais conhecidas, o *ActiveX* da *Microsoft* é uma tecnologia híbrida que permite a integração de componentes em aplicações *desktop* e em ambiente Web. No entanto, está limitada a ambientes *Windows*. O *ActiveX* surgiu em 1996 como uma evolução do *Component Object Model* (COM) e do *Object Linking and Embedding* (OLE) [Mic091]. Um controlo de *ActiveX* tem liberdade para realizar todo o tipo de operações que um utilizador pode realizar no seu computador, incluindo leitura e escrita de ficheiros e alterações no ficheiro de registo do *Windows* (*registry*). Assim, a utilização de controlos *ActiveX*, principalmente em ambientes Web, levanta actualmente um conjunto de preocupações de segurança, já que constitui frequentemente um meio de distribuição de software malicioso [Moz09].

Relativamente ao ambiente *desktop*, a *Eclipse Rich Client Platform* (RCP) afirma-se como uma plataforma sobre a qual podem ser construídas aplicações compostas, sendo que cada componente é responsável por mostrar a sua própria interface com o utilizador. A RCP oferece uma *shell* sobre a qual a aplicação final será desenvolvida e os componentes são integrados nessa *shell* sob a forma de *plug-ins* [Ecl09].

A *Microsoft* oferece duas soluções para composição de elementos ao nível de interface com o utilizador: CAB e *Prism*. A primeira é focada na integração de componentes desenvolvidos em *Windows Forms*, em ambiente *desktop*, enquanto a segunda, mais recente, já permite a integração de componentes WPF e *Silverlight*[4], suportando assim o desenvolvimento para ambientes Web. Estas duas tecnologias serão apresentadas com mais detalhe nas secções seguintes, uma vez que são as que são actualmente utilizadas na empresa em que foi realizado o projecto e que, por isso, foram alvo de um estudo mais aprofundado.

3.3.1 *Composite UI Application Block* (CAB)

Composite UI Application Block, mais comummente conhecido por CAB, é uma *framework* Microsoft que oferece um modelo de composição de componentes de interface com o utilizador, desenvolvidos em *Windows Forms*, seguindo padrões normalmente utilizados pela indústria para o desenvolvimento de aplicações complexas [Mic08].

A utilização de CAB oferece vantagens em três áreas fundamentais[Gom08]:

- Permite que as aplicações sejam criadas a partir de módulos ou *plug-ins*;
- Permite uma separação efectiva entre os elementos de interface com o utilizador e a lógica de negócio;

[4] *Silverlight* é uma tecnologia desenvolvida para a Web, permitindo a apresentação de conteúdos multimédia interactivos.

- Facilita a reutilização, promovendo interacções *loose coupling* entre componentes, ou seja, os vários módulos não têm referências directas entre eles.

Tal como a maioria das *frameworks*, CAB assenta no princípio de *inversion of control*. Segundo este princípio, a *framework* assume o controlo do fluxo de execução da aplicação, chamando os métodos criados pelo utilizador quando for oportuno. A *framework* passa a funcionar como um "esqueleto extensível", definindo a estrutura e a arquitectura de alto nível da aplicação, sendo o programador responsável por implementar as funcionalidades específicas do sistema que está a desenvolver[Joh88].

Uma aplicação em CAB é constituída por módulos, que são integrados numa *Shell*. Cada módulo é um componente reutilizável e independente que define a sua própria interface com o utilizador, sendo desenvolvido numa *assembly* separada e sem considerações específicas sobre o ambiente em que será inserido. Assim, as comunicações entre módulos são normalmente realizadas através de eventos.

Um dos conceitos fundamentais na arquitectura definida pelo CAB é o *WorkItem*. Um *WorkItem* é um *container* de todos os elementos necessários para a implementação de um caso de utilização. Cada *WorkItem* pode também conter outros *WorkItems*, criando uma hierarquia que é partilhada por toda a aplicação. Existe um *WorkItem* que é disponibilizado pela *Shell* e que contém todos os que são definidos nos módulos, actuando como a raiz da árvore.

A interface com o utilizador de cada módulo é definida através de um *UserControl* de *Windows Forms*, constituindo uma *view*. As *views* podem ser injectadas em *Workspaces*, que são controlos disponibilizados pela *framework*. Assim, a *Shell* será um formulário que contém um ou mais *Workspaces*, onde cada módulo poderá injectar as suas *views.*

O desenvolvimento de aplicações em CAB assenta essencialmente num conjunto de *design patterns*:

- **Model-View-Controller**
- **Model-View-Presenter**
- **View Navigation**
- ***Dependency Injection***

A *framework* oferece ainda muitas outras funcionalidades que servem de suporte ao desenvolvimento de aplicações compostas. Estas serão apresentadas com mais detalhe no capítulo 4, integradas na descrição do processo de migração de CAB para *Prism*.

3.3.1.1 Model-View-Controller

Model-View-Controller (MVC) é um padrão de arquitectura de software que tem como principal objectivo a separação entre a lógica de negócio e a apresentação da informação, oferecendo assim uma maior flexibilidade. A interface com o utilizador pode assim ser

desenvolvida, testada e, se necessário, substituída, independentemente da lógica funcional da aplicação.

A aplicação é dividida em três partes: modelo (*Model*), vistas (*views*) e *controllers* [Bus96]. A Figura 9 apresenta a estrutura geral descrita pelo padrão.

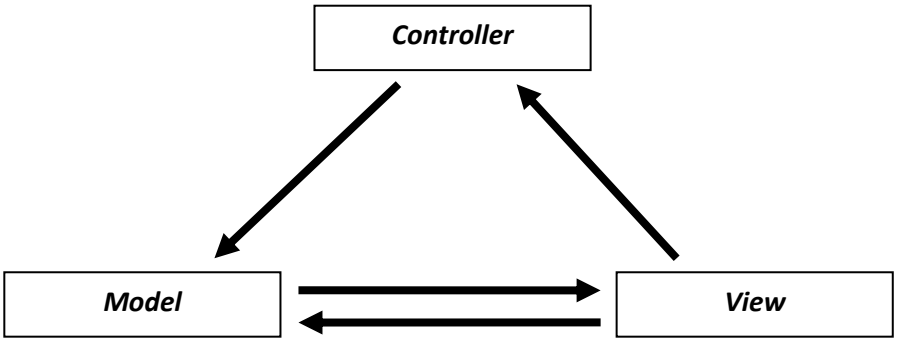


Figura 9 – Estrutura do padrão Model-View-Controller

O modelo tem como principal responsabilidade a gestão dos dados, devendo disponibilizar métodos para consultar e alterar os dados da aplicação. A forma e o local em que devem ser guardados os dados também são geridos pelo modelo. O modelo é ainda responsável por notificar as vistas e os *controllers* quando os dados são alterados, garantindo a consistência dos dados apresentados em toda a aplicação. Normalmente esse mecanismo é implementado com recurso ao padrão *Observer*[Gam95].

As *views* têm como função apresentar os dados ao utilizador, podendo existir várias *views* diferentes sobre a mesma informação. Quando recebe uma notificação do modelo indicando que houve uma alteração dos dados, a *view* deve proceder à sua actualização.

Os *controllers* estão associados unicamente a uma *view*, isto é, cada *view* tem o seu próprio c*ontroller* e cada *controller* só está ligado a uma *view*. O *controller* define a lógica de interacção de uma *view*, respondendo aos eventos lançados pela interacção com o utilizador e chamando os métodos disponibilizados pelo modelo para proceder a alterações nos dados.

3.3.1.2 Model-View-Presenter

O padrão *Model-View-Presenter* (MVP) constitui uma variação ao MVC, tendo por isso o mesmo objectivo de separação da lógica da negócio, apresentação e lógica de apresentação. A distinção entre ambos os padrões nem sempre é clara, já que existem várias interpretações normalmente aceites sob o nome de *Model-View-Presenter*. No entanto, será focada a interpretação dada pela *Microsoft* e que é recomendada para a utilização em CAB.

Apesar de existir uma correspondência entre o conceito de *controller* em MVC e *presenter* em MVP, o *presenter* tem algumas responsabilidades adicionais. Em MVP, a *view* deixa de ter uma ligação directa ao modelo, uma vez que o *presenter* passa a ser responsável pelo fornecimento de dados à *view* [Mic08]. Essa é a principal diferença entre os dois padrões, tornando a *view* menos dependente do modelo. A estrutura definida pelo MVP é apresentada na Figura 10.

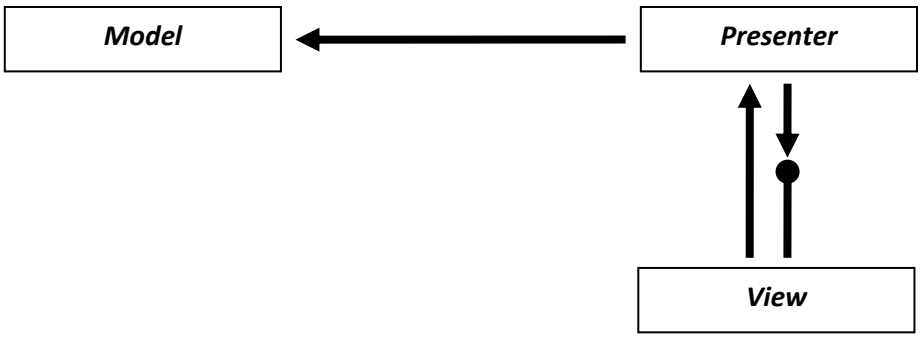


Figura 10 – Estrutura do padrão *Model-View-Presenter*

A comunicação entre o *presenter* e a *view* passa também a ser feita através de uma interface, facilitando a substituição da *view* e permitindo testar a lógica de negócio independentemente da interface com o utilizador [Mic08].

3.3.1.3 View Navigation

Num aplicação composta por várias *views*, torna-se necessária a existência de um mecanismo que permita efectuar alterações numa *view* em resposta a acções realizadas noutra. A *Microsoft* apresenta uma solução para este problema sob a forma de um padrão denominado *View Navigation* (Figura 11).

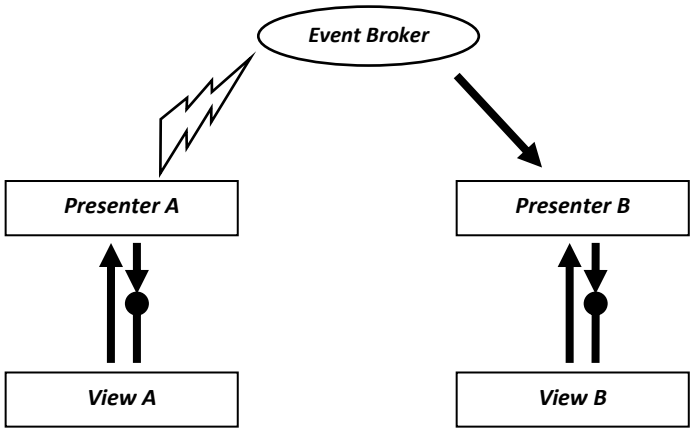


Figura 11 – Modelo de comunicação descrito pelo padrão *View Navigation*

Segundo este padrão, sempre que uma *view* deseje despoletar uma acção noutra *view*, o seu *presenter* deve lançar um evento que será subscrito pelo *presenter* da *view* que será modificada. A comunicação ocorre apenas entre os *presenters*, sem que seja necessário manterem uma referência directa entre eles.

Conceptualmente, o *presenter* A não realiza qualquer acção sobre o *presenter* B. Em vez disso, ele é notificado que um dado evento ocorreu no *presenter* A, podendo assim desencadear as acções necessárias na *view* que lhe está associada.

Na Figura 11, o *presenter* A lança um evento que é subscrito pelo *presenter* B. Os eventos em CAB são geridos por um *Event Broker* e são identificados por um nome, tornando possível a subscrição e publicação de eventos sem que haja referências directas entre os objectos.

3.3.1.4 Dependency Injection

O padrão *dependency injection* descreve a forma como uma classe pode obter uma referência para outra classe, da qual depende. Contrastando com a situação mais usual, em que cada classe é responsável por obter as referências de que necessita, o padrão recomenda que a lógica de instanciação das dependências seja delegada num objecto independente [Fow04]. Esse objecto é denominado *container*, já que deve guardar referências para todos os objectos que podem ser injectados.

No caso concreto de CAB, é o *WorkItem* que actua como *container*. Sempre que um novo objecto é adicionado à colecção *Items* de um *WorkItem*, passa a estar disponível para ser injectado noutros objectos. Além disso, a criação do novo objecto é feita pelo próprio *WorkItem*, permitindo injectar-lhe todas as dependências de que necessita. A Figura 12 apresenta este conceito [Mic082].

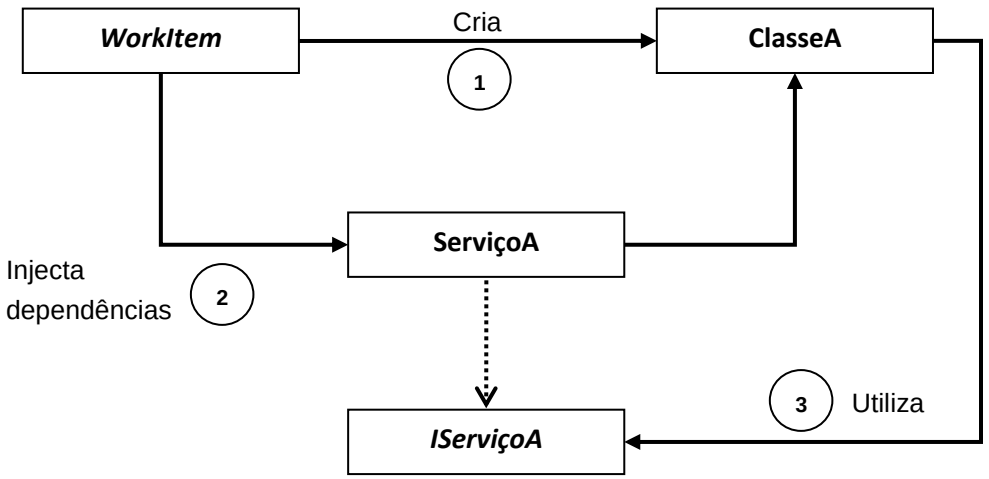


Figura 12 – *Dependency Injection* em CAB

Assume-se que a `ClasseA` declarou que tem uma dependência de um objecto que implemente a interface `IServiçoA`. Assim, quando o *WorkItem* cria um novo objecto dessa classe,

injecta-lhe automaticamente uma referência para um objecto conhecido que implemente a interface desejada.

A principal vantagem que advém da utilização de *dependency injection* reside na maior facilidade com que pode ser substituída a implementação de uma dada interface, sem ter que alterar todas as classes que dela dependem.

3.3.1.5 Smart Client Software Factory

A *Smart Client Software Factory* (SCSF) é um conjunto de extensões para o *Visual Studio 2008* que encapsula alguns dos padrões e melhores práticas recomendadas pela *Microsoft* para a construção de aplicações em CAB [Mic081]. Através da integração com o ambiente de desenvolvimento é possível automatizar a criação dos vários elementos de uma solução, seguindo uma arquitectura pré-definida. Isto facilita a criação de soluções robustas, modulares e reutilizáveis, libertando os programadores para as tarefas essenciais à lógica de negócio da aplicação que desenvolvem.

3.3.1.6 Smart Client Contrib

O projecto *Smart Client Contrib* [Cod08] é uma biblioteca *open-source* de extensões a CAB e a SCSF, permitindo o desenvolvimento em WPF. Na sua versão 1.5, suporta *shells* e *views* em WPF.

A biblioteca disponibiliza novos controlos com as mesmas funcionalidades dos *Workspaces* de CAB, com a excepção de permitirem injectar *views* em WPF.

3.3.2 *Composite Application Library* (*Prism*)

A *Composite Application Library* é normalmente conhecida pelo seu nome de código, utilizado durante a sua fase de desenvolvimento - *Prism*. A primeira versão foi lançada em Junho de 2008, suportando apenas o desenvolvimento de componentes em *Windows Presentation Foundation* (WPF). Em Fevereiro de 2009 foi lançada a segunda versão, introduzindo o suporte para *Silverlight*.

Tal com CAB, *Prism* é uma *framework* que pretende facilitar o desenvolvimento de aplicações complexas baseadas em componentes que incluem a sua própria interface com o utilizador, integrados numa *Shell* desenvolvida em WPF ou *Silverlight*. A *framework* também providencia as funcionalidades necessárias para garantir a comunicação entre os diversos componentes.

Incorporando o *feedback* recolhido com CAB, a nova *framework* procura ser mais leve, mais simples e menos restritiva. Muitas das funcionalidades disponibilizadas por CAB deixaram de existir em *Prism*, passando a ser responsabilidade do programador. Isto permite uma curva de aprendizagem menos acentuada, já que a *framework* é menos complexa. Permite ainda uma

adaptação mais fácil às necessidades específicas de cada solução, uma vez que não força a utilização dos mecanismos oferecidos pela *framework*.

Um dos principais passos dados para esse feito foi a remoção do *WorkItem*. Em CAB, o *WorkItem* concentra várias funções, actuando como *container* para *dependency injection* e como *controller*, coordenando a acção de várias *views*. Ao eliminar o *WorkItem*, essas responsabilidades ficam distribuídas por várias classes. *Prism* não inclui um *container* próprio para *dependency injection*, permitindo ao programador utilizar o *container* da sua preferência[5].

Outra optimização importante foi a tentativa de minimizar a utilização de *reflection* por parte da *framework*. Como será visível no próximo capítulo, CAB recorre frequentemente a atributos para várias das funcionalidades que disponibiliza. Apesar de simplificar a compreensão do código, isso implica que a *framework* seja obrigada a recorrer a *reflection* para analisar quais os métodos e propriedades uma classe estão marcados com atributos. Uma vez que a utilização de *reflection* em *C#* é inerentemente lenta [Pob05], em *Prism* isso é evitado, contribuindo para um melhor desempenho.

A injecção de *views* em *Prism* tira partido do *content model* bastante flexível de WPF. Uma vez que é possível incluir um ou mais elementos dentro de qualquer controlo, já não é necessário um controlo específico onde serão injectadas as *views*. Em *Prism*, utiliza-se o conceito de *region*. Cada *region* não é mais do que um controlo de WPF normal, ao qual foi associado um nome. Posteriormente, é possível aceder a essa *region* utilizando o *RegionManager* e injectar-lhe *views*.

3.4 Plataforma tecnológica

Nesta secção pretende-se apresentar uma visão geral sobre algumas das tecnologias mais relevantes para o projecto, para além de CAB e *Prism*, que já foram apresentadas acima. Serão apresentadas as duas tecnologias de apresentação focadas ao longo de todo o projecto, *Windows Forms* e WPF, a plataforma de base de dados utilizada (*Oracle 10g*) e as ferramentas de geração de código *Codesmith*.

3.4.1 *Windows Forms*

A tecnologia *Windows Forms* foi introduzida em 2002, com o objectivo de facilitar o desenvolvimento de interfaces gráficas em ambientes *Windows* [Nat07]. Permite criar formulários utilizando os elementos gráficos nativos de sistemas operativos *Windows*, evitando a utilização da API *Win32*, aumentando a produtividade.

Os principais aspectos que *Windows Forms* pretende endereçar são os seguintes [Pan07]:

[5] A *framework* é distribuída com suporte pré-configurado para o *Unity Container* da Microsoft.

- Trazer a facilidade e robustez do desenvolvimento Web para o ambiente *desktop*;
- Facilitar o desenvolvimento de aplicações com aspecto e comportamento profissionais;
- Simplificar o desenvolvimento de aplicações;

Em *Windows Forms*, os formulários são desenhados de forma interactiva, utilizando o *Microsoft Visual Studio*, arrastando e largando os vários controlos que se desejam incluir na aplicação. Quando o utilizador final interage com um controlo, é lançado um evento, que pode ser tratado em *code-behind*[6].

Esta tecnologia assenta sobre a *framework* .NET da *Microsoft*, pelo que as aplicações podem ser desenvolvidas em qualquer linguagem suportada por essa plataforma (*C#*, *C++*, *J#*, *Visual Basic .NET*, entre outras).

3.4.2 *Windows Presentation Foundation* (WPF)

A tecnologia *Windows Presentation Foundation* (WPF) surge em 2001, sendo anunciada como a tecnologia de apresentação por excelência para aplicações *Windows*, oferecendo funcionalidades que permitem criar experiências mais ricas e apelativas do ponto de vista de interacção com o utilizador. As principais características de WPF são as seguintes [Nat07]:

- **Integração –** WPF oferece um modelo de programação integrado que suporta funcionalidades como gráficos 3D, vídeo ou reconhecimento de voz, poupando ao programador o esforço que anteriormente era necessário para utilizar essas funcionalidades numa solução;
- **Independência de resolução de ecrã –** As interfaces desenvolvidas em WPF são desenhadas como gráficos vectoriais, tornando-se assim independentes da resolução de ecrã que é utilizada;
- **Aceleração por *hardware* –** Todos os controlos presentes numa aplicação WPF são desenhados com aceleração por *hardware*, beneficiando das capacidades gráficas das máquinas que a executam, com todos os benefícios de performance que daí resultam;
- **Programação declarativa –** As interfaces em WPF são criadas a partir de ficheiros com uma estrutura baseada em XML, denominada XAML (*Extensible Application Markup Language*). O paradigma é semelhante ao que é empregado na Web, utilizando ficheiros HTML para definir o *layout* da aplicação. Cada ficheiro XAML possui um ficheiro de *code-behind* associado, onde pode ser implementada toda a lógica de controlo necessária;
- **Composição e personalização –** Em WPF é possível modificar o aspecto de qualquer controlo, combinando-o com outros controlos. Assim, é possível, por exemplo, criar um

[6] Terminologia normalmente utilizada pela *Microsoft* para designar o código-fonte que está associado a cada formulário.

botão com um vídeo embebido, em vez de texto, como é comum. Além disso, através da utilização de estilos, é possível definir várias aparências distintas para a mesma aplicação, um pouco à imagem do que acontece com as folhas de estilos CSS, na Web;

- **Fácil distribuição –** Para além das opções de distribuição de aplicações tradicionais, como *Windows Installer* ou *ClickOnce*[7], WPF suporta a integração directa de aplicações num *browser*, facilitando a sua distribuição.

3.4.3 *Oracle Database 10g*

A versão original do sistema de gestão de bases de dados relacionais (SGBDr) *Oracle* foi criada em 1979 e desde então tem tido uma posição dominante no mercado.

As bases de dados *Oracle* suportam uma vasta gama de sistemas operativos e arquitecturas, podendo tirar partido de múltiplos processadores e arquitecturas 64bit [Gre01].

A versão mais recente é a 11g, lançada em Maio de 2008. No entanto, na *GlinttHS* ainda é utilizada a versão 10g (*Release 2)*.

3.4.4 *Codesmith*

Codesmith é um conjunto de ferramentas que permitem gerar código com base em *templates*. Os *templates* são definidos em ficheiros com uma sintaxe muito semelhante a ASP.NET e permitem gerar ficheiros de texto em qualquer linguagem de programação.

A utilização de ferramentas de geração de código permite reduzir o tempo gasto em tarefas de codificação repetitivas, reduzindo o risco de erros e produzindo código consistente.

A *GlinttHS* possui um conjunto de *templates* já desenvolvidos, aliados a uma aplicação, que permitem gerar automaticamente o código necessário para criar as entidades (classes) necessárias a partir da estrutura de uma base de dados. Para além de gerar as entidades, o gerador também tem a capacidade de gerar o código necessário para as operações mais simples, como *selects, inserts, updates* e *deletes*. São gerados os procedimentos necessários para bases de dados *Oracle*, os métodos que os invocam em *C#* e os *Web Services* que permitem às aplicações executar essas operações.

Assim, o gerador permite que seja criada automaticamente uma estrutura básica para uma camada de acesso a dados e lógica de negócio.

[7] *ClickOnce* é uma tecnologia de distribuição de aplicações desenvolvidas em *.NET* que tem como objectivo principal facilitar a instalação e a actualização das aplicações.

3.5 Conclusões

Neste capítulo foram apresentados os principais conceitos teóricos que serviram de base para o desenvolvimento do trabalho. As noções de reengenharia de software foram essenciais para determinar a melhor abordagem na estruturação de uma infra-estrutura de migração eficaz, enquanto o estudo de engenharia de software baseada em componentes se revelou importante para a compreensão dos conceitos subjacentes às tecnologias de composição abordadas.

Relativamente às tecnologias de apresentação, após a análise dos benefícios oferecidos por WPF, a decisão da empresa em migrar todas as suas soluções para WPF revela-se a mais indicada, principalmente devido à maior flexibilidade oferecida por essa tecnologia. Assim, uma vez que CAB é uma *framework* direccionada principalmente a *Windows Forms*, e dadas as melhorias introduzidas por *Prism,* a migração para esta última torna-se inevitável, justificando assim o projecto de reengenharia que aqui se apresenta.

O sistema de bases de dados *Oracle 10g* foi utilizado para o desenvolvimento do módulo de definição de protocolos de tratamento oncológico, uma vez que é utilizado pela aplicação de Processo Clínico, onde o módulo desenvolvido se integra.

A utilização do sistema de geração de código desenvolvido pela empresa, baseado em *Codesmith*, justifica-se, já que facilita bastante a criação das entidades e operações correspondentes à estrutura de dados definida na base de dados. Além disso, ao utilizar os *templates* criados pela empresa, garante-se que o código gerado estará de acordo com as práticas internas da *GlinttHS*.

Capítulo 4

Processo de Migração de CAB para *Prism*

Neste capítulo será descrito o trabalho que foi realizado com o objectivo de definir um processo de migração entre CAB e *Prism*, que resultou no desenvolvimento de uma infra-estrutura de migração que permite realizá-la de forma incremental.

O capítulo está organizado de forma a reflectir a sequência do trabalho realizado. Inicialmente, são identificadas as principais diferenças entre as tecnologias, passando depois para uma análise mais detalhada da infra-estrutura desenvolvida. São também apresentados os testes de desempenho realizados e, finalmente, apresentados os passos necessários para a migração de um pequeno exemplo.

É importante referir, tal como já foi apontado no primeiro capítulo, que o trabalho aqui apresentado foi realizado em colaboração com José Carvalho, que também realizou o seu projecto na mesma empresa.

4.1 Identificação das diferenças entre as duas tecnologias

Tal como já foi referido acima, CAB e *Prism* são duas tecnologias que endereçam a mesma problemática e que seguem, em termos gerais, os mesmos princípios. Assim, o primeiro passo para facilitar a migração de CAB para *Prism* foi a identificação dos principais mecanismos que teriam que ser modificados, fazendo o paralelismo entre as acções necessárias numa e noutra tecnologia.

Existem alguns mecanismos de CAB que não têm correspondência em *Prism*, passando a ser responsabilidade do programador. Para esses casos, é sugerida uma possível implementação que facilite a migração das aplicações já existentes.

4.1.1 Desaparecimento do conceito de *WorkItem*

Em CAB o *WorkItem* é visto como o elemento responsável por um caso de utilização, actuando como *container* para *dependency injection* e como *controller*, coordenando a acção das várias *views*. Em *Prism* esse conceito desapareceu, ficando o programador responsável por criar as classes que considere necessárias para conter a lógica da aplicação. As funções de *dependency injection* passam a ser garantidas por um *container* independente.

De forma a garantir que as soluções já desenvolvidas possam ser migradas sem grandes alterações do ponto de vista de arquitectura, torna-se necessário definir uma classe — que será denominada *Controller*[8] — que concentre em si as várias responsabilidades de um *WorkItem* em CAB, passando a existir uma correspondência directa entre o *WorkItem* em CAB e a classe *Controller* em *Prism*. Assim, num processo de migração, as classes que derivam de *WorkItem* em CAB devem passar a derivar da classe *Controller*.

4.1.2 Hierarquia de *WorkItems*

Em CAB existe um *WorkItem* do qual todos os outros descendem, normalmente denominado *Root WorkItem*. Cada *WorkItem*, por sua vez, pode ter outros *WorkItems* que descendem de si, criando efectivamente uma estrutura em árvore.

Uma vez que em *Prism* o conceito de *WorkItem* desapareceu, torna-se imperativo criar um mecanismo que permita simular o funcionamento da hierarquia em CAB. Caso isso não acontecesse, toda a arquitectura das soluções desenvolvidas teria que ser alterada, levando a um processo de migração bastante moroso.

A solução encontrada é baseada nas recomendações da *Microsoft* [Mic09] e tem por base a capacidade de um *container* em *Prism* descender de outro *container*[9], tal como acontece com os *WorkItems* em CAB. A hierarquia passa a ser garantida pelos *containers*, sendo que em cada nível da árvore existe um *Controller* (que corresponde a um *WorkItem* em CAB) associado (Figura 13).

[8] A denominação *Controller* é apenas indicativa, uma vez que é uma classe criada pelo programador, independente da *framework*.

[9] Assume-se que será utilizado o *Unity Container*. No entanto, todos os princípios serão igualmente aplicáveis caso seja utilizado outro *container*, desde que este suporte hierarquias.

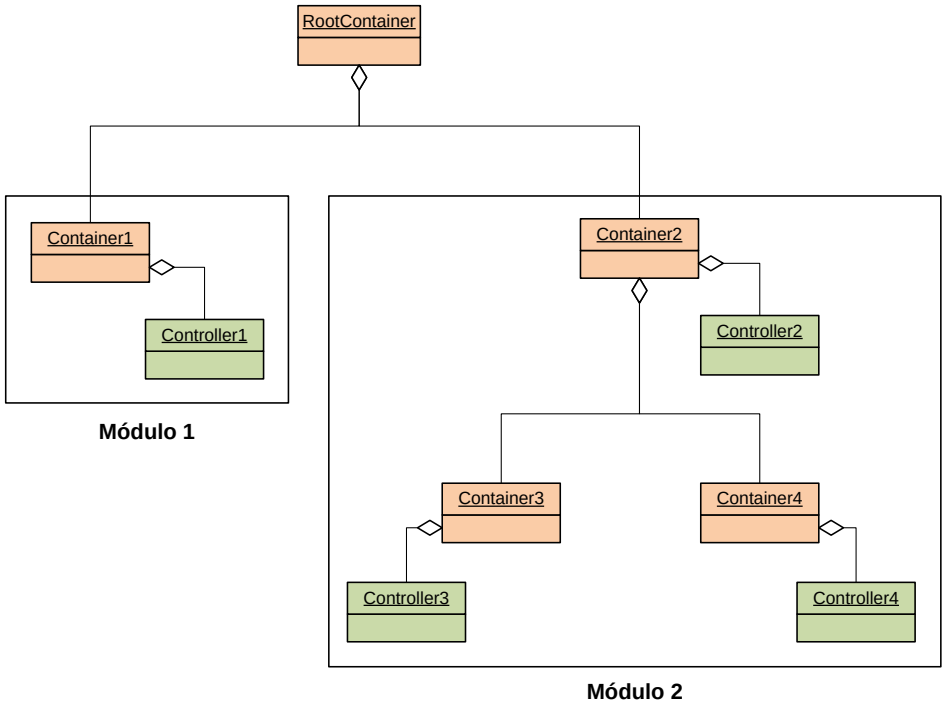


Figura 13 – Simulação da hierarquia de WorkItems utilizando containers

Cada *container* tem acesso ao seu pai e aos seus filhos, podendo assim navegar pela árvore tal como era possível em CAB.

4.1.3 Inicialização de módulos

Cada módulo de CAB deve ter uma classe que descende de *ModuleInit* e que é responsável pela inicialização do mesmo. Em *Prism*, essa classe deixa de derivar de *ModuleInit*, passando a implementar a interface *IModule*.

A principal diferença está no facto de em *Prism* não existir um *Root WorkItem*, pelo que a forma de instanciar e registar novos *WorkItems* (*Controllers* no caso de *Prism*) será necessariamente diferente (Tabela 1).

Tabela 1 - Instanciação e registo de WorkItems

CAB	`MyWorkItem myWorkItem = parentWorkItem.WorkItems.AddNew<MyWorkItem>();` `myWorkItem.Run("tabWorkspace1");`

Prism	`IUnityContainer child = parent.CreateChildContainer();`
	`parent.RegisterInstance<IUnityContainer>(child);`
	`MyController workItem = newMyController(regionManager, child);`
	`child.RegisterInstance<MyController>(workItem);`
	`workItem.Run("regionName");`

Em CAB basta adicionar um novo *WorkItem* ao *Root WorkItem* e executar o método *Run* para o lançar.

Em *Prism* é um pouco mais complexo, uma vez que a estrutura em árvore tem que ser construída recorrendo aos *containers*. Após ter sido criado um *container* "filho", ele é registado no *container* "pai". O novo *Controller* é então criado e registado nesse *container* "filho", antes de ser lançado através do método *Run*.

O código ilustrado na Tabela 1 assume que se dispõe de uma referência para o *Workitem* "pai", no caso de CAB, e para o *Container* "pai" e *RegionManager* no caso de *Prism*. Em ambos os casos, essas referências podem ser obtidas por *dependency injection*.

4.1.4 Definição e injecção de *views*

Tanto em CAB como em *Prism*, as *views* são apenas *user controls*, não existindo diferenças significativas entre as duas tecnologias. Normalmente, a única alteração que é necessário efectuar numa *view* para migrar de CAB para *Prism* é alterar a forma como são declaradas as propriedades que serão alvo de *dependency injection*.

No que diz respeito à injecção de *views*, é necessária cautela, dependendo da tecnologia em que foram desenvolvidas (*Windows Forms* ou WPF). A Tabela 2 ilustra o suporte de CAB e *Prism* para as duas tecnologias em que podem ser desenvolvidas as *views*.

Tabela 2 – Suporte das frameworks com e sem extensões para Windows Forms e WPF.

	Windows Forms	**WPF**
CAB	✓	✗
CAB com extensões *SmartClient Contrib*	✓	✓
Prism	✗	✓
Prism utilizando *WindowsFormsHost*	✓	✓

Apesar de não ser originalmente suportado por CAB, a extensão *SmartClient Contrib* permite a injecção de *views* desenvolvidas em WPF (cf. 3.3.1.6).

Relativamente a *Prism*, não existe suporte explícito para *views* em *Windows Forms*. No entanto, existe um controlo de WPF — *WindowsFormsHost* — que possibilita a inclusão de um controlo *Windows Forms* em WPF. Uma vez que as *views* são efectivamente controlos *Windows Forms*, é possível incluí-los num *wrapper* WPF que será depois injectado em *Prism*, como qualquer outra *view* WPF (Figura 14).

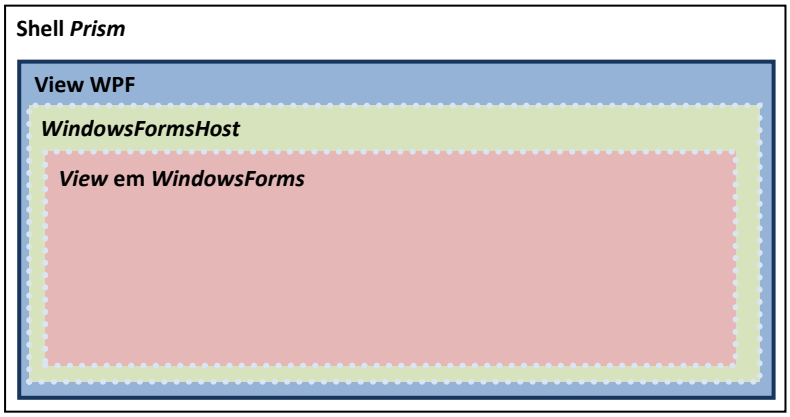


Figura 14 – utilização de um wrapper WPF para injectar uma view em Windows Forms numa shell Prism

De um modo genérico, injecção de *views* em CAB recorre a *Workspaces*, enquanto em *Prism* recorre a regiões. A Tabela 3 ilustra a forma como uma *view* é injectada em CAB e em *Prism*.

Tabela 3 – Injecção de views em CAB e em Prism

CAB	`MyView view = workItem.Items.AddNew<MyView>();` `workItem.Workspaces["workspaceName"].Show(view);`
Prism	`MyView view = Container.Resolve<TView>();` `RegionManager.Regions["regionName"].Add(view);`

O *RegionManager* é uma classe de *Prism*, responsável pela gestão das regiões e injecção de *views*. É possível obter uma referência para o *RegionManager* através de *dependency injection*.

4.1.5 Variáveis de estado

Um *WorkItem* em CAB tem a capacidade de guardar estado através de um dicionário, associando objectos a *strings*. Essas variáveis de estado podem ser utilizadas para preencher propriedades em objectos criados a partir do *WorkItem*, bastando para isso adicionar o atributo *State* à propriedade em que se deseje injectar o valor de uma variável de estado.

Mais uma vez, essa funcionalidade não existe em *Prism*, sendo simulada pela classe *Controller* referida acima. A Tabela 4 exemplifica o acesso a variáveis de estado em CAB e em *Prism*.

Tabela 4 – Acesso a variáveis de estado em CAB e em Prism

CAB	`//Escrita`
	`workItem.State[“key”] = anObject;`
	
	`//Leitura`
	`object stateVariable = workItem.State[“key”];`

Prism	`//Escrita`
	`controller.setState(“key”) = anObject;`
	
	`//Leitura`
	`object stateVariable = controller.getState(“key”);`

Em *Prism*, a classe *Controller* deverá conter um dicionário semelhante ao de CAB, permitindo guardar o estado. O acesso é realizado através de dois métodos, *getState* e *setState*.

A possibilidade de injecção de estado oferecida por CAB pode ser simulada em *Prism* através da criação de um atributo a aplicar às propriedades que se deseja que sejam preenchidas. Além disso, a classe *Controller* deve disponibilizar uma função que receba um objecto onde será injectado o estado nas propriedades marcadas com o novo atributo. Essa função deve ser chamada durante a criação do novo objecto. A Tabela 5 ilustra o processo de injecção de variáveis de estado em CAB e em *Prism*.

Tabela 5 – Injecção de estado em CAB e em Prism

CAB	`[State("mensagem")]` `public string StateMsg { get; set; }`

Prism	`[NewState("mensagem")]`
	`public string StateMsg { get; set; }`
	
	`//Ao criar o objecto...`
	`MyView view = new MyView();`
	`controller.InjectMyState(view);`

Em CAB, a propriedade marcada com o atributo *State* é automaticamente preenchida com o valor actual da variável de estado durante a criação do objecto. Uma vez que esse mecanismo não existe em *Prism*, é necessário chamar um método que realize essa operação. No exemplo acima, a classe *Controller* disponibiliza o método *InjectMyState*, que percorre o objecto que recebe como argumento e determina, utilizando *reflection*, quais as propriedades que devem ser preenchidas.

Uma outra funcionalidade existente em CAB e que tem que ser implementada manualmente em *Prism* é a notificação de alterações nas variáveis de estado. Em CAB é possível associar o atributo *StateChanged* a um método, fazendo com que ele seja executado sempre que uma determinada variável de estado seja alterada. Para simular este comportamento, a classe *Con-*

troller deve lançar um evento sempre que o método *setState* é chamado (recorde-se que este método permite alterar uma variável de estado). Esse evento terá que ser explicitamente subscrito (Tabela 6).

Tabela 6 – Subscrição do evento StateChanged em CAB e em Prism

CAB	`[StateChanged("key")]`
	`public void MyKey_StateChanged(object sender, StateChangedEventArgs args)`
	`{`
	`//...`
	`}`

Prism	`controller.StateChanged += newEventHandler<StateChangedEventArgs>(MyKey_StateChanged);`
	
	`public void MyKey_StateChanged(object sender, StateChangedEventArgs args)`
	`{`
	`if (args.ChangedKey == "key")`
	`{`
	`//...`
	`}`
	`}`

Uma vez que o evento é lançado sempre que o método *setState* é executado, todos os *handlers* para o evento serão executados, independentemente da variável que foi alterada. Assim, é necessário que cada *handler* verifique qual foi a variável que foi alterada, através da propriedade *ChangedKey*.

4.1.6 Comandos

Os comandos são um mecanismo que permite que existam vários controlos na interface com o utilizador que realizem a mesma acção, ou seja, que executem o mesmo método, sem que seja necessário manterem qualquer referência entre eles. Para isso são necessários dois passos distintos: (1) associar um comando a um controlo e (2) associar um método a um comando, que será executado quando o comando for chamado por um dos controlos a que está associado.

O conceito de comando existe em CAB e em *Prism*, embora seja implementado de modos diferentes. A principal diferença está no local onde os comandos são registados: em CAB ficam guardados numa colecção do *WorkItem*, enquanto em *Prism* devem ficar guardados numa colecção definida pelo programador. Uma possível solução é a criação de uma classe *static* contendo um dicionário que associe uma *string* a cada comando.

Na Tabela 7 é apresentada a forma como deve ser associado um evento de um determinado controlo a um comando.

Tabela 7 – Associação de um botão a um comando

CAB	`workItem.Commands["button1Click"].AddInvoker(View.button1, "Click");`
Prism	`View.button1.Click += delegate { GlobalCommands.Commands["button1Click"].Execute(arg); };`

No caso de CAB, basta chamar o método *AddInvoker*, indicando qual o controlo e evento a que se deseja associar a execução do comando. Sempre que esse evento for lançado, o comando será executado. Em *Prism*, basta associar ao evento desejado um *delegate* que execute o comando. O acesso ao comando é feito através da classe *static* criada para esse efeito (*Global-Commands*, no exemplo da Tabela 7).

Para além de associar um evento a um comando, é necessário definir qual o método que será executado quando o comando é lançado. A Tabela 8 exemplifica o modo como isso pode ser conseguido em CAB e em *Prism*.

Tabela 8 – Registo de um handler para um comando

CAB	`[CommandHandler("button1Click")]`
	`public void OnButton1Click(object sender, EventArgs e)`
	`{`
	`//...`
	`}`

Prism	`public void OnButton1Click(object arg)`
	`{`
	`//...`
	`}`
	
	`//Registar o handler`
	`GlobalCommands.Commands.Add("button1Click", new DelegateCommand<object>(OnButton1Click));`

Em CAB, o método é associado ao comando através do atributo *CommandHandler*, enquanto em *Prism* a associação deve ser feita explicitamente. Uma diferença importante está na assinatura do método — em *Prism* o método não pode receber mais do que um parâmetro, e portanto não pode corresponder à assinatura de um *EventHandler*, como acontece em CAB.

4.1.7 Eventos

Tanto CAB como *Prism* possuem mecanismos que permitem publicar e subscrever eventos em componentes distintos, sem que sejam mantidas referências entre eles.

A publicação de eventos em CAB é feita através do atributo *EventPublication*, que pode ser associado a uma declaração de um evento. Em *Prism* é possível obter o mesmo comportamento, mas a publicação tem que ser feita explicitamente em código, utilizando a classe *EventAggregator* disponibilizada pela *framework*. A Tabela 9 ilustra essas diferenças.

Tabela 9 – Publicação de eventos

CAB	`[EventPublication("MyEvent", PublicationScope.Global)]`
	`public event EventHandler<MyEventArgs> MyEvent;`
	`//...`
	`MyEvent(this, e);`

Prism	`// Definição do evento`
	`public class MyEvent : CompositePresentationEvent<MyEventArgs>`
	`{`
	`//...`
	`}`
	`// Publicação`
	`MyEventArgs arg = new MyEventArgs();`
	`eventAggregator.GetEvent<MyEventArgs>().Publish(arg);`

A principal diferença está no facto de em *Prism* ser necessário definir uma classe que derive de *CompositePresentationEvent* para representar e identificar o evento, algo que em CAB era conseguido apenas com a declaração do evento, uma vez que o evento era identificado através de uma *string*.

Convém notar que as duas *frameworks* utilizam uma terminologia ligeiramente diferente. Na documentação de *Prism*, o acto de publicar um evento corresponde a lançar o evento em si, enquanto em CAB a publicação e o lançamento são dois actos independentes. Em CAB, o evento em si é lançado normalmente, como qualquer outro evento em *.NET* (invocando o evento como se fosse um método). Em *Prism*, é necessário obter uma referência para um objecto que representa o evento e utilizá-la para chamar o método *Publish*, que lançará o evento em si.

Relativamente à subscrição de eventos, a utilização do atributo *EventSubscription* em CAB é substituída pela utilização da classe *EventAggregator* em *Prism*. As diferenças são apresentadas na Tabela 10.

Tabela 10 – Subscrição de eventos

CAB	`// Subscrição`
	`[EventSubscription("MyEvent", Thread = ThreadOption.UserInterface)]`
	`public void OnMyEvent(object sender, MyEventArgs e)`
	`{`
	`//...`
	`}`

Prism	

```
// Subscrição
eventAggregator.GetEvent<MyEvent>().Subscribe(MyHandler, ThreadOption.UIThread);
// Definição do handler
public void MyHandler(MyEventArgs arg)
{
//...
}
```

Para além da forma como o evento é subscrito, a principal diferença está na assinatura do *handler*, que em *Prism* recebe apenas os argumentos do evento.

4.1.8 *UIExtensionSites*

CAB oferece uma funcionalidade importante para a composição de elementos visuais, permitindo à *shell* disponibilizar partes da sua interface com o utilizador e torná-las acessíveis aos módulos da aplicação. Uma aplicação típica é a implementação de uma barra de ferramentas, localizada na *shell*, que deve ser adaptada consoante o componente que estiver a ser mostrado a cada momento. A *shell* disponibiliza a barra de ferramentas como um *UIExtensionSite* e os componentes podem modificá-la, adicionando ou removendo os botões apropriados.

Esta funcionalidade não existe em *Prism*, pelo que terá que ser implementada pelo programador. Uma das soluções possíveis consiste em utilizar um dicionário, guardado na s*hell*, que associa um nome a cada um dos controlos que esta deseja disponibilizar. A *shell* deve implementar uma interface que obrigue a disponibilizar esse dicionário como uma propriedade. Por fim, pode ser criada uma classe *static* que guarde uma referência para a *shell*, permitindo aos módulos aceder ao dicionário e consequentemente aos controlos que esta disponibiliza. A Tabela 11 ilustra o modo como esta ideia pode ser implementada.

Tabela 11 – Implementação de UIExtensionSites em Prism

Prism	`public class UIExtensionSitesCollection : Dictionary<string, UIElement>`
	`{`
	`}`
	
	`public interface IExtensibleShell`
	`{`
	`  UIExtensionSitesCollection UIExtensionSites { get; }`
	`}`
	
	`//Prism Shell`
	`public partial class Shell : Window, IExtensibleShell`
	`{`
	`  //...`
	
	`  private UIExtensionSitesCollection _extensionSites;`
	`  public UIExtensionSitesCollection UIExtensionSites`
	`  {`
	`  get`
	`  {`
	`  if (_extensionSites == null)`
	`  {`
	`  _extensionSites = new UIExtensionSitesCollection();`
	`  }`
	`  return _extensionSites;`
	`  }`
	`  }`
	
	`  //...`
	`}`
	
	`public static class UIExtensionService`
	`{`
	`public static IExtensibleShell Shell { get; set; }`
	`}`

A Tabela 12 apresenta as principais diferenças da utilização de *UIExtensionSites* em CAB e em *Prism*, considerando que é utilizada a implementação sugerida acima.

Tabela 12 – Utilização de UIExtensionSites

CAB	`// Registo de um menu`
	`RootWorkItem.UIExtensionSites.RegisterSite("SiteName", Shell.MainMenuStrip);`
	
	`// Adicionar elementos ao menu`
	`WorkItem.UIExtensionSites["SiteName"].Add(newButton);`

Prism	`// Registo de um menu`
	`Shell.UIExtensionSites.Add("SiteName", Shell.MainMenuStrip);`
	`UIExtensionService.Shell = Shell;`
	
	`// Adicionar elementos ao menu`
	`MenuItem mainMenuStrip = UIExtensionService.Shell.UIExtensionSites["SiteName"] as`
	`  MenuItem;`
	`mainMenuStrip.Items.Add(newButton)`

O registo de um *UIExtensionSite* em CAB é feito através da função *RegisterSite*, enquanto em *Prism*, utilizando a implementação sugerida, basta adicionar o controlo a registar à colecção *UIExtensionSites* da *shell*. Adicionalmente, é necessário guardar uma referência para a *shell* na classe estática *UIExtensionService*.

O acesso aos controlos disponibilizados em CAB é feito através da colecção *UIExtension-Sites* do *WorkItem*. Em *Prism*, é necessário aceder à *shell* através da classe *UIExtensionService* e utilizar a colecção *UIExtensionSites* para obter o controlo.

4.2 Descrição da infra-estrutura de migração

Na secção anterior foram identificadas as principais diferenças entre as duas *frameworks*. Nos casos em que um mecanismo de CAB não existe em *Prism,* foi sugerida uma possível implementação que permita simulá-lo.

No entanto, de forma a evitar que a migração tivesse que ser realizada segundo uma abordagem do tipo *big-bang*, foi desenvolvida uma infra-estrutura de migração que tem como principal objectivo garantir a interoperabilidade entre CAB e *Prism*, possibilitando uma transição suave entre as duas tecnologias. A implementação dessa infra-estrutura procura encapsular as diferenças entre as duas tecnologias, abordadas na secção anterior.

Uma vez que ambas as tecnologias servem propósitos bastante semelhantes, procurou-se definir uma abstracção que permitisse que os vários componentes de uma solução deixassem de depender directamente de CAB ou de *Prism*, passando a depender apenas da infra-estrutura de migração. Dessa forma, em condições ideais, a migração poderia ser realizada substituindo a implementação da infra-estrutura, de forma a utilizar CAB ou *Prism*.

Uma vez que todos os módulos existentes na empresa dependem directamente de CAB, o primeiro passo para a migração seria um processo de *refactoring*[10], alterando as soluções já existentes para passarem a utilizar exclusivamente a infra-estrutura desenvolvida, removendo todas as dependências explícitas de CAB. Uma vez que a infra-estrutura é apenas uma abstracção, é possível realizar a migração de forma iterativa. A infra-estrutura pode ser introduzida gradualmente. Por exemplo, é possível alterar as soluções para que os comandos passem a ser registados e chamados através da infra-estrutura, mesmo que outros mecanismos (como a injecção de *views* ou eventos) impliquem que o módulo ainda tenha dependências directas de CAB. Numa fase seguinte, os eventos poderiam passar a ser tratados através da infra-estrutura, mantendo sempre o sistema funcional durante o processo. Isto seria repetido até que todas as funcionalidades estivessem migradas e todas as dependências de CAB tivessem sido removidas, ficando assim o sistema pronto a ser migrado para *Prism*.

[10] *Refactoring* é definido por Fowler [Fow99] como o processo de modificação de código pré-existente, alterando a sua estrutura interna sem mudar o seu comportamento externo.

Na secção seguinte será apresentada uma visão geral sobre a estruturação da infra-estrutura desenvolvida, seguida por uma descrição mais detalhada sobre a utilização da mesma, juntamente com alguns detalhes de implementação relevantes. Para cada funcionalidade será apresentada uma comparação entre a forma como ela era conseguida em CAB e como é conseguida através da infra-estrutura de migração.

4.2.1 Arquitectura da infra-estrutura

A Figura 15 apresenta uma visão geral da arquitectura da infra-estrutura de migração desenvolvida. Para efeitos de simplicidade, durante esta descrição da infra-estrutura, será ignorado o prefixo *MigrationInfrastructure*.

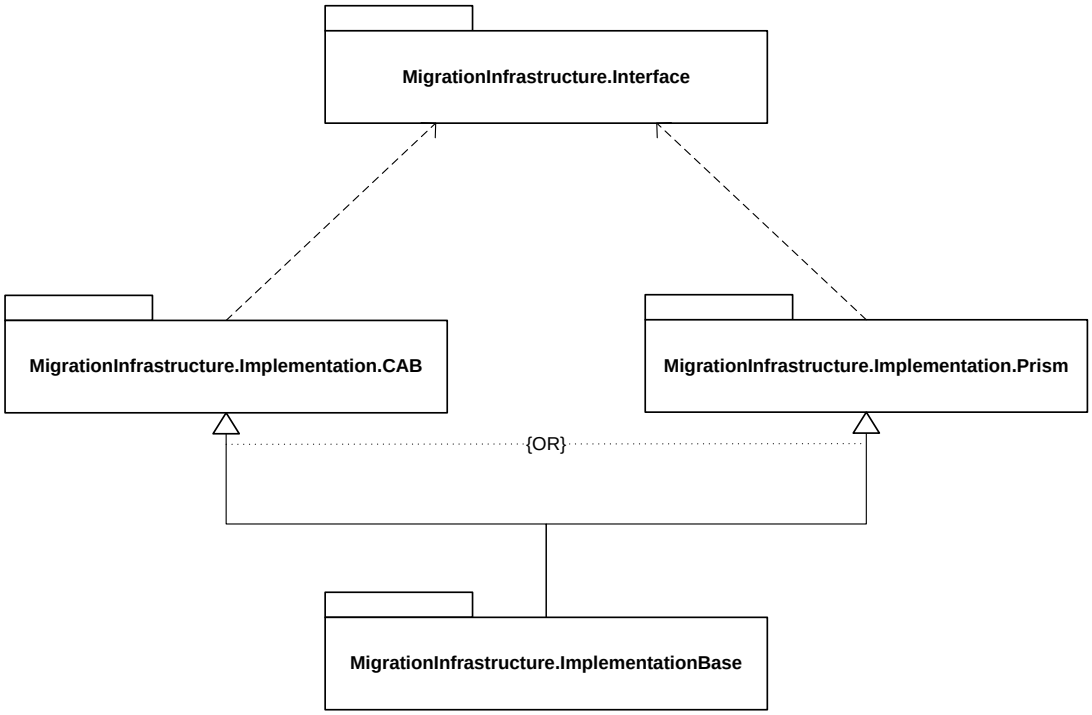


Figura 15 – Arquitectura da infra-estrutura de migração

O módulo *ImplementationBase* contém classes que derivam das classes do módulo *Implementation.CAB* ou *Implementation.Prism*. Essas classes não contêm qualquer funcionalidade e existem apenas para que seja possível seleccionar qual a *framework* que se deseja que a infra-estrutura utilize. Isso é feito através de uma *flag* de compilação, que selecciona qual o módulo do qual cada classe deve derivar. A Tabela 13 apresenta um exemplo.

Tabela 13 – Implementação de uma classe do módulo ImplementationBase

```
#if CAB
    public abstract class ModuleInit : GlinttHS.MigrationInfrastructure.CAB.ModuleInit
#else
    public abstract class ModuleInit : GlinttHS.MigrationInfrastructure.Prism.ModuleInit
#endif
    {
     }
```

Os componentes da solução final (os "clientes" da infra-estrutura) apenas referenciam este módulo e nunca o *Implementation.CAB* ou *Implementation.Prism*. Caso a *flag* de compilação "CAB" esteja activada, as classes em *ImplementationBase* derivarão das classes que implementam a infra-estrutura para CAB, no módulo *Implementation.CAB*. Inversamente, caso a *flag* não seja activada, a infra-estrutura utilizará a implementação de *Prism* em *Implementation.Prism*. Dessa forma, com a simples alteração de uma *flag* de compilação deverá ser possível escolher qual a *framework* a utilizar.

O módulo *Interface* define interfaces que deverão ser implementadas pelas classes dos módulos *Implementation.CAB* e *Implementation.Prism*, obrigando a que disponibilizem os mesmos métodos e propriedades. Isto garante que os clientes da infra-estrutura terão acesso às mesmas funcionalidades, independentemente de ser utilizado CAB ou *Prism.*

4.2.2 *Controllers*

A classe *Controller* constitui um ponto central da infra-estrutura de migração. O seu principal objectivo é substituir o *WorkItem* em CAB, pelo que, durante o processo de migração, todas as classes que derivem de *WorkItem* devem passar a derivar da classe *Controller*.

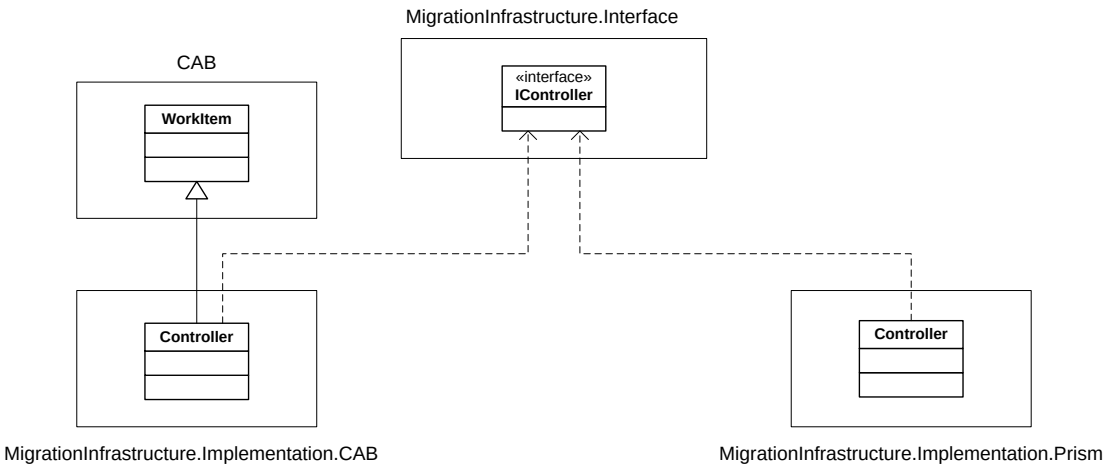


Figura 16 – Implementação das classes *Controller* em CAB e em *Prism*

Tal como é visível na Figura 16, ambas as implementações da classe *Controller*, em CAB e em *Prism*, implementam a interface *IController*, que especifica métodos para a gestão de variáveis

de estado, injecção de *views* e criação de *Controllers* "filho". No caso de CAB, a classe *Controller* deriva directamente de *WorkItem*, e portanto esses métodos são implementados com base nas funcionalidades oferecidas pelo *WorkItem* de CAB.

O facto de a implementação em CAB derivar de *WorkItem* facilita bastante o processo de migração. Numa fase inicial, os *WorkItems* já existentes na solução podem passar a derivar da classe *Controller* da infra-estrutura, mantendo todo o código já existente, ainda que continue a depender directamente de funcionalidades disponibilizadas pela classe *WorkItem* de CAB. Essas dependências poderão então ser progressivamente removidas, passando a utilizar as funcionalidades oferecidas pela classe *Controller*, que actua como um *Adapter* [Gam95] entre os clientes da infra-estrutura e a *framework* que se deseja utilizar.

4.2.3 Inicialização de módulos

Tanto em CAB como em *Prism* deve existir uma classe em cada módulo que é responsável pela sua inicialização. No caso de CAB, essa classe deve descender da classe *ModuleInit* disponibilizada pela *framework*, enquanto em *Prism* deve apenas implementar a interface *IModuleInit*. A Figura 17 apresenta a estrutura disponibilizada pela infra-estrutura de migração para esse efeito.

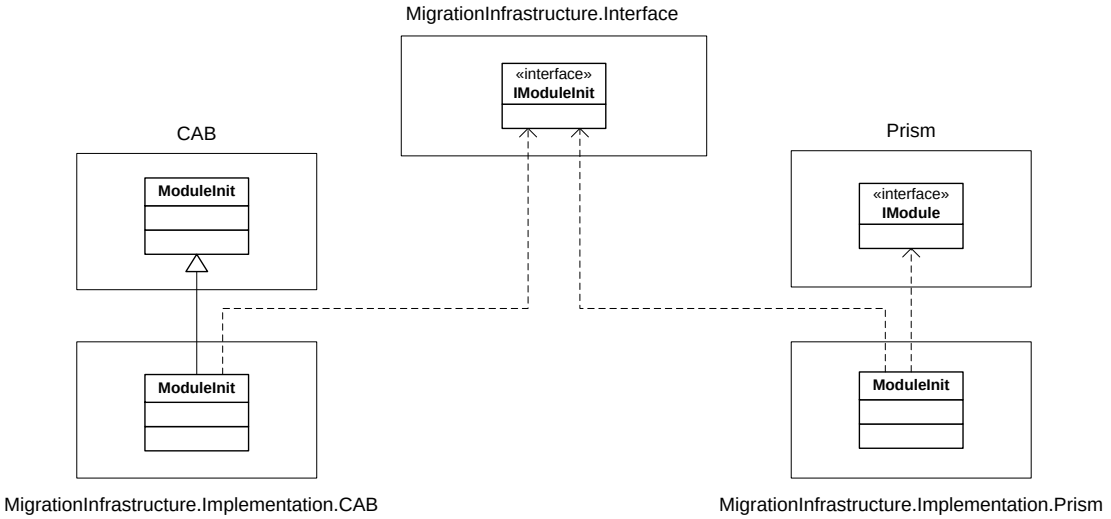


Figura 17 – Implementação das classes *ModuleInit*

A principal funcionalidade disponibilizada pelas classes *ModuleInit* da infra-estrutura é a criação e lançamento de *Controllers*, que encapsula as diferenças identificadas em 4.1.3. A Tabela 14 apresenta a forma como isso era conseguido em CAB e como pode ser conseguido utilizando a infra-estrutura.

Tabela 14 – Comparação entre a forma de criação de um WorkItem em CAB e a criação de um Controller utilizando a infra-estrutura

CAB	`MyWorkItem myWorkItem = parentWorkItem.WorkItems.AddNew<MyWorkItem>();` `myWorkItem.Run("tabWorkspace1");`
Infra-estrutura	`MyWorkItem myWorkItem = this.createNewController<MyWorkItem>();` `myWorkItem.Run("tabWorkspace1“);`

Convém notar que quando a infra-estrutura é utilizada, a classe que é instanciada deve ser uma subclasse da classe *Controller*, ou seja, não é possível criar directamente *WorkItems* de CAB.

4.2.4 Definição e injecção de *views*

A infra-estrutura oferece duas classes base que devem ser utilizadas pelos clientes para definir *views* em WPF — *WpfUserView* — e em *Windows Forms* — *WinFormsUserView*. Essas duas classes, juntamente com a classe *Presenter*, permitem implementar de forma transparente o padrão *Model-View-Presenter*, recomendado pela *Microsoft*. A Figura 18 apresenta as relações entre essas classes. Não são explicitadas as diferentes implementações em CAB e em *Prism* já que não existem diferenças em termos conceptuais.

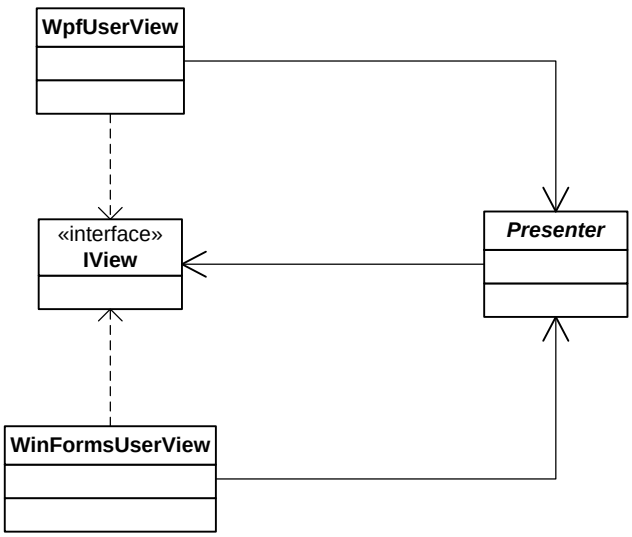


Figura 18 – Implementação do padrão Model-View-Presenter na infra-estrutura

Assim, a definição de uma *view* resume-se apenas à criação de um *UserControl* que derive de uma das duas classes da infra-estrutura: *WpfUserView* no caso de WPF e *WinFormsUserView* no caso de *WindowsForms*. A classe *Presenter* é abstracta, devendo ser utilizada como base de uma classe definida no módulo cliente. Uma vez que é a *view* que cria o respectivo *Presenter*, cada uma das classes de *views* recebe um parâmetro de tipo[11] que permite indicar qual o tipo concreto de *Presenter* a injectar na *view*.

Relativamente à injecção de *views*, passa a ser efectuada através de métodos disponibilizados pela classe *Controller*. Esses métodos permitem criar *views* de um determinado tipo e injectá-las numa região ou *Workspace* de forma transparente, seja uma *view* em WPF ou em *WindowsForms*, de acordo com o que foi proposto na secção 4.1.4. A Tabela 15 apresenta o modo como são criadas e mostradas as *views* em CAB e utilizando a infra-estrutura.

Tabela 15 – Comparação entre a forma de criação e injecção de views em CAB e utilizando a infra-estrutura de migração

CAB	`MyView view = workItem.Items.AddNew<MyView>();` `workItem.Workspaces["workspaceName"].Show(view);`
Infra-estrutura	`MyView view = this.CreateView<MyView>();` `this.ShowView(view, "workspaceName");`

A *view* que é criada pode ser WPF ou *Windows Forms* e a *string* que identifica o local onde ela será injectada pode corresponder a um *Workspace* ou a uma região.

4.2.5 Variáveis de estado

A gestão de variáveis de estado é feita através de métodos disponibilizados pela classe *Controller*, permitindo aceder e modificar variáveis de estado e injectar o seu valor noutros objectos. A Tabela 16 ilustra o modo como deve ser feito o acesso e escrita a variáveis de estado em CAB e através da infra-estrutura.

[11] *Type parameter* em inglês

Tabela 16 – Acesso a variáveis de estado em CAB e através da infra-estrutura

CAB	`//Escrita`
	`workItem.State["key"] = anObject;`
	
	`//Leitura`
	`object stateVariable = workItem.State["key"];`

Infra-estrutura	`//Escrita`
	`controller.setState("key", anObject);`
	
	`//Leitura`
	`object stateVariable = controller.getState("key");`

A classe *Controller* também contém um método que permite injectar o valor de variáveis de estado em propriedades de outros objectos, realizando a função correspondente ao atributo *State* em CAB. A Tabela 17 apresenta o modo como isso pode ser conseguido.

Tabela 17 – Injecção de variáveis de estado em CAB e utilizando a infra-estrutura

CAB	`[State("mensagem")]` `public string StateMsg{ get; set; }`

Infra-estrutura	`[NewState("mensagem")]`
	`public string StateMsg{ get; set; }`
	
	`//Ao criar o objecto...`
	`MyView view = new MyView();`
	`controller.InjectMyState(view);`

De acordo com o que foi discutido na secção 4.1.5, a infra-estrutura oferece um atributo *NewState* que marca as propriedades onde deverá ser injectado estado. Depois, quando é criado o objecto em que se deseje injectar variáveis de estado, deve ser chamado o método *InjectMyState*, definido na classe *Controller*, que realiza a injecção de estado propriamente dita.

4.2.6 Comandos

A infra-estrutura de migração permite associar comandos a controlos, tal como acontece em CAB. Para isso, foi criada uma classe *CommandBroker*, com implementações distintas para CAB e para *Prism*. A Figura 19 representa a forma como são implementadas estas classes.

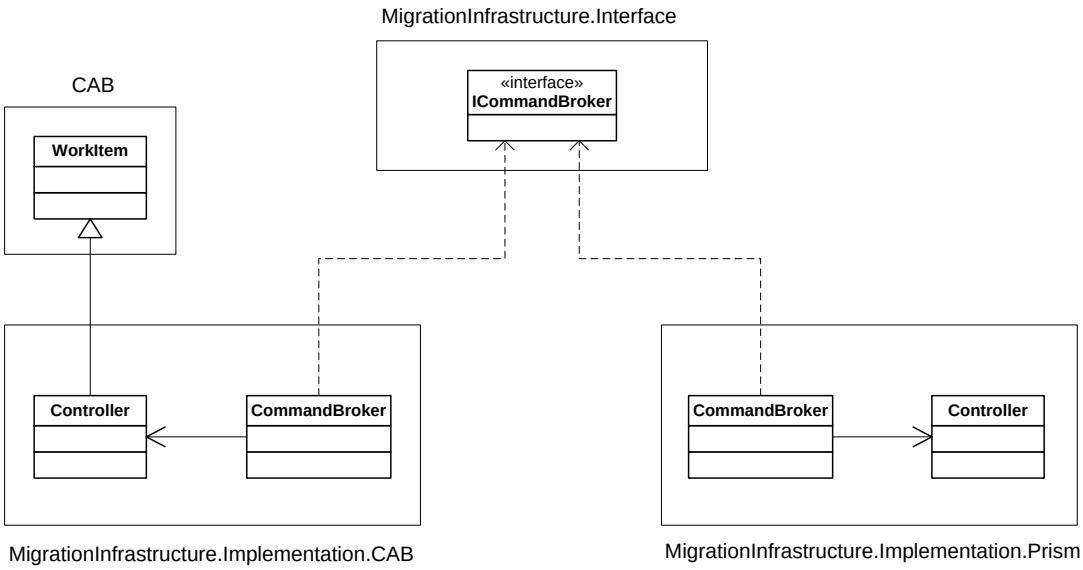


Figura 19 – Implementação das classes CommandBroker

Em CAB é necessário que a classe *CommandBroker* tenha uma referência para um *WorkItem*, de forma a poder utilizar as funcionalidades de CAB para os comandos. No caso de *Prism*, a referência para o *Controller* não é necessária. Ainda assim, ela é mantida para que a utilização da classe seja igual do ponto de vista dos módulos clientes, independentemente de ser utilizada a implementação de CAB ou de *Prism*.

A Tabela 18 mostra a forma de associar um comando a um controlo.

Tabela 18 – Associação de um comando a um controlo

CAB	`workItem.Commands["button1Command"].AddInvoker(View.button1, "Click");`
Infra-estrutura	`ICommandBroker cmdBroker = new CommandBroker(controller);` `cmdBroker.BindInvoker(View.button1, "Click", "button1Command");`

A instalação de um *handler* para um comando também é realizada recorrendo à classe *CommandBroker*. A Tabela 19 ilustra o modo como isso pode ser conseguido.

Tabela 19 – Instalação de um handler para um comando

CAB	`[CommandHandler("button1Command")]`
	`public void OnButton1Command(object sender, EventArgs e)`
	`{`
	`//...`
	`}`

Infra-estrutura	`public void OnButton1Command(EventArgs e)`
	`{`
	`//...`
	`}`
	`//Registar o handler`
	`ICommandBroker cmdBroker = new CommandBroker(controller);`
	`cmdBroker.InstallHandler("button1Command", OnButton1Command);`

Convém notar que, no caso da infra-estrutura, o *handler* do comando recebe apenas um parâmetro do tipo *EventArgs*, não recebendo uma referência para o objecto que o chamou, uma vez que isso não é suportado em *Prism*.

4.2.7 Eventos

Tal como foi dito anteriormente, tanto CAB como *Prism* oferecem funcionalidades que permitem publicar e subscrever eventos. No entanto, a forma de o fazer é ligeiramente diferente em cada uma delas, pelo que a infra-estrutura de migração adopta uma arquitectura que permite encapsular essas diferenças, oferecendo uma interface comum para ambas as *frameworks*. A Figura 20 apresenta um diagrama lógico das classes envolvidas.

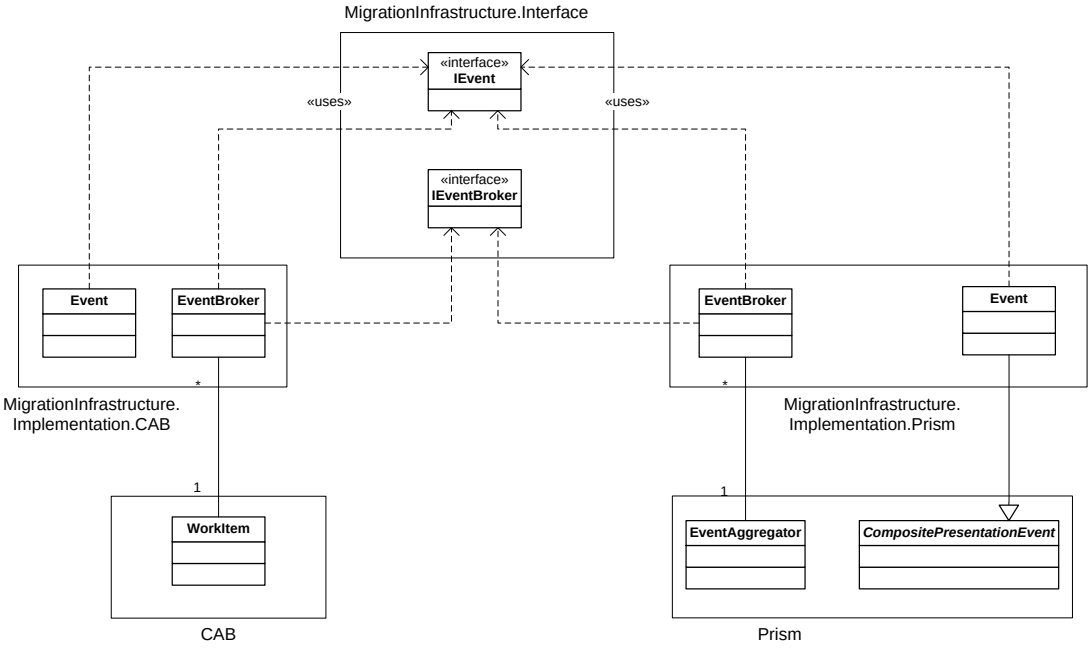


Figura 20 – Implementação das classes Event e EventBroker

Mais uma vez, existe uma interface partilhada entre as implementações para CAB e *Prism*. A classe *EventBroker* é a responsável por realizar as publicações e subscrições de eventos junto da respectiva *framework*. No caso de CAB, essa classe deve manter uma referência para um *WorkItem*, já que é junto dessa classe que deve ser realizado o registo dos eventos. Em *Prism*, os eventos são registados pela classe *EventAggregator*, pelo que o *EventBroker* mantém uma referência para um objecto desse tipo.

Tal como já foi dito acima, os eventos em CAB são identificados por uma *string*, enquanto em *Prism* são identificados por um tipo, descendente da classe *CompositePresentationEvent*. Assim, os métodos para publicar e subscrever eventos recebem um parâmetro de tipo e uma *string*, permitindo identificar o evento em ambas as *frameworks* e mantendo uma interface comum entre as duas implementações da infra-estrutura.

A Tabela 20 apresenta o modo como pode ser publicado um evento.

Tabela 20 – Publicação de eventos utilizando a infra-estrutura

CAB	`[EventPublication("MyEvent", PublicationScope.Global)]`
	`public event EventHandler<MyEventArgs> MyEvent;`
	`//...`
	`MyEvent(this, e);`

Infra-estrutura	`// Definição do evento`
	`public class MyEvent : Event`
	`{`
	`  //...`
	`}`
	`// Publicação`
	`EventBroker broker = new EventBroker(WorkItem);`
	`broker.fireEvent<MyEvent>("MyEvent", sender, new MyEventArgs(eventArgs));`

A subscrição de eventos é exemplificada na Tabela 21.

Tabela 21 – Subscrição de eventos

CAB	`// Subscrição`
	`[EventSubscription("MyEvent", Thread = ThreadOption.UserInterface)]`
	`public void OnMyEvent(object sender, MyEventArgs e)`
	`{`
	`//...`
	`}`

Infra-estrutura	`// Subscrição`
	`EventBroker broker = new EventBroker(controller);`
	`broker.subscribeEvent<CopyEvent>("MyEvent", OnCopyEvent);`
	`// Definição do handler`
	`public void OnMyEvent(object sender, EventArgs e)`
	`{`
	`  //...`
	`}`

A classe *EventBroker* é utilizada para publicar e subscrever eventos, substituindo a utilização dos atributos *EventPublication* e *EventSubscription* em CAB.

4.2.8 *UIExtensionSites*

Tal como já foi explicado acima, a noção de *UIExtensionSites* não é suportada por *Prism*, passando a ser uma responsabilidade do programador. De acordo com a implementação sugerida em 4.1.7, a infra-estrutura procura encapsular essas diferenças, tornando a utilização deste mecanismo independente da *framework* que está a ser utilizada. A Figura 21 apresenta a estrutura lógica da infra-estrutura no que toca a *UIExtensionSites*.

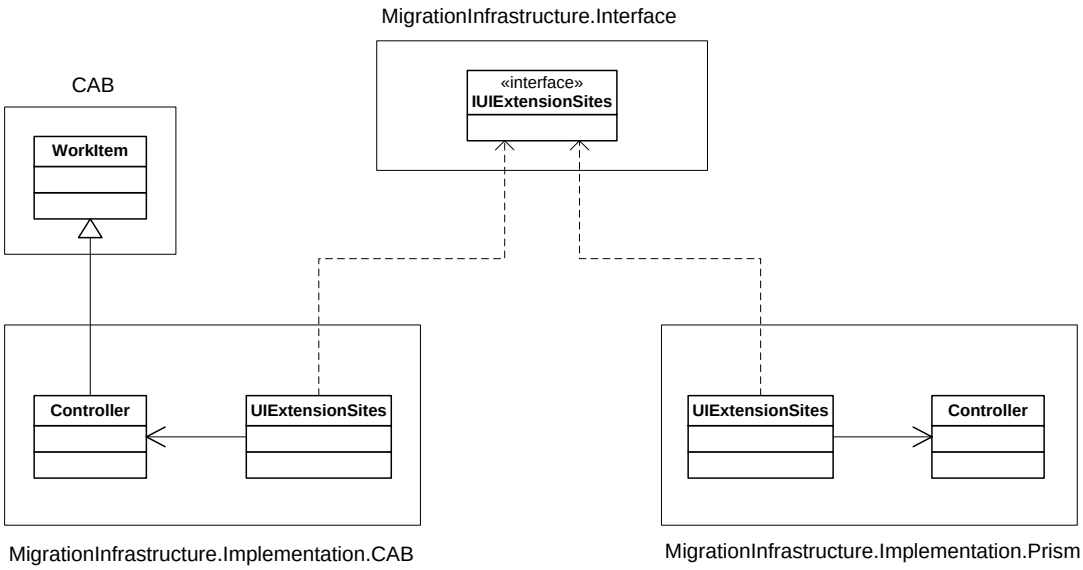


Figura 21 – Implementação de UIExtensionSites

Tal como nos outros mecanismos já apresentados, existem duas implementações distintas da interface *IUIExtensionSites*, uma para CAB e outra para *Prism*. Como no caso de CAB o registo de *UIExtensionSites* é feito através do *WorkItem*, é necessário manter uma referência para um objecto desse tipo, de modo a aproveitar as funcionalidades oferecidas pela *framework*. Para manter a consistência entre as duas implementações, a infra-estrutura obriga a que a implementação de *Prism* também mantenha uma referência para um *Controller*.

A Tabela 22 apresenta a forma de utilização deste mecanismo em CAB e na infra-estrutura de migração.

Tabela 22 – Utilização de UIExtensionSites

CAB	`// Registo`
	`RootWorkItem.UIExtensionSites.RegisterSite("SiteName", Shell.MainMenuStrip);`
	
	`// Adicionar elementos`
	`WorkItem.UIExtensionSites["SiteName"].Add(newButton);`

Infra-estrutura	`// Registo`
	`IUIExtensionSites extensionSites = new UIExtensionSites(controller);`
	`extensionSites.RegisterSite("SiteName", Shell.MainMenuStrip);`
	`UIExtensionService.Shell = Shell;  // Apenas em Prism`
	
	`// Adicionar elementos`
	`extensionSites.Add("SiteName", newButton);`

A classe *UIExtensionSites* permite registar *sites* e adicionar elementos a um *site* previamente registado. Convém notar que existe um passo adicional que é preciso realizar para a utilização de *UIExtensionSites* em *Prism*: guardar uma referência para a *shell* na classe estática

UIExtensionService. Este é um ponto em que a utilização da *framework* difere consoante seja utilizada a implementação de CAB ou de *Prism*.

4.3 Análise de desempenho

A infra-estrutura de migração desenvolvida esconde os detalhes específicos de cada *framework*, e portanto introduz uma camada de complexidade adicional. Assim, é importante analisar até que ponto isso implica uma redução de desempenho nas aplicações finais.

Serão aqui apresentadas as análises de *performance* realizadas, comparando o desempenho de três *views* de exemplo, desenvolvidas em WPF. As características de cada *view* são apresentadas na Tabela 23.

Tabela 23 – Constituição das view utilizadas para testes de desempenho

View	Tipo de Controlos	Número
Controlos *Standard*	*Label*	1
	Textbox	1
	Combobox	1
	Botão	1
	Listbox	1
Controlos *Infragistics*	*Label*	1
	Textbox	1
	DataGrid	1
	Chart	1

View	Tipo de Controlos	Número
Muitos controlos *Infragistics*	*DataGrid*	11
	CarouselListBox	7
	Chart	6
	DataPresenter	5
	MaskedEditor	5
	Ribbon	3
	DateTimeEditor	1
	Label	1
	Textbox	1

As três *views* têm diferentes níveis de complexidade, de forma a testar o desempenho da infra-estrutura em diferentes situações. A primeira *view* (controlos *standard*) procura representar um formulário muito simples, com poucos controlos, enquanto a segunda já é composta por dois controlos complexos da *Infragistics*[12]. Finalmente, a última *view* foi desenvolvida como um teste de desempenho mais agressivo, tendo como objectivo analisar o comportamento da infra-estrutura sob condições extremas.

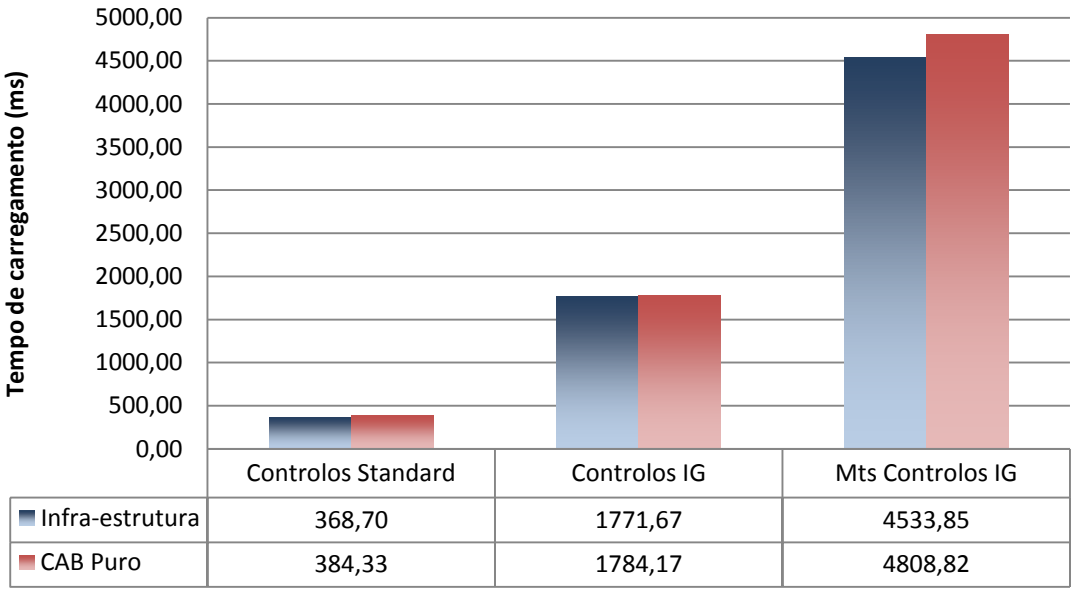
Todas as *views* foram integradas numa *shell* desenvolvida em *Windows Forms*, estando acessíveis através de um botão.

[12] A *Infragistics* é uma empresa que fornece controlos complexos para várias plataformas, incluindo WPF. Mais informações em http://www.infragistics.com/.

4.3.1 Tempo de carregamento

Os tempos de carregamento foram medidos desde que o botão que mostra a *view* é clicado até ao momento em que a *view* está complemente carregada.

A Figura 22 apresenta uma comparação entre os tempos de carregamento das várias *views* em CAB puro e utilizando a infra-estrutura sobre CAB.

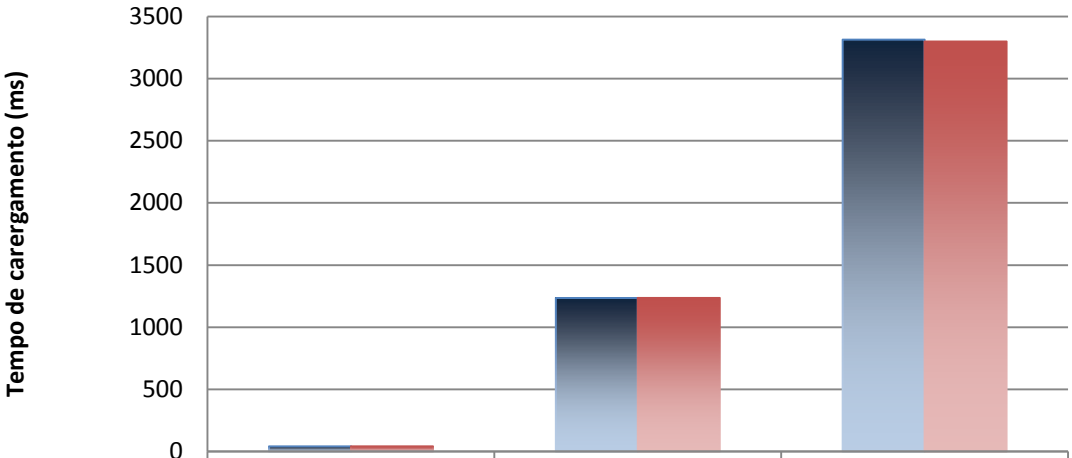


	Controlos Standard	Controlos IG	Mts Controlos IG
Infra-estrutura	368,70	1771,67	4533,85
CAB Puro	384,33	1784,17	4808,82

Figura 22 – Comparação dos tempos de carregamento em CAB

Tal como é possível constatar a partir do gráfico, a infra-estrutura consegue uma *performance* comparável à utilização de CAB puro, apresentando uma ligeira vantagem nas *views* mais complexas. A explicação reside no facto de a infra-estrutura realizar a maioria das operações (publicação/subscrição de eventos, instalação de comandos, etc.) através da chamada de métodos, não recorrendo à utilização de atributos como é usual em CAB. Tal como foi dito em 3.3.2, isso implica a utilização de *reflection*, com a consequente penalização no desempenho.

Foi realizada a mesma análise em *Prism*, utilizando a mesma metodologia. Os resultados são apresentados na Figura 23.



	Controlos Standard	Controlos IG	Mts Controlos IG
Infra-estrutura	40,56	1233,90	3311,68
Prism Puro	40,62	1234,25	3296,56

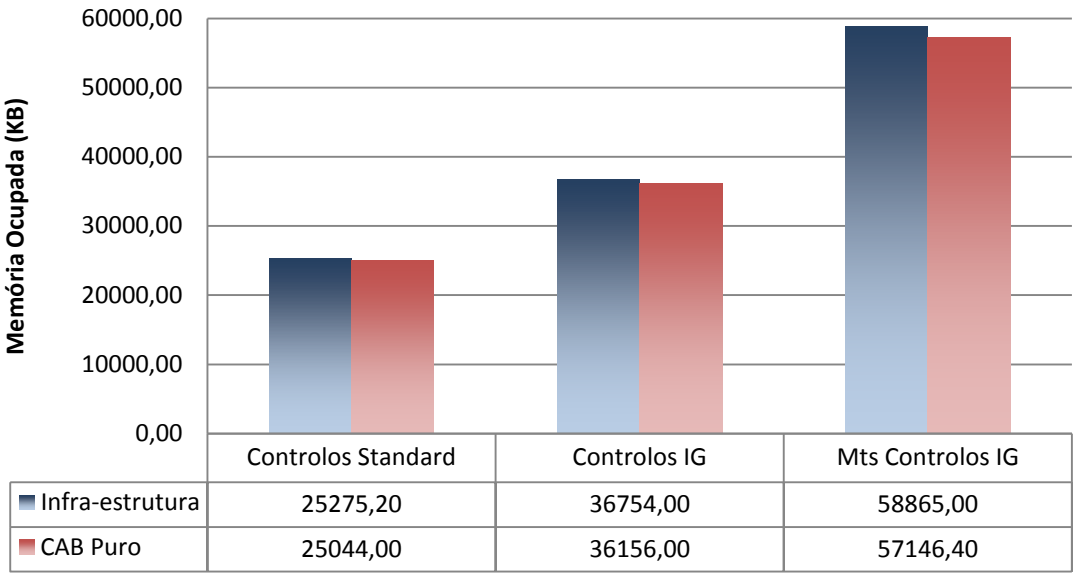
Figura 23 – Comparação dos tempos de carregamento em Prism

Tal como em CAB, verifica-se que a infra-estrutura apresenta um desempenho semelhante à *framework* simples. Com efeito, as diferenças entre a utilização da infra-estrutura sobre *Prism* e *Prism* puro são praticamente negligenciáveis, cifrando-se na ordem dos milésimos de segundo.

4.3.2 Memória

Para além dos tempos de carregamento, foi medida a quantidade de memória consumida por cada *view*. Para isso, ao total de memória ocupada pela aplicação com cada *view* activa subtraiu-se a memória ocupada pela aplicação apenas com a *shell* visível. Obteve-se assim uma estimativa da memória consumida por cada *view*.

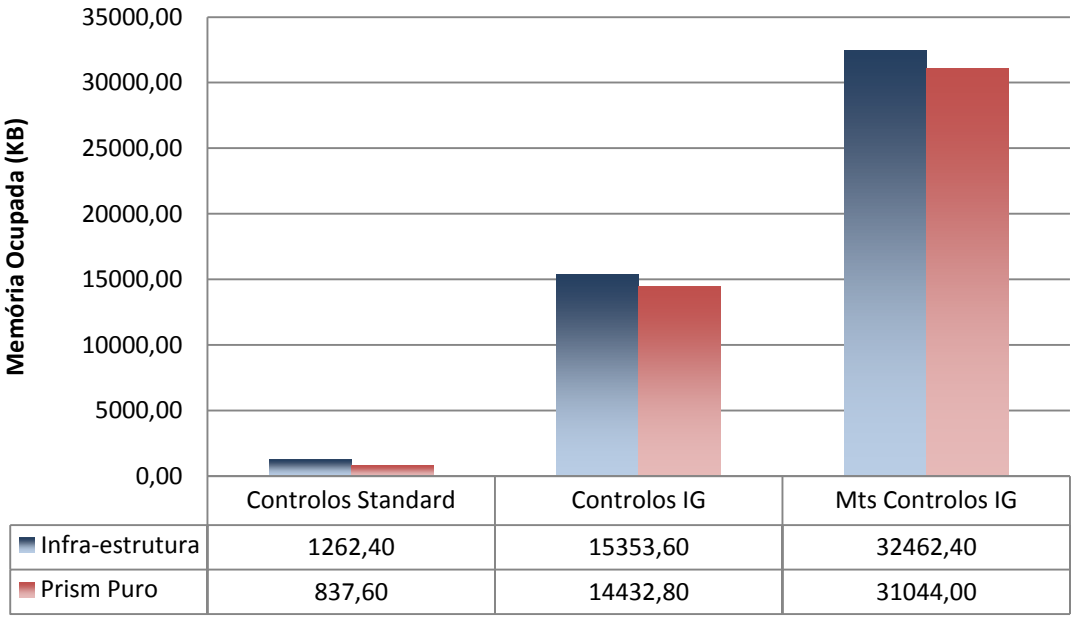
A Figura 24 apresenta uma comparação dos resultados dos testes da infra-estrutura sobre CAB e de CAB puro. A utilização de memória é bastante semelhante nos dois casos, verificando-se que a infra-estrutura não introduz, nesta situação, um *overhead* significativo em termos de ocupação de memória.



	Controlos Standard	Controlos IG	Mts Controlos IG
Infra-estrutura	25275,20	36754,00	58865,00
CAB Puro	25044,00	36156,00	57146,40

Figura 24 – Quantidade de memória ocupada em CAB

Relativamente a *Prism,* a Figura 25 apresenta os resultados da infra-estrutura comparados com a utilização de *Prism* puro.



	Controlos Standard	Controlos IG	Mts Controlos IG
Infra-estrutura	1262,40	15353,60	32462,40
Prism Puro	837,60	14432,80	31044,00

Figura 25 – Quantidade de memória ocupada em Prism

Tal como em CAB, existe um grande equilíbrio entre a utilização de memória com e sem a infra-estrutura.

A infra-estrutura ocupa sempre um pouco mais de memória, mas tal resultado era inteiramente esperado, dada a introdução de mais uma camada de abstracção que a sua utilização implica.

4.4 Exemplo de migração

Nesta secção será apresentado um caso prático, exemplificando a migração de uma pequena aplicação de exemplo de CAB para *Prism*, permitindo ilustrar os passos necessários para a concretizar.

A aplicação em questão é bastante simples constituída apenas por um módulo, servindo para exemplificar a migração de cada um dos mecanismos identificados anteriormente. Foi desenvolvida inicialmente em CAB com *views* em *Windows Forms*, sendo aqui apresentados os passos necessários para a migrar para *Prism*. A Figura 26 apresenta a sua janela principal.

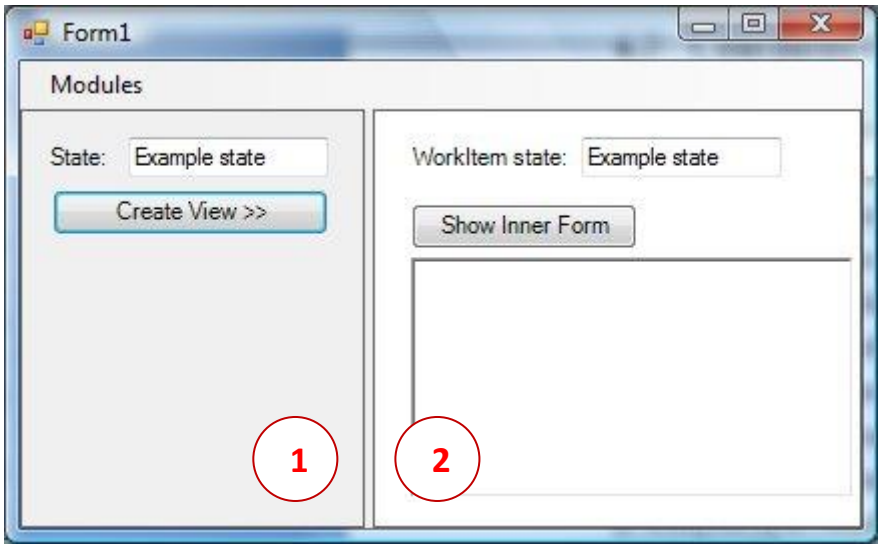


Figura 26 – Janela da aplicação de exemplo

A *Shell* em CAB disponibiliza dois *Workspaces* (marcados na figura pelos números 1 e 2), onde são injectadas duas *views*, que comunicam entre si utilizando eventos. Inicialmente, apenas é mostrada a *view* no *Workspace 1*, enquanto o *Workspace 2* permanece vazio. Quando o utilizador clica no botão "Create view", é lançado um evento que despoleta a criação e a injecção da *view* no *Workspace 2*. O texto que tiver sido escrito na caixa de texto "State", na *view* do lado esquerdo, é guardado numa variável de estado e injectado automaticamente na nova *view* que é criada do lado direito. O botão "Show Inner Form" permite mostrar um formulário de teste em WPF na área interior, exemplificando a interoperabilidade entre *Windows Forms* e WPF. O menu "Modules" contém apenas um item que permite actualizar a caixa de texto da *view* do lado esquerdo, exemplificando a utilização de comandos e *UIExtensionSites*.

4.4.1 Introdução da infra-estrutura

O primeiro passo para migrar a aplicação de exemplo de CAB para *Prism* é introduzir a infra-estrutura de migração na solução. Isso pode ser feito gradualmente, substituindo as operações que são realizadas com recurso a CAB pelas substitutas que a infra-estrutura de migração disponibiliza, de acordo com os vários aspectos apresentado em 4.2.

Tome-se como exemplo a criação e a injecção da *view* no *Workspace 2*. A Tabela 24 apresenta o código que era inicialmente utilizado pela aplicação para realizar essa operação.

Tabela 24 – Criação da view do Workspace 2, em CAB

```
ExampleView view = workItem.Items.AddNew<ExampleView>();
workItem.Workspaces[“Workspace2”].Show(view);
```

Tal como é possível verificar, esta operação utiliza o *WorkItem*, pelo que depende directamente de CAB. Ao realizar esta operação através da infra-estrutura, essa dependência é removida (Tabela 25).

Tabela 25 – Criação da view do Workspace 2, utilizando a infra-estrutura

```
ExampleView view = controller.CreateView<ExampleView>();
controller.ShowView(view, "Workspace2");
```

Agora, os métodos necessários são disponibilizados pela classe *Controller*, da infra-estrutura, deixando de existir uma referência directa ao *WorkItem*. Este processo pode ser realizado de forma gradual, ou seja, é possível que algumas *views* sejam injectadas através da infra-estrutura e outras através de CAB.

Deve ser seguido um processo semelhante para todos os mecanismos de CAB utilizados na aplicação, até que não seja utilizada qualquer classe de CAB em todo o seu código. É importante referir que durante este processo a aplicação pode ser compilada e executada, mesmo que a introdução da infra-estrutura não esteja completa.

4.4.2 Migração

Depois de a infra-estrutura ter sido introduzida, a aplicação continua a funcionar em CAB, mas está agora inteiramente suportada pela infra-estrutura. Nesse momento, o módulo encontra-se pronto para ser migrado para *Prism*. No entanto, para que seja possível migrar a aplicação na sua totalidade, é necessário desenvolver uma *Shell* em *Prism*, que irá substituir a de CAB assim que a migração for efectivada.

Reunidas todas as condições, basta apenas recompilar o projecto da infra-estrutura de migração sem a *flag* de compilação "CAB" activa, tal como é mostrado na figura.

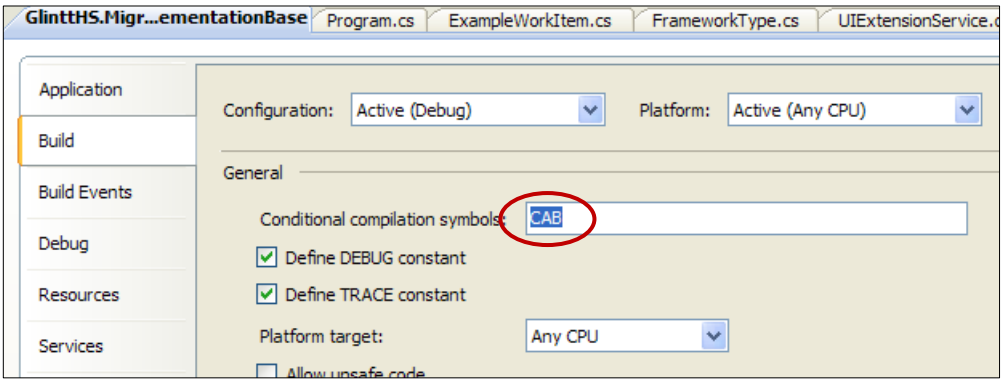


Figura 27 – Flag de compilação para CAB

Isto faz com que a infra-estrutura passe a utilizar a sua implementação para *Prism*, concretizando a migração. As *views* não precisam de ser modificadas, podendo continuar a ser utilizadas as que eram utilizadas em CAB, já que a infra-estrutura suporta a injecção de *views* em *Windows Forms* e em WPF de forma transparente. Posteriormente, caso se deseje também migrar as *views* para WPF, elas podem ser progressivamente redesenhadas na nova tecnologia.

Depois de desenvolvida a *Shell* em *Prism*, a aplicação de exemplo pode ser migrada livremente, no espaço de segundos, entre as CAB e *Prism*. Apesar de se tratar de um caso meramente exemplificativo, o processo de migração desta aplicação simples pode ser facilmente estendido a aplicações mais complexas, como é o caso do Processo Clínico.

4.5 Conclusões

A infra-estrutura de migração foi desenvolvida como um "esqueleto", disponibilizando, numa primeira fase, as funcionalidades essenciais para a utilização dos principais mecanismos disponibilizados pelas duas *frameworks*. Foram integradas as principais funcionalidades que a aplicação de Processo Clínico utiliza, procurando facilitar ao máximo a migração. No entanto, em soluções de grande dimensão, é natural que existam alguns pontos em que há dependências explícitas de alguns elementos de CAB. Assim, a infra-estrutura define um modelo sobre o qual pode assentar o desenvolvimento de mais funcionalidades que sejam eventualmente necessárias para suportar a migração total da aplicação de Processo Clínico.

À data da conclusão deste projecto, a infra-estrutura de migração era já utilizada em todas as soluções relacionadas com o Processo Clínico para a publicação e subscrição de eventos, registo e lançamento de comandos e inicialização de módulos. Isso significa que nesses aspectos já não existe qualquer dependência explícita de CAB, o que facilitará o processo de migração, tal como era objectivo do projecto.

Relativamente à *performance* da infra-estrutura desenvolvida, pode-se afirmar que é bastante satisfatória. Tal como é visível na secção anterior, a utilização da infra-estrutura não implica qualquer degradação significativa de desempenho.

Conclui-se, portanto, que a infra-estrutura de migração responde a todos os requisitos identificados e cumpre na totalidade os objectivos traçados no início do projecto.

Capítulo 5

Definição de protocolos de tratamento

A segunda fase do projecto consistiu na implementação de um módulo que seria integrado na aplicação de processo clínico, já existente na empresa. Tendo como principal destinatário e cliente o Instituto Português de Oncologia (IPO) Porto, o sistema tem como principal objectivo suportar a definição e prescrição de protocolos de tratamento oncológico.

A parte de definição de protocolos constitui o foco deste projecto, sendo que a prescrição e agendamento dos mesmos foi tratada num projecto que decorreu em paralelo, da autoria de José Carvalho.

O sistema desenvolvido constitui uma prova de conceito, exemplificando a utilização da infra-estrutura de migração desenvolvida.

Neste capítulo serão descritos os requisitos e a arquitectura do sistema. Os requisitos serão apresentados sob a forma de casos de utilização, pretendendo ilustrar todas as funcionalidades disponíveis. Será também detalhada a arquitectura do módulo desenvolvido segundo diversas perspectivas. Finalmente, a título de exemplo, será detalhada a implementação de um caso de utilização.

5.1 Requisitos da aplicação

Os requisitos do módulo desenvolvido foram previamente recolhidos pela *Glintt HS*, uma vez que a recolha de requisitos não fazia parte do âmbito do projecto. No entanto, esses requisitos foram identificados de modo informal durante duas reuniões, com o apoio de um protótipo da interface com o utilizador. Com base na informação recolhida durante essas reuniões, foram identificados os requisitos que aqui são descritos sob a forma de casos de utilização.

5.1.1 Casos de utilização

A Figura 28 apresenta um diagrama de casos de utilização, em UML, que resume os casos de utilização identificados para o módulo que foi desenvolvido.

Figura 28 – Diagrama de casos de utilização

Cada caso de utilização será seguidamente apresentado, juntamente com uma breve descrição.

Tabela 26 – Descrição dos casos de utilização

Caso de Utilização	Descrição
Configurar Protocolo	O utilizador introduz os dados de identificação e categorização do protocolo e configura algumas propriedades básicas como o número de ciclos, o número de sessões por ciclo e número de ciclos a agendar aquando da prescrição do protocolo.
Categorizar Protocolo	O utilizador selecciona o tipo de protocolo e agrupador que serão utilizados para categorizar o protocolo.
Definir Objectivos	Através de uma lista de possíveis objectivos apresentadas como um conjunto de *checkboxes*, o utilizador selecciona quais os mais apropriados para o protocolo em questão.
Seleccionar protocolos incompatíveis	O utilizador recorre a uma visualização em árvore dos protocolos já indicados no sistema, marcando os que são considerados incompatíveis com o que está a ser definido.
Indicar diagnósticos	O utilizador indica quais os diagnósticos para os quais o protocolo é aplicável, realizando uma pesquisa simples e adicionando os diagnósticos desejados a uma lista.
Indicar locais e postos de tratamento	O utilizador pode indicar um local e um posto de tratamento por defeito para todas as sessões definidas no protocolo. Além disso, é possível indicar um local e posto de tratamento específico para cada sessão de tratamento.
Editar critérios de inclusão e exclusão	O utilizador tem a possibilidade de inserir, eliminar ou reordenar critérios de inclusão (que determinam em que condições o protocolo é aplicável) e exclusão (que indicam situações em que o protocolo não pode ser aplicado).
Definir MCDT's obrigatórios	Podem ser adicionados MCDT's que deverão ser incluídos prescrição do protocolo.
Definir MCDT's opcionais	Podem ser adicionados MCDT's opcionais que podem ou não ser incluídos no protocolo no momento da prescrição.
Definir MCDT's a realizar	A definição dos MCDT's a realizar num protocolo, quer sejam obrigatórios ou opcionais, inclui a selecção do MCDT e a definição dos valores de referência aceitáveis.
Definir valores de referência	O utilizador deve indicar os valores de referência para cada MCDT. Deve ser indicado um intervalo aceitável, representado pela cor verde, e um intervalo de alarme, representado pela cor amarela.
Seleccionar MCDT	O utilizador selecciona o MCDT desejado a partir de uma lista. Inicialmente selecciona o serviço que será responsável pela execução do exame, escolhendo depois um dos exames desse serviço para incluir no protocolo.
Definir medicação	O utilizador define a medicação a ser administrada ao doente no âmbito do protocolo que está a ser definido.
Adicionar grupo	O utilizador adiciona um grupo de medicamentos (cocktail) pré-configurado ao protocolo.

Adicionar medicamento	O utilizador adiciona um medicamento específico ao grupo actual.
Definir actividades	O utilizador define quais os actos médicos que constituirão o protocolo, para além de exames e medicação.
Seleccionar acto médico	É apresentada ao utilizador uma lista de serviços. Depois de seleccionar o serviço, o utilizador pode seleccionar um acto médico executado por esse serviço para ser adicionado ao protocolo.
Definir calendário de actividades	Depois de definidas as actividades constituintes do protocolo, o utilizador recorre a uma grelha para definir qual o calendário de actividades. As linhas representam actividades enquanto as colunas representam dias. É possível marcar quais os dias em que será realizada cada actividade.
Definir permissões	O utilizador tem a possibilidade de definir permissões para o protocolo, indicando quais os médicos ou serviços que o podem prescrever.

5.1.2 Requisitos não-funcionais

Para além dos requisitos funcionais apresentados acima sob a forma de casos de utilização, foram também identificados alguns requisitos não funcionais:

- **Simplicidade de interacção:** dado que os utilizadores finais do sistema serão profissionais de saúde, não se pode assumir que tenham conhecimentos profundos sobre tecnologias de informação. Assim, o sistema deve ser simples de utilizar, garantindo que as suas funcionalidades são facilmente acessíveis para todos os tipos de utilizadores;
- **Consistência com o Processo Clínico:** o módulo deve ter uma interface com o utilizador que garanta a consistência a nível visual e de interacção com a aplicação em que será integrado;
- **Utilização de WPF:** o novo módulo deverá ser desenvolvido completamente em WPF, sem depender de qualquer controlo em *Windows Forms*, com a excepção da *shell* principal;
- **Utilização da infra-estrutura de migração:** a infra-estrutura de migração desenvolvida na primeira parte do projecto deverá ser utilizada, de forma a garantir que o módulo possa ser facilmente migrado da tecnologia CAB para *Prism*;
- **Performance:** o sistema deverá ter um tempo de resposta satisfatório em todas as operações. Cada ecrã do módulo desenvolvido deve ser carregado no máximo em 2 segundos.

5.2 Descrição da arquitectura

Nesta secção pretende-se dar uma visão geral da arquitectura do módulo desenvolvido e da forma como interage com o sistema onde se insere. Para isso, serão apresentadas várias vistas sobre a arquitectura, que permitirão descrever com algum detalhe os aspectos mais relevantes.

5.2.1 Arquitectura lógica

A arquitectura lógica do módulo desenvolvido é necessariamente coincidente com a do sistema em que será inserido. Assim, de um ponto de vista da organização lógica, o sistema pode ser decomposto em três camadas fundamentais, apresentadas na Figura 29.

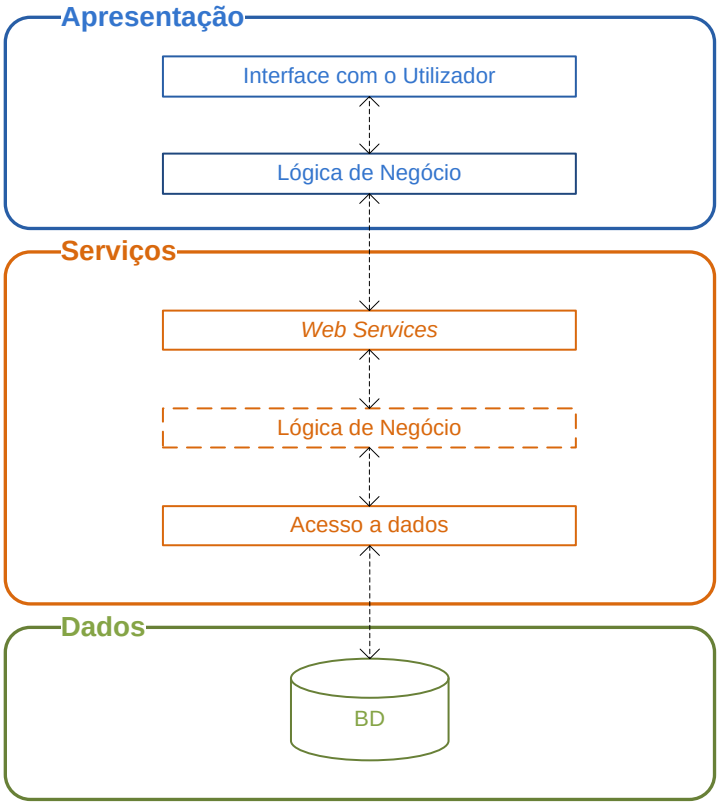


Figura 29 – Diagrama ilustrativo da arquitectura lógica

A camada de dados é constituída pela base de dados e pelos procedimentos que aí estão definidos para as operações CRUD[13]. Estes elementos garantem a persistência e consistência dos dados da aplicação.

A camada intermédia, denominada camada de serviços, pode ser dividida em três constituintes fundamentais:

[13] *Create, Retreive, Update, Delete*

- **Acesso a dados** – esta camada é responsável por garantir a comunicação entre a camada de serviços e a camada de dados. Contém classes que representam as várias entidades no sistema e classes responsáveis por chamar os procedimentos definidos pela camada de dados, mapeando o modelo relacional da base de dados num modelo orientado a objectos, utilizável pelas camadas superiores.
- **Lógica de Negócio –** Esta camada é responsável por realizar quaisquer operações de processamento complexo que não seja desejável executar na camada de apresentação. Apesar de ser não ser utilizada no módulo de definição de protocolos, esta camada é utilizada em algumas circunstâncias na aplicação em que o módulo será integrado, pelo que é apresentada a tracejado na Figura 29.
- ***Web Services*** – os *Web Services* constituem uma interface entre a camada de apresentação e a camada de serviços, permitindo que as duas camadas corram em máquinas diferentes, caso isso seja necessário.

Finalmente, a camada de apresentação é constituída pela implementação da interface com o utilizador e pela maior parte da lógica de negócio da aplicação.

É importante esclarecer o facto de existirem duas camadas de lógica de negócio, de acordo com a Figura 29. A maior parte das regras de negócio são normalmente implementadas na camada de apresentação, já que são geralmente simples e podem ser facilmente executáveis na máquina cliente. Isso evita que sejam realizadas demasiadas comunicações com a máquina onde correm os serviços, reduzindo o tempo de resposta da aplicação. Por outro lado, podem existir algumas situações em que o processamento é relativamente complexo, ou envolve comunicações com outros sistemas. Nesses casos, de forma a não sobrecarregar os clientes, esses processamentos são realizados na camada de serviços, justificando assim a existência de duas camadas de lógica de negócio.

5.2.2 Arquitectura física

O sistema em que o módulo de definição de protocolos será integrado assume a estrutura de um *rich client*. Isto significa que a arquitectura básica é a de um sistema cliente-servidor, em que o cliente possui capacidades de processamento e pode oferecer funcionalidades complexas, independentemente do servidor. A Figura 30 apresenta uma visão da arquitectura física do sistema.

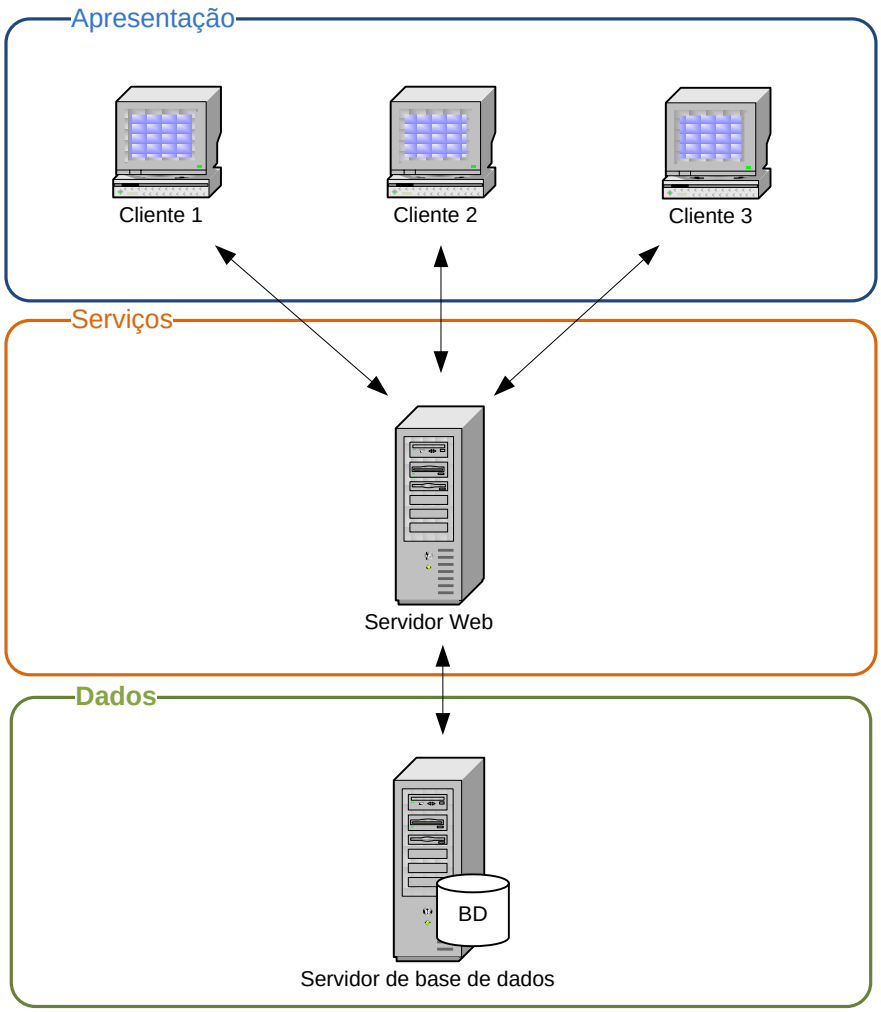


Figura 30 – Diagrama ilustrativo da arquitectura física

Como é possível verificar através do diagrama, a arquitectura física é bastante simples e corresponde em grande medida à arquitectura lógica já apresentada, já que cada uma das camadas pode ser executada em máquinas distintas.

A separação da camada de serviços da camada de apresentação através da utilização de *Web Services* é particularmente relevante, uma vez que permite a reutilização dos serviços por qualquer outra aplicação que o queira fazer.

A existência de um servidor dedicado para a base de dados é principalmente devida a razões de desempenho.

5.2.3 Arquitectura de dados

O módulo desenvolvido assenta numa base de dados partilhada com as restantes aplicações, de modo a poder aceder aos protocolos definidos em Gestão Hospitalar e em Farmácia. A Figura 31 apresenta a estrutura da base de dados.

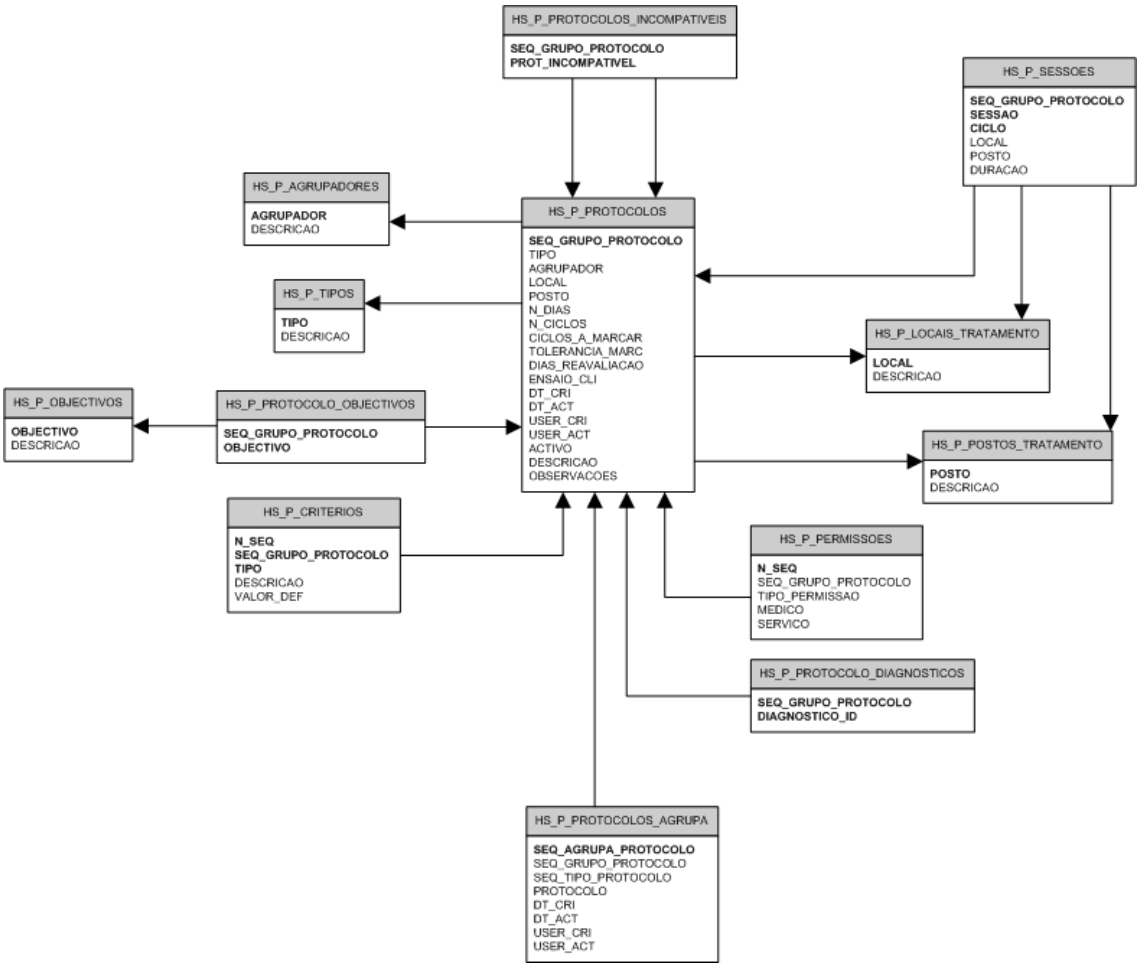


Figura 31 – Diagrama representativo da estrutura da base de dados

Todas as tabelas apresentam o prefixo `HS_P`, que identifica o módulo de definição de protocolos na base de dados.

A tabela `HS_P_PROTOCOLOS` guarda as informações genéricas de cada protocolo. A ligação com os protocolos definidos nas aplicações de Gestão Hospitalar e Farmácia é conseguida através da tabela `HS_P_PROTOCOLOS_AGRUPA`.

Uma descrição mais detalhada das tabelas e dos campos pode ser encontrada em anexo.

5.3 Implementação de um caso de utilização

Nesta secção será descrita detalhadamente a implementação de um caso de utilização, escolhido de forma a exemplificar os aspectos mais relevantes da arquitectura do módulo desenvolvido.

O caso de utilização escolhido foi o primeiro dos que são descritos na Tabela 26, "Configurar protocolo". Constitui o primeiro passo para a definição de um protocolo de tratamento e

envolve várias entidades, sem estar dependente de serviços oferecidos por outras aplicações ou estruturas de dados definidas por outros módulos do Processo Clínico, permitindo analisar a arquitectura do módulo de definição de protocolos sem a complexidade que a interacção com outros módulos acarreta.

Inicialmente será apresentada uma descrição mais detalhada do caso de utilização em análise, seguida por uma visão da implementação nas várias camadas lógicas: dados, serviços e apresentação.

5.3.1 Descrição do caso de utilização – Configurar protocolo

A Tabela 27 apresenta uma descrição detalhada do caso de utilização em análise.

Tabela 27 – Descrição do caso de utilização "Configurar Protocolo"

Descrição:	O utilizador introduz o nome do protocolo e configura as seguintes propriedades: • Tipo e agrupador; • Local e posto de tratamento por defeito para cada sessão de tratamento; • N.º máximo de ciclos; • N.º de sessões por ciclo; • N.º de ciclos a marcar quando o protocolo é prescrito; • Tolerância de marcação – número máximo de dias de folga para o agendamento de actividades; • Ensaio clínico (S/N); • Nº de dias de reavaliação – número de dias necessários para a reavaliação do estado do doente antes da marcação de cada novo ciclo; • Objectivos do protocolo; • Protocolos incompatíveis; • Diagnósticos aos quais o protocolo é aplicável.
Actores:	Médico
Pré-condições:	O médico deve estar identificado perante o sistema. A base de dados contém todos os dados necessários sobre: • Tipos e agrupadores de protocolos; • Locais e postos de tratamento; • Objectivos de protocolos; • Diagnósticos.
Sequência principal:	1. O utilizador indica que deseja definir um novo protocolo. 2. O sistema apresenta ao utilizador uma janela que permite definir as

	várias características do novo protocolo. 3. O utilizador preenche todos os campos obrigatórios. 4. O utilizador grava o protocolo através do botão da barra de ferramentas para esse efeito.
Sequência alternativa:	1. O utilizador indica que deseja definir um novo protocolo. 2. O sistema apresenta ao utilizador uma janela que permite definir as várias características do novo protocolo. 3. O utilizador não preenche todos os campos obrigatórios. 4. O utilizador tenta gravar o protocolo através do botão da barra de ferramentas para esse efeito. 5. O sistema apresenta uma mensagem indicando que não foi possível guardar o protocolo e indicando quais os campos que devem ser preenchidos.
Pós-condições:	Os dados introduzidos são correctamente guardados na base de dados.
Notas:	N.A.

A Figura 32 apresenta um protótipo da interface com o utilizador desenhada para este caso de utilização.

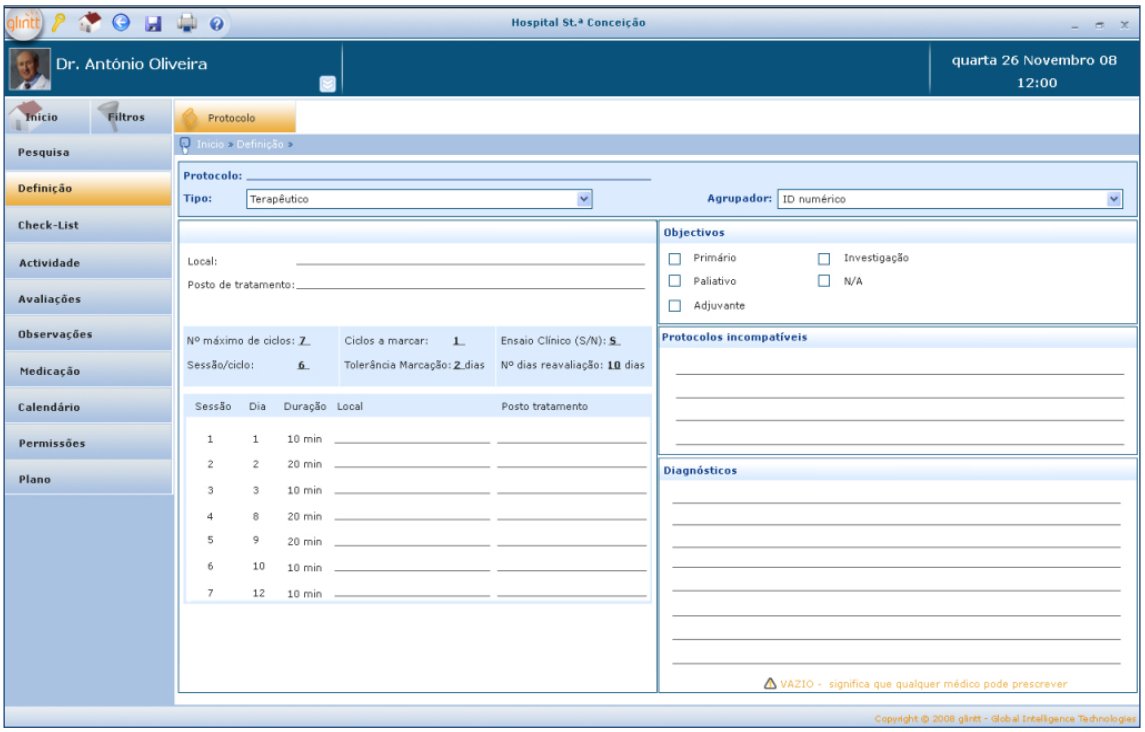


Figura 32 – Protótipo da interface com o utilizador para a configuração de um protocolo

O desenho do protótipo não fez parte do projecto, uma vez que já tinha sido desenvolvido pela *GlinttHS* anteriormente ao início do mesmo.

Apesar de não ser visível no protótipo, ficou acordado que a selecção de protocolos incompatíveis seria feita através de uma árvore, categorizando os protocolos pelo seu tipo e agrupador.

5.3.2 Camada de dados

A implementação da camada de dados baseia-se na estrutura de dados definida em 5.2.3. Foram definidos procedimentos na base de dados para as operações básicas, incluindo a obtenção de todos os tipos de protocolos, agrupadores, objectivos, locais e postos de tratamento. Uma vez que correspondem a simples *queries* a uma só tabela, esses procedimentos foram gerados automaticamente através de *Codesmith* e do gerador de código da empresa. Para operações que envolvem mais do que uma tabela, os procedimentos foram criados manualmente. A Tabela 28 apresenta alguns procedimentos relevantes para o caso de utilização em análise.

Tabela 28 – Exemplos de alguns procedimentos da base de dados

`procedure GetProtocol(`
`  protocolo IN number,`
`  p_result OUT pck_types.external_cursor);`
	Retorna um protocolo, juntamente com as descrições do tipo, agrupador, local e posto de tratamento que lhe estão associados.

`procedure GetObjectivesByProtocol(`
`  protocol IN number,`
`  p_result OUT pck_types.external_cursor);`
	Retorna todos os objectives associados ao protocolo.

`procedure InsertNewProtocObjective(`
`  p_PROTOCOLO in`
`  HS_P_PROTOCOLOS.SEQ_GRUPO_PROTOCOLO%type,`
`  p_OBJECTIVO in`
`  HS_P_OBJECTIVOS.OBJECTIVO%type);`
	Associa um novo objectivo ao protocolo com o código `P_PROTOCOLO`.

`procedure InsertNewProtocol(`
`  p_TIPO in HS_P_PROTOCOLOS.TIPO%type,`
`  p_AGRUPADOR in`
`  HS_P_PROTOCOLOS.AGRUPADOR%type,`
`  p_LOCAL in HS_P_PROTOCOLOS.LOCAL%type,`
`  p_POSTO in HS_P_PROTOCOLOS.POSTO%type,`
`  p_N_DIAS in HS_P_PROTOCOLOS.N_DIAS%type,`
`  p_N_CICLO in HS_P_PROTOCOLOS.N_CICLO%type,`
`  p_CICLOS_A_MARCAR in`
`  HS_P_PROTOCOLOS.CICLOS_A_MARCAR%type,`
`  p_TOLERANCIA_MARC in`
`  HS_P_PROTOCOLOS.TOLERANCIA_MARC%type,`
`  p_DIAS_REAVALIACAO in`
`  HS_P_PROTOCOLOS.DIAS_REAVALIACAO%type,`
`  p_ENSAIO_CLI in`
`  HS_P_PROTOCOLOS.ENSAIO_CLI%type,`
`  p_ACTIVO in HS_P_PROTOCOLOS.ACTIVO%type,`
`  p_DESCRICAO in`
`  HS_P_PROTOCOLOS.DESCRICAO%type,`
`  p_OBSERVACOES in`
`  HS_P_PROTOCOLOS.OBSERVACOES%type`
`, p_USER_CRI VARCHAR2`
`, p_sequence OUT NUMBER`
`);`
	Insere um novo protocolo, retornando o código do protocolo gerado.
	
	(Procedimento gerado automaticamente)

Tal como é possível constatar, o procedimento gerado para inserção de um novo protocolo não insere quaisquer dados que façam parte de relações *muitos-para-muitos*, como é o caso da associação entre protocolos e objectivos. Os procedimentos para inserir os dados dessas associações são criados manualmente, como é o caso do procedimento `InsertNewProtocObjective`.

5.3.3 Camada de serviços

A camada de serviços define as classes correspondentes às entidades necessárias para a implementação do caso de utilização. Essas classes são geradas automaticamente pelo gerador de código, com base na estrutura da base de dados. Neste caso, a entidade central é a que representa um protocolo de tratamento, a classe `Protocol`. A Figura 33 apresenta as relações da entidade `Protocol` com outras entidades relevantes para a configuração de um protocolo.

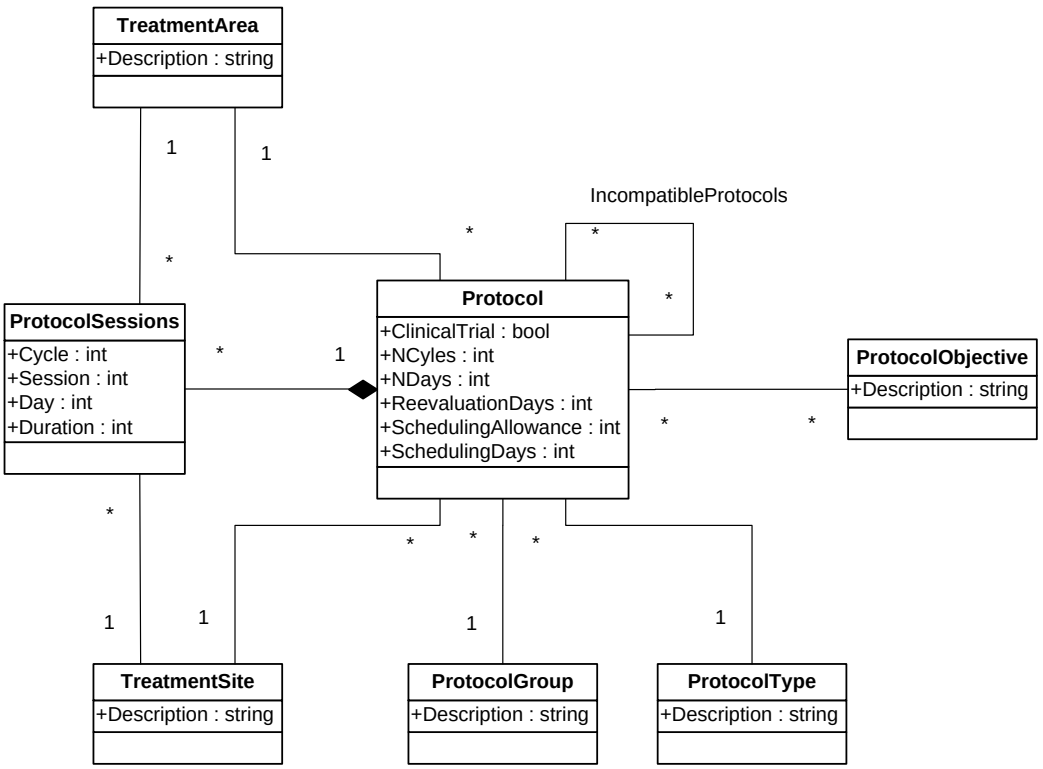


Figura 33 – Diagrama de classes para a configuração de um protocolo

De forma a oferecer a disponibilizar à camada de apresentação as funcionalidades necessárias para a implementação do caso de utilização em estudo, foi definida uma API que seria disponibilizada através de *Web Services*. A Tabela 29 apresenta a assinatura dos métodos que foram implementados.

Tabela 29 – Especificação da API disponibilizada pela camada de serviços

`public ProtocolGroupList GetAllProtocolGroups()`	Retorna uma lista de todos os agrupadores de protocolos existentes no sistema.
`public ProtocolObjectiveList GetAllProtocolObjectives()`	Retorna uma lista de todos os objectivos existentes no sistema.
`public ProtocolTypeList GetAllProtocolTypes()`	Retorna uma lista de todos os tipos de protocolos existentes no sistema.
`public TreatmentAreaList GetAllTreatmentAreas()`	Retorna uma lista de todos os locais de tratamento existentes no sistema.
`public TreatmentSiteList GetAllTreatmentSites()`	Retorna uma lista de todos os postos de tratamento existentes no sistema.
`public ProtocolList GetAllProtocols()`	Retorna uma lista de todos os protocolos existentes no sistema.
`public Protocol GetProtocol(` `  long protocol)`	Retorna um protocolo.
`public Protocol InsertNewProtocol(` `  Protocol obj)`	Insere um novo protocolo.

O último método, `InsertNewProtocol`, merece especial destaque. Tal como foi dito acima, o procedimento `InsertNewProtocol`, definido na camada de dados, apenas insere na base de dados o registo correspondente a um novo protocolo. No entanto, é necessário inserir também dados em outras tabelas, devido às relações *muitos-para-muitos*. Assim, é responsabilidade desse método chamar todos os procedimentos da base de dados necessários para garantir que todos os dados do novo protocolo são correctamente guardados. Uma vez que isto envolve operações em várias tabelas, todas as chamadas a procedimentos são envoltas numa transacção, permitindo cancelar a inserção do protocolo, caso haja algum erro de inserção numa das tabelas.

5.3.4 Camada de apresentação

A interface com o utilizador do módulo de definição de protocolos baseia-se numa estrutura de menus em árvore, disponibilizada por um controlo criado para o efeito, denominado *DistributedTree*. Existia já uma versão desse controlo desenvolvido em *Windows Forms*, utilizado noutros módulos do Processo Clínico. No entanto, para que o módulo de definição de protocolos fosse totalmente desenvolvido em WPF, foi necessário criar uma versão da *DistributedTree* nessa tecnologia. A lógica de interacção do controlo em *Windows Forms* foi separada, possibilitando a sua reutilização na versão WPF do mesmo.

O módulo desenvolvido foi estruturado de acordo com o padrão *Model-View-Presenter*, descrito em 3.3.1.2. Assim, a *view* correspondente ao caso de utilização que aqui se analisa tem

um *Presenter* associado, responsável por carregar os dados necessários para a configuração de um protocolo, chamando os serviços apropriados para esse efeito.

Uma vez que o carregamento dos dados necessários para a *view* pode ser potencialmente demorado, optou-se por realizá-lo de forma assíncrona. Assim, quando a view é inicialmente carregada, é imediatamente apresentada. Nesse momento, é lançada uma nova *thread*, responsável por chamar os *Web Services* apropriados. Entretanto, é apresentada uma janela (*splash screen*) que informa os utilizadores que os dados estão a ser carregados. A Figura 34 mostra o fluxo de execução.

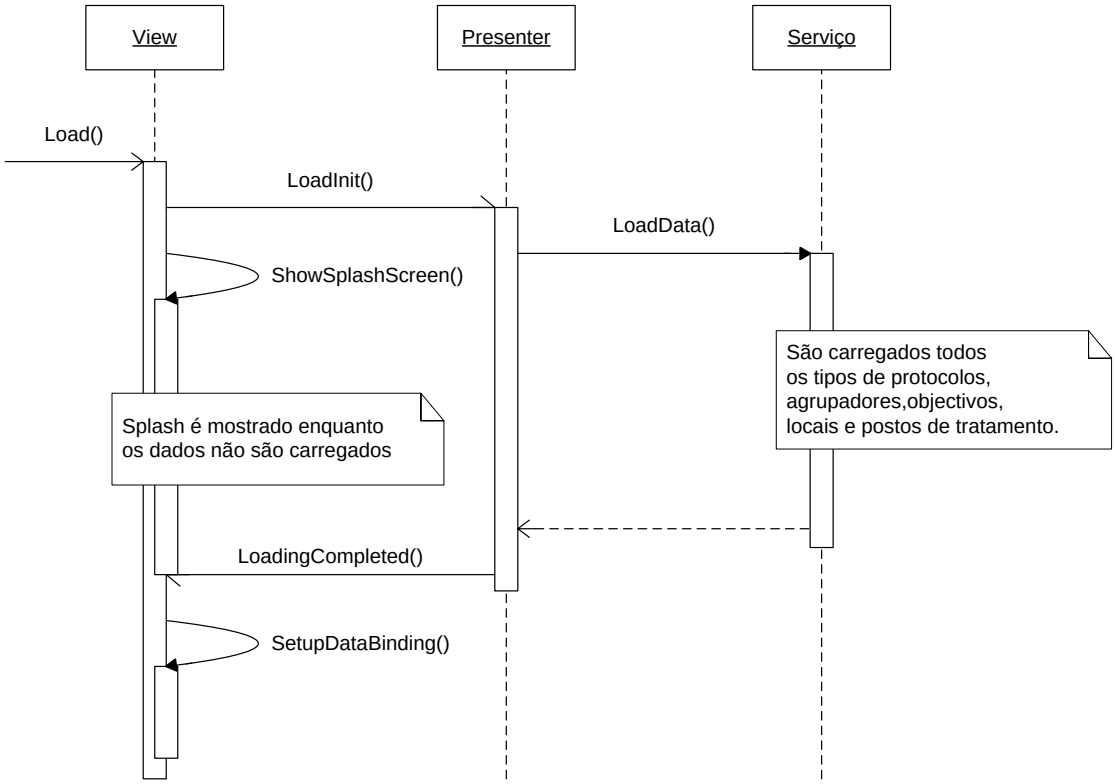


Figura 34 – Diagrama de sequência do carregamento de uma view

Caso não se tivesse optado por realizar o carregamento de forma assíncrona, a *view* não seria apresentada enquanto todos os dados não tivessem sido carregados, não oferecendo qualquer *feedback* ao utilizador durante esse espaço de tempo.

5.4 Estado actual do módulo desenvolvido

Nesta secção serão apresentadas capturas de ecrã exemplificativas do estado actual do módulo de definição de protocolos e do seu aspecto gráfico. A Figura 35 mostra o ecrã que é apresentado quando o utilizador selecciona o módulo de definição de protocolos.

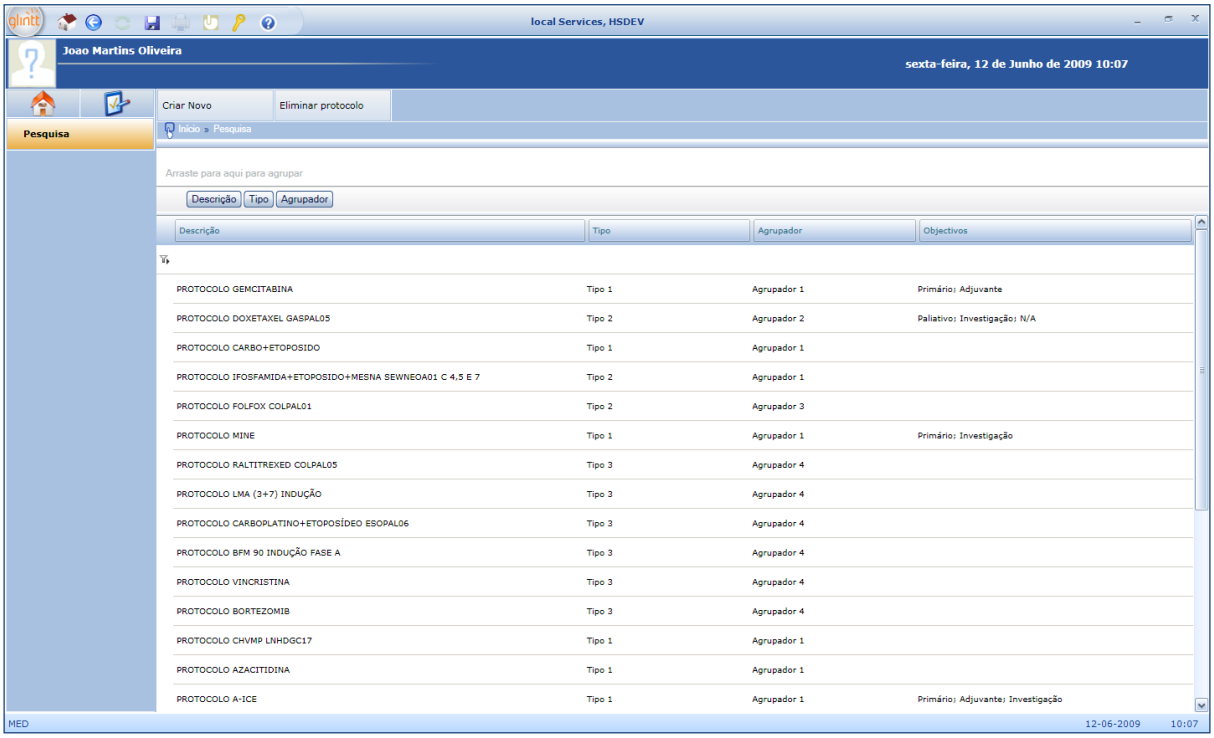


Figura 35 – Listagem de protocolos inicial

É apresentada uma listagem de todos os protocolos existentes no sistema. A listagem pode ser filtrada, ordenada e agrupada (Figura 36).

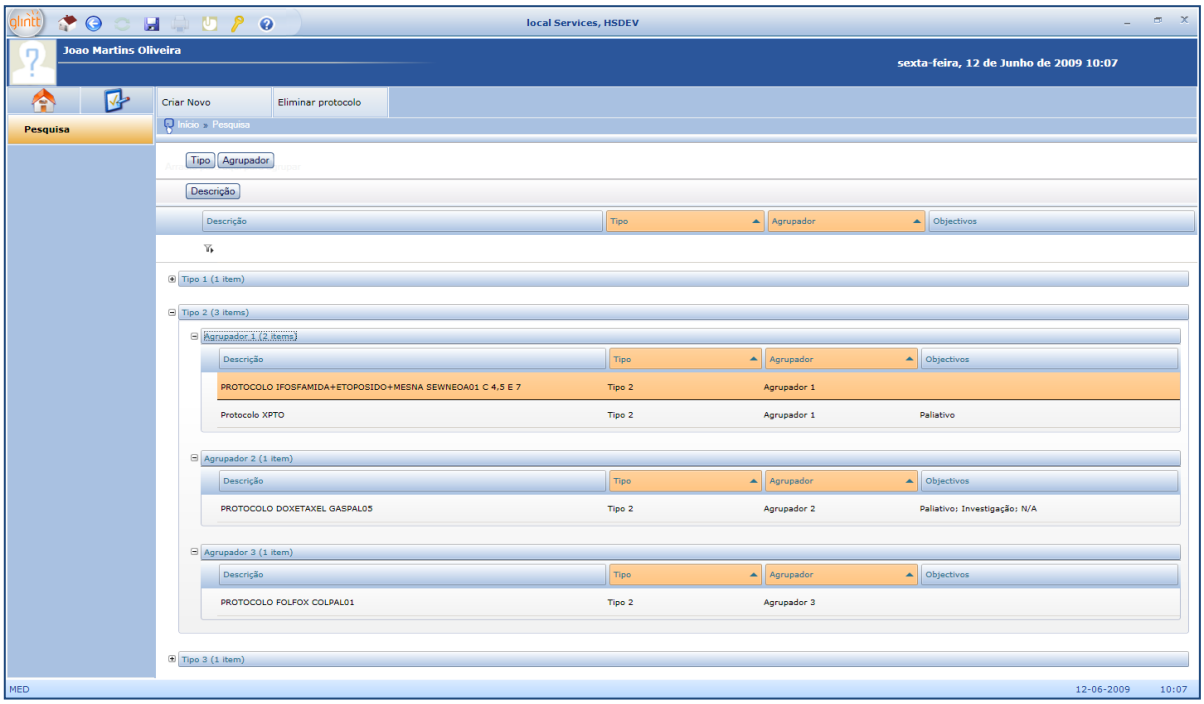


Figura 36 – Listagem de protocolos agrupada por tipo e agrupador

Imediatamente acima da listagem, são apresentados dois botões que permitem criar um novo protocolo ou eliminar um existente. Para editar um protocolo, basta realizar duplo clique sobre a linha correspondente. Será então apresentado o ecrã da Figura 37.

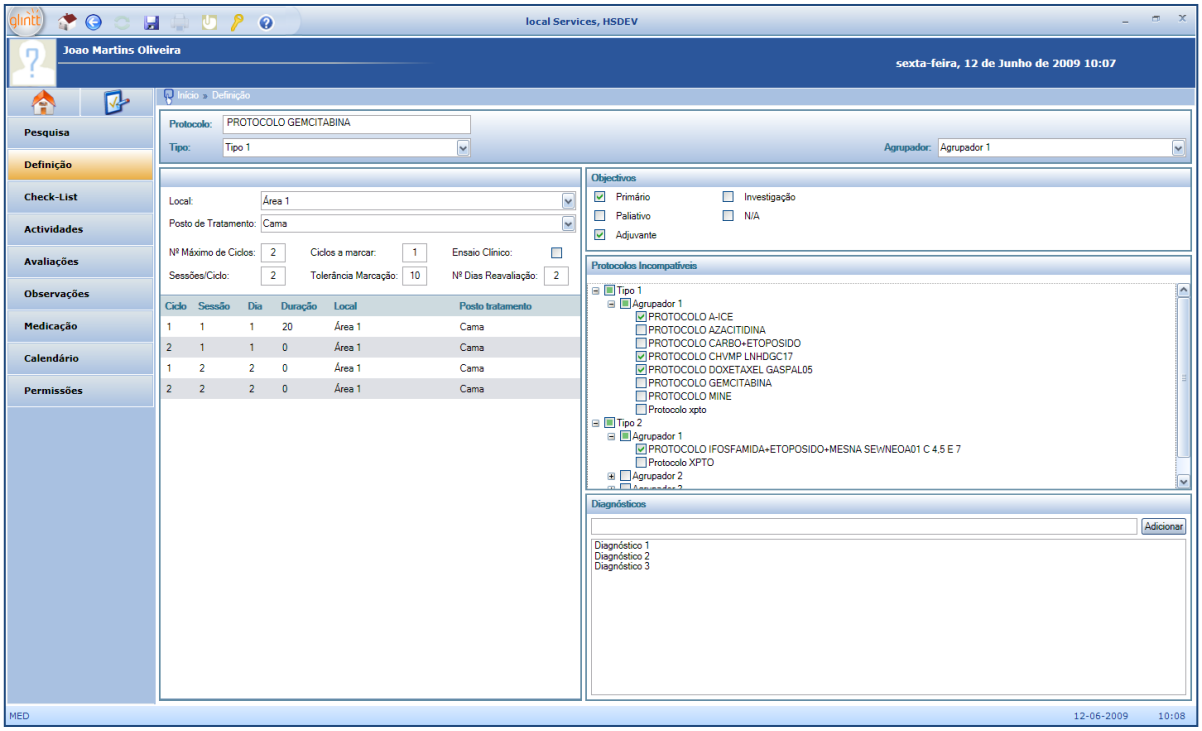


Figura 37 – Ecrã de definição de protocolo

Neste ecrã é possível configurar os principais parâmetros do protocolo, de acordo com o que foi descrito em 5.3. A barra de navegação do lado esquerdo passou a incluir botões que permitem abrir os vários ecrãs para a definição dos restantes parâmetros do protocolo.

A Figura 38 apresenta, a título de exemplo, o ecrã que permite definir os critérios de inclusão e exclusão.

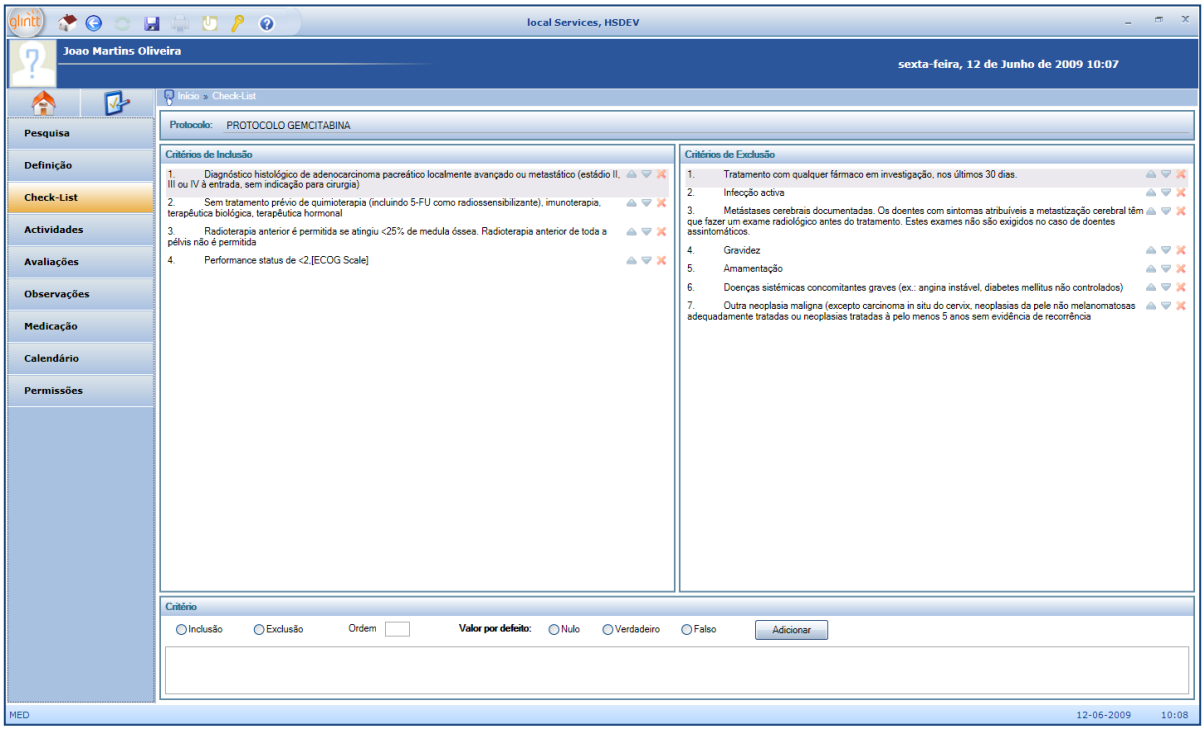


Figura 38 – Ecrã de definição dos critérios de inclusão e exclusão

Neste ecrã é apresentada uma lista ordenada de critérios de inclusão e exclusão. É possível reordená-los e eliminá-los com os pequenos botões que se encontram em cada linha. A adição de novos critérios é feita através da área inferior do ecrã.

Capítulo 6

Conclusões e Trabalho Futuro

Após a apresentação do trabalho, pretende-se com este capítulo apresentar um resumo do trabalho realizado, efectuando uma análise crítica sobre a satisfação dos objectivos que foram traçados inicialmente, assim como uma discussão sobre possíveis desenvolvimentos futuros que possam dar continuidade ao trabalho desenvolvido durante a duração do projecto.

6.1 Satisfação dos Objectivos

No início do projecto que aqui foi apresentado foram traçados dois objectivos principais:

- A criação de uma infra-estrutura de migração que permitisse facilitar a migração entre duas tecnologias de suporte ao desenvolvimento de aplicações compostas (CAB e *Prism*);
- Desenvolver um módulo de definição de protocolos de tratamento oncológico que fosse uma prova de conceito, exemplificando a aplicabilidade da infra-estrutura desenvolvida.

Após um período inicial de estudo das tecnologias envolvidas, foram identificadas as principais diferenças entre CAB e *Prism*, assim como os pontos em comum. Partindo das conclusões dessa fase inicial, foi possível desenvolver a infra-estrutura de migração de forma a encapsular as principais diferenças entre CAB e *Prism* e tornando o desenvolvimento de um módulo efectivamente independente da tecnologia que é utilizada.

Os módulos já desenvolvidos em CAB podem ser alvo de um processo de reengenharia gradual, passando a utilizar a infra-estrutura como intermediário para aceder às funcionalidades de CAB. Durante esse processo, as dependências directas de CAB vão sendo reduzidas, até que os módulos dependam apenas da infra-estrutura de migração. Estando esse processo concluído, a migração pode ser realizada num espaço de tempo bastante curto, já que apenas implica a troca da implementação da infra-estrutura.

Para além de facilitar a migração entre CAB e *Prism*, a infra-estrutura oferece também funcionalidades de integração entre interfaces com o utilizador desenvolvidas em *Windows Forms* e WPF, quer em CAB quer em *Prism*.

O desenvolvimento da infra-estrutura de migração resultou numa mais-valia inequívoca para a *GlinttHS*, demonstrando que é possível realizar a migração entre CAB e *Prism* de forma incremental e oferecendo o elemento de suporte que poderá ser utilizado para o conseguir.

A aplicabilidade da infra-estrutura desenvolvida foi comprovada através do desenvolvimento de um módulo de definição de protocolos de tratamento. Foi possível comprovar que a infra-estrutura é aplicável nas soluções desenvolvidas na empresa e não constitui um factor de redução do desempenho das aplicações.

O módulo de definição de protocolos responde também a uma necessidade identificada pela empresa, pelo que deverá, a curto prazo, fazer parte das soluções que a empresa oferece aos seus clientes, em conjunto com a componente de agendamento que foi desenvolvida num projecto que decorreu em paralelo.

Conclui-se, portanto, que os objectivos traçados para o projecto foram atingidos com êxito.

6.2 Trabalho Futuro

A infra-estrutura de migração desenvolvida disponibiliza as principais funcionalidades requeridas pelos módulos de uma aplicação baseada em componentes. No entanto, dada a complexidade das soluções envolvidas, é inevitável que durante o processo de reengenharia da aplicação de Processo Clínico surja a necessidade de incorporar mais funcionalidades na infra-estrutura. Apesar de a infra-estrutura permitir já o desenvolvimento de módulos sem qualquer dependência de CAB ou *Prism*, como foi comprovado pelo desenvolvimento do módulo de definição de protocolos, ela deve ser vista como uma base sobre a qual serão adicionadas mais funcionalidades à medida que forem sendo necessárias.

A continuidade do trabalho respeitante à infra-estrutura de migração passará pelo prosseguimento do processo de integração da mesma na aplicação de Processo Clínico.

A implementação do módulo de definição de protocolos de tratamento oncológico também deverá será continuada no futuro, já que se trata de um produto no qual a *GlinttHS* tem interesse e que deverá responder a necessidades dos seus clientes.

Como nota final, refira-se que a realização deste projecto constituiu, para o autor, uma experiência profissional enriquecedora, que permitiu não só uma aprendizagem de tópicos mais técnicos, mas também um primeiro contacto bastante gratificante com o mundo do desenvolvimento de software a nível empresarial.

Referências e bibliografia

[Bis99] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, *Legacy Information Systems: Issues and Directions*. IEEE Software, 1999, vol. 16.

[Bro96] A. W. Brown and K. C. Wallnau, *Component-Based Software Engineering*. IEEE Computer Society Press, 1996.

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. Wiley, 1996, vol. 1.

[Chi92] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," in *Software Reengineering*, 1992, p. 54–58.

[Cle02] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 2002.

[Cod08] Codeplex. (2008,) Codeplex - Smart Client Contrib. [Online]. http://www.codeplex.com/scsfcontrib

[Dan07] F. Daniel, B. Benatallah, F. Casati, M. Matera, and R. Saint-Paul, "Understanding UI Integration: A survey of problems, technologies and opportunities," *Internet Computing, IEEE*, vol. 11, no. 3, pp. 59-66, 2007.

[Dem08] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Kehrsatz, Switzerland: Square Bracket Associates, 2008.

[DSo99] D. D'Souza and A. Wills, *Objects, Components, and Frameworks – The catalysis Approach*. Reading, MA: Addison-Wesley, 1999.

[Ecl09] Eclipse Foundation. (2009,) Eclipsepedia - Rich Client Platform. [Online]. http://wiki.eclipse.org/index.php/Rich_Client_Platform

[Fer05] D. F. Ferguson, M. L. Stockton, and M. Nally. (2005) SOA programming model for implementing Web services, Part 6: The evolving component model. [Online]. http://www.ibm.com/developerworks/webservices/library/ws-soa-progmodel6/

[Fow04] M. Fowler. (2004, Jan.) Inversion of Control Containers and the Dependency Injection pattern. [Online]. http://martinfowler.com/articles/injection.html

[Fow99] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Gam95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns : elements of reusable object-oriented software*. Addison Wesley, 1995.

[Gom08] J. Gomes, "Windows Presentation Foundation no Processo Clínico," Relatório de Projecto, 2008.

[Gra87] G. Booch, *Software Components with Ada: Structures, Tools and Subsystems*. Redwood City, CA: Benjamin-Cummings, 1987.

[Gre01] R. Greenwald, S. R., and J. Stern, *Oracle Essentials: Oracle9i, Oracle8i & Oracle8* , 2ªçãth ed. Cambridge, EUA: O'Reilly, 2001.

[Hei01] G. T. Heinemann and W. T. Council, *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

[Joh88] R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22-35, 1988.

[Kaz98] R. Kazman, S. G. Woods, and S. J. Carrière, "Requirements for Integrating Software Architecture and Reengineering Models: CORUM II," in *Working Conference On Reverse Engineering*, Honolulu, HI, 1998, pp. 154-163.

[Mic08] Microsoft Corporation. (2008,) MSDN - Introduction to the Composite UI Application Block. [Online]. http://msdn.microsoft.com/en-us/library/cc540685.aspx

[Mic081] Microsoft Corporation. (2008,) MSDN - Smart Client Software Factory. [Online]. http://msdn.microsoft.com/en-us/library/aa480482.aspx

[Mic082] Microsoft Corporation. (2008, Jun.) MSDN - Dependency Injection. [Online]. http://msdn.microsoft.com/en-us/library/cc707845.aspx

[Mic09] Microsoft Corporation. (2009,) MSDN - Composite Application Guidance for WPF and Silverlight. [Online]. http://msdn.microsoft.com/en-us/library/dd458809.aspx

[Mic091] Microsoft Corporation. (2009) About ActiveX Controls. [Online]. http://msdn.microsoft.com/en-us/library/aa751971(VS.85).aspx

[Moz09] Mozilla. (2009) ActiveX. [Online]. http://support.mozilla.com/en-US/kb/Activex

[Nat07] A. Nathan, *Windows Presentation Foundation Unleashed*. Sams Publishing, 2007.

[Obj99] Object Management Group, *Unified Modelling Language (UML), version 1.3*. 1999.

[Pan07] S. Pant. (2007) Windows Forms in Visual Studio 2005: An Overview. [Online]. http://windowsclient.net/downloads/folders/presentations/entry1316.aspx

[Pob05] J. Pobar. (2005, Jul.) Reflection - Dodge Common Performance Pitfalls to Craft Speedy Applications. [Online]. http://msdn.microsoft.com/en-gb/magazine/cc163759.aspx

[Pot96] M. Potel, "MVP: Model-View-Presenter - The Taligent Programming Model for C++

and Java," 1996.

[Ros98] L. H. Rosenberg. (1998) Software Re-engineering. [Online]. http://cisas.unipd.it/didactics/STS_school/Software_development/SW_re-engineering(NASA%20doc).pdf

[Sam97] J. Sametinger, *Software Engineering with Reusable Components*. Berlin: Springler-Verlag, 1997.

[Som06] I. Sommerville, *Software Engineering*, 8th ed. Addison Wesley, 2006.

[Sun09] Sun Microsystems, "JavaServer Faces Specification," 2009.

Apêndice A. Estrutura da Base da Dados

A estrutura da base de dados é apresentada em forma de tabela. Para cada tabela é apresentada uma curta descrição, seguida de uma listagem de todos os campos que a constituem. Os campos marcados a vermelho fazem parte da chave primária da tabela.

HS_P_PROTOCOLOS:

Definição de um protocolo global.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
SEQ_GRUPO_PROTOCOLO	**ID único do protocol global**	**Não**	**NUMBER**		
TIPO	Tipo de protocolo		NUMBER	HS_P_TIPOS_PROTOCOLO	TIPO
AGRUPADOR	Agrupador de protocolo		NUMBER	HS_P_AGRUPADORES_PROTOCOLO	AGRUPADOR
LOCAL	Local pré-definido para as sessões de tratamento		NUMBER	HS_P_LOCAIS_TRATAMENTO	LOCAL
POSTO	Posto pré-definido para as sessões de tratamento		NUMBER	HS_P_POSTOS_TRATAMENTO	POSTO
N_DIAS	Periodicidade do protocolo		NUMBER		
N_CICLOS	Número máximo de ciclos		NUMBER		
CICLOS_A_MARCAR	Número de ciclos a marcar na primeira prescrição		NUMBER		

TOLERANCIA_MARC	Tolerância de marcação (dias)		NUMBER		
DIAS_REAVALIACAO	Nº. de dias para reavaliação		NUMBER		
ENSAIO_CLI	Se é um protocolo de ensaio clínico (S/N)		VARCHAR		
DT_CRI	Data de criação		DATE		
DT_ACT	Data de activação		DATE		
USER_CRI	Utilizador que criou o protocolo		VARCHAR		
USER_ACT	Utilizador que activou o protoco-lo		VARCHAR		
ACTIVO	Se o protocolo está activo (S/N)	Não	VARCHAR		
DESCRICAO	Descrição do protocolo	Não	VARCHAR		
OBSERVACOES	Observações		VARCHAR		

HS_P_TIPOS:

Definição de tipos de protocolos.

				FOREIGN KEY	
CAMPO	**DESCRIÇÃO**	**NULL?**	**TIPO**	**TABELA**	**CAMPO**
TIPO	**ID único do tipo de protocolo**	**Não**	**NUMBER**		
DESCRICAO	Descrição do tipo de protocolos	Não	VARCHAR		

HS_P_AGRUPADORES:

Definição dos agrupadores de protocolos.

				FOREIGN KEY	
CAMPO	**DESCRIÇÃO**	**NULL?**	**TIPO**	**TABELA**	**CAMPO**
AGRUPADOR	**ID único do agrupador**	**Não**	**NUMBER**		
DESCRICAO	Descrição do agrupador	Não	VARCHAR		

HS_P_LOCAIS_TRATAMENTO:

Definição dos locais de tratamento.

				FOREIGN KEY	
CAMPO	**DESCRIÇÃO**	**NULL?**	**TIPO**	**TABELA**	**CAMPO**

LOCAL	**ID único do local de tratamento**	**Não**	**NUMBER**		
DESCRICAO	Descrição do local de tratamento	Não	VARCHAR		

HS_P_POSTOS_TRATAMENTO:

Definição dos postos de tratamento.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
POSTO	**ID único do posto de tratamento**	**Não**	**NUMBER**		
DESCRICAO	Descrição do posto de tratamento	Não	VARCHAR		

HS_P_SESSOES:

Definição das sessões de um protocolo.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
SESSAO	**Número da sessão no ciclo**	**Não**	**NUMBER**		
CICLO	**Número do ciclo**	**Não**	**NUMBER**		
SEQ_GRUPO_PROTOCOLO	**ID único do protocol global**	**Não**	**NUMBER**	**HS_P_PROTOCOLOS**	**SEQ_GRUPO_PROTOCOLO**
LOCAL	Local pré-definido para as sessões de tratamento		NUMBER	HS_P_ LOCAIS_TRATAMENTO	LOCAL
POSTO	Posto pré-definido para as sessões de tratamento		NUMBER	HS_P_ LOCAIS_TRATAMENTO	POSTO
DURACAO	Duracão da sessão de tratamento, em minutos		NUMBER		

HS_P_OBJECTIVOS:

Definição dos objectivos de protocolos.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
OBJECTIVO	**ID único do objectivo**	**Não**	**NUMBER**		
DESCRICAO	Descrição do objectivo	Não	VARCHAR		

HS_P_PROTOCOLO_OBJECTIVOS:

Associação de protocolos e objectivos.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
OBJECTIVO	ID único do objectivo	Não	NUMBER	HS_P_OBJECTIVOS	OBJECTIVO
SEQ_GRUPO_PROTOCOLO	ID único do protocol global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO

HS_P_DIAGNOSTICOS:

Definição dos diagnósticos a que um protocolo é aplicável.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
DIAGNOSTICO	ID único do diagnóstico	Não	NUMBER	CM_COD_DIAGNOSTICOS	CODIGO
SEQ_GRUPO_PROTOCOLO	ID único do protocol global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO

HS_P_PROTOCOLOS_INCOMPATIVEIS:

Definição da incompatibilidade entre protocolos.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
SEQ_GRUPO_PROTOCOLO	ID único de um protocolo global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO
PROT_INCOMPATIVEL	ID único de um protocolo global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO

HS_P_CRITERIOS_PROTOCOLO:

Definição dos critérios de inclusão e exclusão de um protocolo.

				FOREIGN KEY	
CAMPO	DESCRIÇÃO	NULL?	TIPO	TABELA	CAMPO
N_SEQ	Número de ordem do crité-rio na lista do protocolo	Não	NUMBER		
SEQ_GRUPO_PROTOCOLO	ID único do protocolo global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO
TIPO	Tipo de critério (INCLUSAO ou EXCLUSAO)	Não	VARCHAR		
DESCRICAO	Descrição do critério	Não	VARCHAR		

VALOR_DEF	Valor por defeito (V, F ou N)		VARCHAR		

HS_P_PERMISSOES_PROTOCOLO:

Definição das permissões de um protocolo.

				FOREIGN KEY	
CAMPO	**DESCRIÇÃO**	**NULL?**	**TIPO**	**TABELA**	**CAMPO**
N_SEQ	**Identificador da permissão**	**Não**	**NUMBER**		
SEQ_GRUPO_PROTOCOLO	ID único do protocolo global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO
TIPO_PERMISSAO	Tipo de permissão (MEDICO ou SERVICO)	Não	VARCHAR		
MEDICO	Identificador do médico (caso o tipo seja MEDICO)		NUMBER	SD_PESS_HOSP_DEF	N_MECAN
SERVICO	Identificador do serviço (caso o tipo seja SERVICO)		NUMBER	SD_SERV	COD_SERV

HS_P_PROTOCOLOS_AGRUPA:

Definição dos protocolos agregados (Farmácia e Gestão Hospitalar). O campo PROTOCOLO aponta para tabelas diferentes, consoante o tipo de protocolo a associar. Por essa razão, não tem uma *foreign key* definida.

				FOREIGN KEY	
CAMPO	**DESCRIÇÃO**	**NULL?**	**TIPO**	**TABELA**	**CAMPO**
SEQ_AGRUPA_PROTOCOLO	**ID único**	**Não**	**NUMBER**		
SEQ_GRUPO_PROTOCOLO	ID único do protocolo global	Não	NUMBER	HS_P_PROTOCOLOS	SEQ_GRUPO_PROTOCOLO
SEQ_TIPO_PROTOCOLO	Identificador do tipo de protocolo associado		NUMBER	HS_P_TIPOS_PROTOCOLOS	SEQ_TIPO_PROTOCOLO
PROTOCOLO	Identificador do protocolo associado		NUMBER		

Índice Remissivo