FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Agendamento de Protocolos de Tratamento

José Tiago Pereira de Carvalho

Relatório de Projecto
Mestrado Integrado em Engenharia Informática e Computação

Orientador: Ana Paula Rocha (Professora Auxiliar)

Julho de 2009

Agendamento de Protocolos de Tratamento

José Tiago Pereira de Carvalho

Relatório de Projecto
Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Professor Doutor António Carvalho Brito (Professor Auxiliar da FEUP)

Arguente: Professor Doutor Carlos Costa (Professor Auxiliar da Universidade de Aveiro)

Vogal: Professora Doutora Ana Paula Rocha (Professora Auxiliar da FEUP)

15 de Julho de 2009

Dedicatória

Todo o meu percurso académico e crescimento pessoal só foram possíveis graças ao apoio e dedicação dos meus pais, que sempre estiveram presentes nos momentos mais decisivos da minha vida. Por isso quero agradecer-lhes todos os esforços que fizeram por mim e dedicar-lhes esta tese, sem eles eu não teria conseguido chegar aqui.

José Tiago Pereira de Carvalho

Resumo

A presente tese de mestrado insere-se no contexto da área de saúde e trata toda a problemática associada ao Agendamento de Protocolos. O Agendamento de Protocolos, tal como o próprio nome sugere, consiste na marcação de um conjunto de actos médicos para um determinado paciente. Os actos médicos pertencentes a um Protocolo relacionam-se entre si e pretendem tratar em conjunto uma determinada patologia.

Na tese é investigada a relação existente entre o problema de agendamento de protocolos num hospital e o problema de agendamento de recursos no fabrico de um produto na área da indústria. Esta relação é analisada ao nível da definição da estratégia que permite resolver o problema de alocação dos recursos disponíveis às tarefas a cumprir.

Para além da análise conceptual do agendamento é também efectuada uma análise tecnológica, permitindo definir quais as tecnologias que melhor se adaptam à concretização do projecto proposto. As tecnologias ponderadas concentram-se em duas áreas distintas: a arquitectura da aplicação e a construção das interfaces. Para a arquitectura da aplicação são analisadas as *Frameworks* CAB (*Composite UI Application Block*) e PRISM (*Composite Application*), enquanto na área das interfaces são analisadas as tecnologias WPF (*Windows Presentation Foundation*) e *Windows Forms*.

Em resultado da análise tecnológica efectuada surgiu a necessidade de desenvolver uma Infra-estrutura que permitisse criar aplicações independentemente da *Framework* base CAB ou PRISM. Fundamentalmente, a Infra-estrutura desenvolvida permite endereçar a problemática de integração com as soluções da empresa e ainda serve de plataforma para migração das soluções existentes para uma nova tecnologia.

Por último é importante realçar que foi depositada especial atenção nas questões de usabilidade, capacidade de manutenção e expansibilidade da aplicação. Tal justifica-se pelo facto da área onde o projecto se insere ser bastante volátil ao nível dos requisitos e ser necessário privilegiar a interacção com os utilizadores.

Abstract

This master's thesis urges from the context of healthcare and treats all the problems related to Protocol Scheduling. The Protocol Scheduling, as the name suggests, is about scheduling a set of medical procedures for a given patient. The medical procedures of a Protocol are related and together seek to treat a particular pathology.

In this thesis we research the relationship between the Healthcare Protocol Scheduling Problem and the Manufacturing Resource Scheduling Problem. The goal of this research is try to understand if the strategies used in the industry can be transposed for the hospitals.

Beyond the conceptual research on the schedule subject is also carried out an analysis in technology, allowing defining the technologies that best suit the implementation of the proposed project. The technologies selected address two main areas: the architecture of the application and the development of user interfaces. For the architecture of the application issue are considered two Frameworks: CAB (Composite UI Application Block) and PRISM (Composite Application). For the interface issue are considered two technologies: WPF (Windows Presentation Foundation) and Windows Forms.

As a result of the technology study, a new need emerged: the development of a new Infrastructure for building applications regardless the Framework base used (CAB or PRISM). The Infrastructure has the ability to deal with integration problems, simplifying the connection with the company solutions. Moreover, the infrastructure still supports the migration of existing solutions to a new technology.

At last it’s important to emphasize the special attention placed on issues like usability, maintenance and expansibility of the application. This is justified by the surrounding context of the project, that is quite volatile, and the need to enlarge and clarify interaction with users.

Agradecimentos

Em primeiro lugar queria agradecer a toda a minha família que me tem apoiado e ajudado ao longo de todo este processo de crescimento, com especial ênfase para a minha maninha e a minha afilhada linda.

Se a minha família me pôde acompanhar de perto ao longo de todo este processo houve quem não tivesse essa oportunidade, mas ainda assim desempenha um papel cada vez mais preponderante e decisivo na minha vida. O meu muito obrigado à mulher que me despertou para a vida, és e sempre serás o meu grande amor, Isabel Torres.

No final do meu percurso académico para além do conhecimento o que me resta são os bons momentos que passei com os meus amigos e as noitadas juntos a tentar contornar os desafios que foram surgindo. Para o Fernando Júnior, Francisco Correia, Hélder fontes, Hugo Zenha, João Silva, Micael Queiroz e Pedro Silva um grande abraço e o desejo que tenham uma carreira promissora e cheia de sucesso.

Todo o meu percurso académico também fica marcado pelos docentes que me acompanharam e ajudaram a desenvolver as minhas capacidades, fico-lhes extremamente grato por todo o vosso empenho e dedicação. Não posso deixar de destacar a Doutora Ana Paula Rocha por ter aceite o meu convite para orientadora e ter desempenhado esse papel de forma exímia, contribuindo assim para o sucesso do meu projecto.

Por último mas não menos importante, quero agradecer à empresa onde decorreu o meu projecto, Glintt-Hs, por ter-me proporcionado todas as condições necessárias para conseguir desenvolver o meu trabalho em harmonia. O Eng. Nuno Ribeiro teve um papel importante no meu percurso na empresa, ajudando-me a ultrapassar todas as contrariedades que foram surgindo.

José Tiago Pereira de Carvalho

Índice

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

ACO	Ant Colony Optimization
API	Application Programming Interface
ASP	Active Server Pages
BD	Base de dados
CAB	Composite UI Application Block
CRUD	Create Read Update and Delete
FEUP	Faculdade de Engenharia da Universidade do Porto
FIFO	First In First Out
FSSP	Flow Shop Scheduling Problem
GA	Genetic Algorithms
Glintt - Hs	Global Intelligent Technologies - Healthcare Solutions
GPRS	General Packet Radio Service
HTML	Hyper Text Markup Language
IoC	Inversion of Control
ISO	International Organization Standardization
JSSP	Job Shop Scheduling Problem
KISS	Keep It Simple, Stupid
MA	Memetic algorithms
MIEIC	Mestrado Integrado em Engenharia Informática e Computação
MVC	Model View Controller
MVP	Model View Presenter
OSSP	Open Shop Scheduling Problem
PRISM	Composite Application Library
RDA	Rich Desktop Applications
SA	Simulated Annealing
TS	Tabu Search
UI	User Interface
WPF	Windows Presentation Foundation
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language

Glossário

CAB	A Composite UI Application Block é uma *Framework* desenvolvida pela Microsoft que permite desenvolver aplicações complexas baseadas em *Windows Forms*. Esta *Framework* fornece uma arquitectura que reúne um conjunto de padrões que facilitam a construção de aplicações modulares, de fácil manutenção e extensíveis.
Framework	Uma *Framework* é um conjunto de conceitos que pretende evidenciar o comportamento de um tipo de aplicações, para um dado contexto. Uma *Framework* fornece um esqueleto de uma aplicação que pretende ser o ponto de partida para o seu desenvolvimento.
Infra-estrutura de Migração	Infra-estrutura que permite desenvolver aplicações independentemente da *Framework* base, CAB ou PRISM. A Infra-estrutura é da autoria do José Carvalho e Luís Ponte.
UI	Uma *User Interface* também conhecida por *Human Computer Interface* é a ponte entre o utilizador e o sistema. Através de uma *Interface* os sistemas informáticos conseguem transmitir informação para os utilizadores e receber indicações da sua manipulação.
MVP	O *Model View Presenter* é um padrão que define a organização de uma UI de uma aplicação. Através do MVP é possível construir interfaces mais modulares e de fácil manutenção.
Padrão	Um padrão é um conjunto de conceitos extraídos do conhecimento prático do dia-a-dia que ilustram uma solução para um tipo de problema.
PRISM	*Composite Application* é o nome de uma biblioteca desenvolvida pela Microsoft que pretende agilizar a construção de aplicações em WPF e *Silverlight*. À semelhança de CAB também PRISM oferece uma arquitectura que permite desenvolver aplicações modulares, extensíveis e de fácil manutenção.
Windows Forms	*Windows Forms* foi o nome dado à API de desenvolvimento de aplicações gráficas construída para complementar a Microsoft .Net

	Framework. Através de *Windows Forms* é possível modelar interfaces.
WPF	*Windows Presentation Foundation* é o novo subsistema gráfico do Windows Vista. WPF é usado para construir interfaces mais complexas e ricas (RDA – Rich Desktop Applications).

Capítulo 1

Introdução

O presente relatório tem como objectivo documentar o projecto de final de curso elaborado no contexto do Mestrado Integrado de Engenharia Informática e Computação (MIEIC) realizado na Faculdade de Engenharia da Universidade do Porto (FEUP). O título do projecto é Agendamento de Protocolos de Tratamento e foi realizado em parceria com a empresa Global Intelligent Technologies - Healthcare Solutions (Glintt – HS).

O projecto enquadra-se na área da saúde e pretende agilizar algumas das rotinas diárias do pessoal hospitalar. Ao mecanizar essas tarefas pretende-se que o pessoal hospitalar se foce mais nas tarefas em que realmente são especializados/necessários, contribuindo assim para a qualidade do atendimento aos pacientes.

A rotina de um hospital passa por gerir eficazmente a relação entre os seus recursos e os seus pacientes, tentando assim diminuir o tempo de resposta e aumentar a qualidade dos serviços. É nesse contexto que se enquadra este projecto, pretendendo desenvolver uma aplicação que permita marcar novos protocolos para um paciente e depois agendar os procedimentos que o constituem.

Este capítulo pretende dar um enquadramento geral do projecto, descrever as suas motivações, os seus objectivos e a estruturação geral do documento. Através deste capítulo pretende-se que o leitor seja capaz de perceber a dimensão do projecto, bem como o seu contexto.

1.1 Contexto

Para começar é importante descrever mais detalhadamente a empresa onde decorreu o projecto. Como já foi referido, a empresa denomina-se Glintt – HS e opera na área da saúde. O seu principal mercado alvo são os hospitais portugueses embora os seus negócios já se estejam a alargar ao mercado internacional, nomeadamente, Espanha, Angola e América Latina. As principais actividades da empresa incluem:

- A prestação de serviços, desenvolvimento, manutenção e suporte de aplicações informáticas na área dos Sistemas de Informação, com especial ênfase no domínio das Tecnologias da Saúde e Gestão Hospitalar.
- O licenciamento, implementação, parametrização, formação e consultoria, seja de produtos próprios ou representados.
- A consultoria e gestão de projectos.
- A venda de soluções integradas de Sistemas de informação.

Na figura seguinte (Figura 1.1) é possível verificar a panóplia de soluções informáticas que a empresa já oferece. Pode-se verificar que a empresa já disponibiliza uma solução abrangente que pretende endereçar a maioria das problemáticas encontradas num hospital, oferecendo assim uma solução completa e totalmente integrada.

Figura 1.1 – Soluções disponibilizadas pela empresa

O projecto descrito neste relatório será integrado na área do processo clínico e pretende complementar as funcionalidades oferecidas por um módulo já existente (módulo presente na parte superior direita da figura anterior). O processo clínico é um sistema integrado de informação clínica, que pretende dotar os profissionais clínicos de uma ferramenta de registo de informação clínica. O seu principal objectivo é fornecer o acesso a toda a informação clínica dos doentes necessária para prestar cuidados de saúde de qualidade.

Tendo em conta as tendências do mercado onde a empresa está inserida, em que cada vez mais a oferta é à escala mundial e diversificada, é necessário reunir um conjunto de factores que a realcem face á concorrência. Um dos caminhos possíveis será a aposta na qualidade do software.

A construção de bom software não é uma tarefa trivial e subentende a concordância com um conjunto de factores que o regulam. A qualidade de um produto pode ser medida através de quatro factores preponderantes [Som04]:

- **Flexibilidade** – O software deve ser desenvolvido de tal forma que seja fácil dar resposta a alterações solicitadas pelo cliente. Este factor é crítico na construção de software, porque a alteração dos requisitos de uma aplicação é quase inevitável, quer

seja fruto das alterações ocorridas no ambiente envolvente do negócio quer seja por falta de concordância com as necessidades do cliente.

- **Confiança** – A confiança numa aplicação é uma característica extremamente importante, porque reflecte a capacidade de uma aplicação em não causar danos físicos ou monetários na eventualidade de ocorrer uma falha no sistema. As características que normalmente estão associadas à confiança numa aplicação são a segurança e a fiabilidade. A segurança pode ser vista por dois primas, quer pela probabilidade de não ocorrerem erros, quer pela protecção da informação que a aplicação contém. A fiabilidade é a capacidade de um sistema funcionar correctamente durante um determinado período de tempo e sob um determinado conjunto de condições de operação.
- **Eficiência** – O software não deve desperdiçar os recursos que estão ao seu dispor. A eficiência inclui o tempo de resposta de uma aplicação, o tempo que demora a processar e a quantidade de memória usada, entre outros factores. De nada serve ter uma aplicação que cumpre com os requisitos estabelecidos se depois na prática não fornece respostas em tempo útil.
- **Usabilidade** – Durante a construção de software é necessário ter preocupações ao nível da sua usabilidade, adequando-o ao segmento de utilizadores ao qual se destina. A adaptação do software aos seus destinatários implica o desenvolvimento de uma interface e documentação apropriada. Os utilizadores finais da aplicação devem ser capazes de interagir com esta de forma fácil e intuitiva, dissipando assim possíveis ambiguidades que possam existir.

Os factores anteriores são todos preponderantes para o sucesso de um produto de software, mas no âmbito deste projecto foi dada uma ênfase especial à flexibilidade e à usabilidade. A flexibilidade é fruto do reconhecimento, da empresa e do autor deste documento, da necessidade de acompanhar a volatilidade dos requisitos. Durante o ciclo de desenvolvimento do software é necessário haver uma adaptação gradual aos novos detalhes que vão inevitavelmente surgindo. Os requisitos normalmente alteram-se devido a dois factores:

- **Dificuldades na fase de levantamento de requisitos** – nesta fase, as dificuldades podem surgir por: dificuldade na interacção com o cliente, o cliente não consegue expressar claramente o que pretende; o cliente ainda não sabe muito bem o que quer ou a equipa de levantamento de requisitos considera erradamente que sabe o que o cliente realmente pretende.
- **Alteração do ambiente de negócio** – no decorrer do projecto o ambiente que rodeia o negócio do cliente pode sofrer algumas alterações, sendo necessário reflecti-las na aplicação que está a ser desenvolvida.

Outra justificação para a aposta da empresa na flexibilidade do software surge derivado a estratégia que delineou. A empresa aposta em metodologias ágeis de construção de software, conseguindo assim encurtar o *time to market* [1]dos seus produtos.

[1] Tempo necessário para colocar um novo produto no mercado.

A aposta na usabilidade deve-se mais uma vez ao reconhecimento conjunto, da empresa e do autor, da importância de estabelecer um bom canal de comunicação entre a aplicação e os utilizadores. De que serve ter uma aplicação capaz de responder a todas as necessidades dos seus utilizadores, se no final eles não conseguem interagir com ela. Segundo a norma ISO 9241-11 [ISO98], a construção de aplicações intuitivas e fáceis de utilizar é um grande desafio. A construção de interfaces deve privilegiar a experiência dos utilizadores, simplificando ao máximo todas as funcionalidades que oferece. As interfaces devem ser desenvolvidas de acordo com o público-alvo, o tipo de actividades onde estão inseridas e o contexto de utilização. Uma interface de qualidade caracteriza-se por oferecer uma curva de aprendizagem reduzida e originar uma baixa percentagem de erros por parte dos utilizadores.

1.2 Projecto

Tal como já foi referido na introdução, o projecto incide na área da saúde e pretende agilizar o mecanismo de agendamento de protocolos. O conceito de protocolo na área da saúde é utilizado para ilustrar uma estratégia de combate a uma determinada patologia. Um protocolo não é um conceito trivial de mapear, pelo contrário, um protocolo é composto por um conjunto de actividades diversas, que podem englobar exames, administração de medicamentos, consultas, etc. Para além das actividades, o protocolo também contém um conjunto de restrições que regulam as actividades que o integram. As restrições do protocolo vão desde a dosagem de medicamentos a tomar, a sua periodicidade, frequência, etc.

O projecto reportado neste documento subentende a construção de dois módulos, um módulo de agendamento de protocolos e um módulo de gestão de conflitos. O módulo de agendamento de protocolos consiste numa aplicação que permite agendar as tarefas de um dado protocolo para um determinado paciente. O agendamento de protocolos é uma tarefa complexa, na medida em que implica encontrar um conjunto de recursos que satisfaçam as necessidades das actividades e que ao mesmo tempo respeitem o conjunto de restrições que o protocolo define. Os recursos de um hospital podem ser materiais ou humanos e na maior parte das vezes é preciso conjugá-los para conseguir obter uma vaga.

O módulo de gestão de conflitos entra em plano quando a tentativa de marcar um protocolo não é bem sucedida, derivado à inexistência de vagas que o satisfaçam. Nessas situações é preciso analisar mais cuidadosamente as vagas disponíveis e na pior das hipóteses forçar a marcação de algumas actividades. Os clientes de um hospital são vidas humanas e na maior parte das vezes é simplesmente impossível informá-los que não existem vagas disponíveis ou retardar demasiadamente os seus tratamentos. O impacto de uma decisão desse género poderia ter efeitos nefastos, conduzindo à deterioração do estado clínico de um paciente. Como um hospital não se pode dar ao luxo de ocorrer uma situação dessas é preciso ter uma gestão eficaz e controlada dos recursos existentes, porque eles são demasiadamente preciosos para serem desperdiçados ou mal aproveitados. O módulo de gestão de conflitos deve ser manual e é direccionado para o pessoal administrativo ou enfermeiros.

O módulo de agendamento, pelo contrário, é direccionado para os médicos, que apenas necessitam de uma aplicação que opere automaticamente e forneça uma solução final que respeite as condições que eles indicam.

1.3 Motivação e Objectivos

A motivação deste projecto é tentar encontrar um mecanismo que permita facilitar todo o processo de agendamento de protocolos hospitalares. Considera-se que esta vertente das soluções hospitalares, fornecida pela empresa, tem sido colocada um pouco de parte. Caso se reflicta um bocado facilmente se chega à conclusão que é uma lacuna importante do software existente, uma vez que a base do funcionamento de um hospital são os seus recursos e os pacientes. Através deste projecto pretende-se agilizar todo o processo de agendamento de protocolos, facilitando assim a tarefa do pessoal hospitalar e melhorando os serviços prestados aos pacientes. A principal fonte de lucro de um hospital são os seus pacientes, por isso é natural que a maior parte dos esforços de um hospital sejam na direcção de melhorar a qualidade dos seus serviços. Uma gestão mais eficaz dos recursos do hospital conduz a uma melhoria da sua capacidade de resposta.

O principal objectivo deste projecto é conseguir construir tal aplicação com um elevado grau de usabilidade. O facto de ser difícil traçar o perfil do utilizador que irá interagir com o sistema, conduz a cuidados redobrados na construção da interface. Pretende-se construir uma interface simples e intuitiva, através da qual o utilizador seja capaz de facilmente perceber como interagir. Por outro lado, também se pretende conseguir uma interface poderosa que consiga reunir todas as funcionalidades necessárias para levar a cabo as acções pretendidas. O equilíbrio entre a simplicidade e a conjugação de uma panóplia de funcionalidades num só ecrã não é uma tarefa fácil, perspectivando-se uma longa fase de análise e construção de protótipos.

Em segundo lugar surge a questão da flexibilidade e da integração com os produtos já existentes na empresa. É necessária uma análise cuidada da tecnologia a adoptar para a construção da aplicação. A empresa está numa fase de mudança e pretende avaliar as vantagens e desvantagens de continuar a utilizar a tecnologia actual. Foi proposta uma análise cuidada em relação a esta temática, tendo em atenção os custos inerentes à migração de todas as soluções existentes para uma nova plataforma. A evolução da tecnologia utilizada pela empresa para desenvolver software é inevitável, mas se esta é a altura certa, se este projecto já deverá ser abrangido, são algumas das questões que devem ser respondidas com a análise a efectuar.

Para conseguir dar uma resposta eficaz aos objectivos propostos para o projecto, foi delineado um plano que define as diversas etapas a ser cumpridas.

A primeira etapa consiste no estudo das tecnologias propostas para o projecto. Uma vez concluída a etapa da análise tecnológica, é hora de passar à definição do produto. Nesta etapa é feita uma análise cuidada das necessidades do cliente, existindo uma preocupação redobrada na sua validação. A análise das necessidades do cliente é muito importante e foi efectuada com o auxílio de protótipos, uma vez que são um meio eficaz de esclarecer as necessidades e propor novas formas de resolução do problema. Por último surge a etapa de desenvolvimento e testes, que não é incluída neste projecto, mas estima-se que seja facilitada devido à exaustiva fase de análise.

1.4 Estrutura do Documento

Para além da introdução, este documento contém mais cinco capítulos. No capítulo 2, é feita uma descrição exaustiva do estado da arte, evidenciando todos os temas que foi necessário analisar para conseguir concluir o projecto com sucesso. O capítulo 3 apresenta a análise tecnológica do projecto, incluindo documentação da Infra-estrutura de migração criada. A Infra-estrutura surgiu da necessidade de criar uma plataforma de integração entre a tecnologia utilizada na empresa e a tecnologia considerada pelo autor como a mais indicada para o desenvolvimento do projecto. O capítulo 4 descreve de forma detalhada o problema, indo desde a exposição dos casos de uso do projecto até à apresentação da arquitectura do projecto. O capítulo 5 contém a descrição do estado actual da aplicação, proporcionando uma perspectiva geral do cumprimento dos objectivos. O capítulo 6 apresenta a conclusão, em que é descrito o grau de satisfação dos objectivos propostos e o planeamento de possíveis trabalhos futuros.

Este documento contém ainda quatro anexos. O anexo A é a descrição detalhada do módulo de marcação, pretende-se através deste anexo apresentar as interfaces do módulo e explicar as suas funcionalidades. O anexo B é similar ao anexo A, só que neste caso é apresentado o módulo de prescrição e agendamento de um protocolo. O anexo C apresenta o conjunto de protótipos inicialmente produzidos para o módulo de marcação. Por último surge o anexo D que é a cópia integral de um documento produzido para a empresa. Esse documento contém um conjunto de passos necessários para migrar um módulo de CAB para PRISM e foi produzido durante a fase de análise tecnológica.

Capítulo 2

Estado da arte

Neste capítulo é apresentada toda a investigação efectuada no intuito de conseguir as bases para poder elaborar o projecto. O estado da arte é composto por três partes. A primeira parte foca a análise do problema de agendamento inerente ao projecto e pretende fornecer algumas bases para a descoberta de um possível caminho a seguir. A segunda parte descreve um conjunto de padrões que foram importantes na definição da arquitectura da aplicação. Na terceira e última parte é apresentada toda a informação recolhida sobre as tecnologias ponderadas para utilização neste projecto.

2.1 Agendamento

Este subcapítulo apresenta toda a pesquisa efectuada no contexto da resolução do problema de agendamento de um protocolo. Esta análise dividiu-se em duas fases: numa primeira fase tentou-se verificar se este tipo de problema já era reconhecido pela comunidade científica e se já existia alguma abordagem recomendada; numa segunda fase pesquisou-se um conjunto de algoritmos passíveis de utilização e capazes de dar uma resposta efectiva ao problema.

2.1.1 Definição do tipo do problema

Uma parte do problema proposto para a actual tese de mestrado caracteriza-se pela necessidade de um algoritmo que permita agendar um protocolo médico de tratamento de um doente. O processo de agendamento deve tentar melhorar a qualidade de vida dos pacientes, optimizando a marcação dos protocolos (procurando minimizar o tempo dos pacientes no hospital). Um protocolo médico é constituído por um conjunto de procedimentos que necessitam ser marcados na agenda de um hospital, respeitando as vagas disponíveis e os recursos necessários para os satisfazer. Um protocolo médico é pouco flexível e estabelece um conjunto de dependências entre os procedimentos que o compõem. O tempo necessário para cumprir cada procedimento é estático e inclui o tempo de preparação. Quando um paciente

executa mais do que um procedimento seguido num mesmo recurso, então é necessário subtrair os tempos adicionais de preparação. Um hospital é composto por um conjunto de serviços que por sua vez contém um conjunto de recursos. Cada recurso (humano ou físico) é capaz de satisfazer um conjunto de procedimentos, sendo que, para cada procedimento pode haver mais do que recurso habilitado para o cumprir.

Resultante da procura realizada no intuito de tentar encontrar uma solução para o problema de agendamento de protocolos na área da saúde, encontrou-se um paralelismo com as problemáticas existentes na área da indústria. Numa indústria também é necessário escalonar um conjunto de recursos para produzir um conjunto de produtos, sendo que, à produção de cada produto está a associado um conjunto de tarefas. A esta problemática chama-se agendamento, e assume um papel preponderante no contexto de uma empresa, podendo ser um factor de vantagem competitiva face à concorrência. Esta opinião é também expressa por Blum [Blu02], quando afirma que o agendamento trata da alocação de recursos escassos a tarefas ao longo do tempo. O agendamento é um processo de decisão que pretende optimizar um ou mais objectivos.

O agendamento na indústria é uma tarefa complexa e pode ser enquadrada em diversos contextos. A distinção entre os diversos contextos pode ser efectuada segundo um conjunto de características, sendo as mais relevantes [Xha08]:

- **Distribuição da chegada dos pedidos** – pode ser considerada estática ou dinâmica, dependendo se os pedidos chegam todos ao mesmo tempo, ou se são dispersos temporalmente.
- **Política de gestão do inventário -** um plano pode ser considerado aberto, se os produtos são todos feitos por encomenda, ou fechado, se todos os produtos são feitos para stock.
- **Atributos dos trabalhos –** os planos são classificados como determinísticos, se os recursos e os pedidos estão definidos à priori, caso contrário são classificados como probabilísticos. Outra característica importante dos pedidos, que condiciona a organização da fábrica, é a necessidade de um produto ser processado por uma ou várias máquinas. Essa necessidade depende da constituição de um produto, ou seja, das tarefas que o constituem. Caso os produtos apenas necessitem de uma máquina, então o ambiente da fábrica é considerado de apenas um estado, caso contrário, é multi-estado.
- **Atributos gerais –** o número de máquinas necessárias, o número de pedidos e o percurso efectuados pelos produtos, são mais algumas propriedades importantes na caracterização de um processo de agendamento de uma fábrica.

Da conjugação das características do agendamento do plano de produção na fábrica destacam-se três tipos de problemas, sendo esses os mais referenciados na bibliografia científica: *Flow Shop Scheduling Problem* (FSSP); *Job Shop Scheduling Problem* (JSSP) e *Open Shop Scheduling Problem* (OSSP). Esses tipos evidenciam-se pela sua complexidade (NP-Difícil) e pelo seu enquadramento no plano real.

Aos tipos de problema apresentados pode ser ainda adicionada outra característica (*Flexible*) que permite a duplicação dos recursos existentes, introduzindo o conceito de grupo. Um grupo é composto por um conjunto de máquinas que desempenham tarefas idênticas,

máquinas essas que funcionam em paralelo e permitem aumentar a produtividade de uma operação.

Em todos estes problemas de agendamento existe um objectivo comum, pretende-se alocar um conjunto de tarefas aos recursos existentes, optimizando alguns factores. Os factores a optimizar podem ir desde: maximizar a taxa de ocupação de cada recurso; diminuir o tempo necessário para produzir cada produto; maximizar o número de produtos que é possível fabricar num dia ou diminuir os custos associados ao ciclo de produção de um produto.

Os parágrafos seguintes descrevem de forma sumária os três tipos de problemáticas enumeradas.

Flow Shop Scheduling Problem (FSSP)

No FSSP parte-se do pressuposto que existe um conjunto de máquinas em série. Todos os produtos têm de ser processados por cada uma das máquinas segundo uma ordem preestabelecida, isto é, todos os processos devem iniciar o processamento na máquina um, seguir para a máquina dois e assim sucessivamente. Adicionalmente, cada um dos produtos só pode ser processado numa máquina de cada vez e as máquinas, também só podem processar um produto por instante. As operações nas máquinas são atómicas, não podendo ser interrompidas. Considera-se que o tempo de preparação da máquina para efectuar uma operação já está incluído no tempo da operação. Depois de um produto completar uma tarefa numa máquina deve seguir para a seguinte, ficando momentaneamente à espera de ser processado. Normalmente, as filas de espera das máquinas seguem o princípio de o primeiro a chegar é o primeiro a sair (FIFO – *First In First Out*).

Job Shop Scheduling Problem (JSSP)

Tal como no FSSP, o JSSP também parte do pressuposto de que existe um conjunto de produtos que necessitam de ser processados por um conjunto de máquinas. No JSSP mais tradicional todos os produtos têm de ser processados por todas as máquinas, embora para cada tipo de produto possa estar associado um percurso diferente. Existe uma distinção entre o JSSP em que cada produto pode revisitar uma máquina ao longo do seu percurso e o JSSP que não permite, sendo que no primeiro caso diz-se que permite recirculação. As restantes considerações efectuados para o FSSP também são válidas para o JSSP.

Open Shop Scheduling Problem (OSSP)

O OSSP é um caso especial do JSSP. No OSSP não existe nenhuma sequência predefinida de operações para cada produto, cada produto tem apenas um conjunto de operações que precisa de cumprir. Esta característica alarga drasticamente a dimensão do espaço de procura, face ao JSSP em circunstâncias semelhantes (o mesmo número de máquinas e operações). Mais uma vez todas as considerações efectuadas para o modelo anterior são válidas para este [Pin08].

Conclusões

Com base na apresentação e identificação dos pressupostos dos três tipos de agendamento mais comuns na área da indústria, pode-se concluir qual é o mais adequado ao problema em análise nesta tese.

Considerando que os protocolos médicos são muito vastos, que cada protocolo é constituído por diversas combinações de tarefas e as tarefas entre si possuem uma ordem pré-estabelecida, o processo de agendamento que melhor caracteriza esta situação é o JSSP.

2.1.2 Análise de Algoritmos para o Problema de Agendamento

Tendo em conta o processo de agendamento seleccionado no subcapítulo anterior (JSSP), são apresentadas e analisadas algumas das propostas feitas pela comunidade científica para a resolução do agendamento em JSSP.

Antes de escolher qualquer tipo de algoritmo para tentar solucionar o problema, é necessário definir um conjunto de regras que regulam o problema. Por exemplo, antes de poder associar um produto aos recursos é necessário definir o tipo de recursos que ele necessita e qual a sua ordem. Tal como, também é preciso indicar que tarefas cada um dos recursos é capaz de executar e qual a sua capacidade. Este tipo de regras define o espectro do problema e permite ajudar o algoritmo a validar as soluções. Normalmente, este tipo de regras estão definidas na função de avaliação, cujo papel é avaliar a qualidade das soluções. A qualidade de uma solução pode ser vista por dois prismas: a consonância com as restrições do problema e a valorização das características que se pretende optimizar.

Os algoritmos citados na literatura para a resolução de problemas de agendamento são normalmente algoritmos de optimização. Nos problemas de optimização já existe inicialmente uma solução para o problema, solução essa que o algoritmo tenta optimizar. As soluções alternativas são validadas através de uma função de avaliação que mede a qualidade da solução para o problema a resolver.

Um dos caminhos possíveis para resolver os problemas de optimização de um JSSP pode ser a utilização de meta-heurísticas. Uma meta-heurística é um processo iterativo que coordena um conjunto de heurísticas, tendo por objectivo produzir uma solução de maior qualidade [OL96].

A utilização de meta-heurísticas facilmente se justifica pelo facto de a optimização de um JSSP ser um problema de complexidade NP-difícil. A utilização de métodos exactos para resolver problemas de optimização combinatória na maior parte das vezes requer tempos de processamento impraticáveis [Xha08]. Um algoritmo normal não consegue alcançar uma solução num espaço de tempo razoável.

As meta-heurísticas são uma estratégia de alto nível, que permite balancear uma exploração diversificada (tentar explorar todo o espaço de procura) com uma exploração mais focada e especializada (intensificar a procura num local). Este equilíbrio é importante porque permite identificar rapidamente as zonas em que existem soluções de melhor qualidade, restringindo assim o espaço de procura. Uma vez identificadas as zonas com maior probabilidade de encontrar uma boa solução é possível intensificar a procura nessas zonas e alcançar uma melhor solução [Xha08].

2.1.2.1 Estudos apresentados pela comunidade científica

Algumas das soluções apresentadas pela comunidade científica que utilizam a abordagem sugerida são enumeradas nos pontos seguintes:

- Algoritmos Genéticos [YZFGW08] [FB91] [Xha08]
- Algoritmos Genéticos e Pesquisa Tabu [ZSG08]
- Arrefecimento Simulado [KGR95] [Xha08]
- Algoritmos Meméticos [CF08] [Xha08]
- *Ant Colony Optimization* [MZ99] [PXH04] [Xha08]

De seguida é apresentada uma breve descrição da lógica inerente a cada um dos algoritmos enumerados.

Algoritmos genéticos

Os algoritmos genéticos (*Genetic Algorithms* - GA) são uma técnica probabilística de pesquisa, que tem as suas raízes nos princípios da genética. John Holland é o nome do pai dos GA, ele desenvolveu os seus princípios base por volta dos anos 60 e 70. Segundo Falkenauer e Bouffouix [FB91], a grande inovação de Holland foi seguir os princípios da natureza na sua busca por espécies cada vez mais adaptadas ao ambiente. Os GA são inspirados nas capacidades da natureza de evoluir os seres vivos, adaptando-os ao seu meio ambiente.

Os algoritmos genéticos são caracterizados como um modelo computacional evolutivo. O modelo é constituído por uma população, em que cada elemento é representado por um cromossoma. Um cromossoma representa uma possível solução para o sistema e deve respeitar as restrições do problema. O modelo evolui segundo os princípios da evolução natural, em que os elementos da população são cruzados dando origem a novos elementos que constituem uma nova população, população esta que constitui novas soluções para o problema. O algoritmo parte do pressuposto que a consecutiva combinação da sua população conduzirá a uma melhor solução.

Para que a combinação da população traga frutos ela não pode ser feita de forma leviana, ou pelo menos a maior parte das vezes não. Na figura seguinte (Figura 2.1) pode-se ver um diagrama ilustrativo do funcionamento do algoritmo.

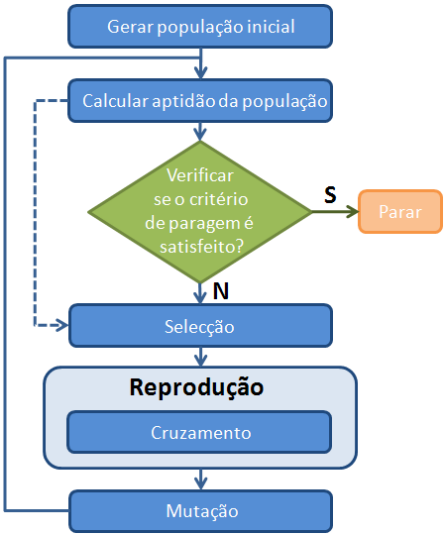

Figura 2.1 – Ilustração do funcionamento de um Algoritmo Genético

Em cada iteração do algoritmo (Figura 2.1) um conjunto de operadores é aplicado a alguns elementos da população, construindo assim os elementos da próxima geração. Normalmente, o algoritmo utiliza dois tipos de operadores: o cruzamento e a mutação. O cruzamento é utilizado na geração de novos elementos pela combinação de dois ou mais elementos da população, conjugando a sua informação. A mutação, tal como o próprio nome sugere, é aplicado de forma individual e permite modificar ligeiramente um elemento.

O elemento chave deste algoritmo é a selecção dos elementos da população baseada na propriedade de aptidão. A propriedade de aptidão é associada a um elemento e representa a qualidade da sua solução, evidenciado a sua proximidade de uma solução óptima. Elementos com um maior valor de aptidão têm uma maior probabilidade de serem seleccionados para uma nova iteração da população. Este comportamento baseia-se no princípio da sobrevivência, seguindo a evolução natural do mundo biológico [Blu02] [CZ01].

Pesquisa Tabu

A Pesquisa Tabu (*Tabu Search* - TS) é uma meta-heurística baseada em pesquisa local. A pesquisa local básica é normalmente conhecida por melhoramento iterativo, uma vez que, dentro do espaço de procura (vizinhança de uma solução) um movimento só é permitido se melhorar a solução.

A principal ideia por trás da TS é muito simples e consiste em apenas adicionar o conceito de memória a um algoritmo normal de melhoramento iterativo. O mecanismo de memória permite forçar a exploração de novas áreas no espaço de procura, tentando assim evitar os ciclos ou os mínimos locais. Na memória do algoritmo são guardados os últimos movimentos efectuados, evitando assim que em futuras decisões se volte a percorrer os mesmos locais. A memória do algoritmo é finita e guarda apenas os N últimos passos, sendo necessário descartar a cada iteração os movimentos mais antigos. Na figura seguinte (Figura 2.2) é ilustrado todo o comportamento do algoritmo.

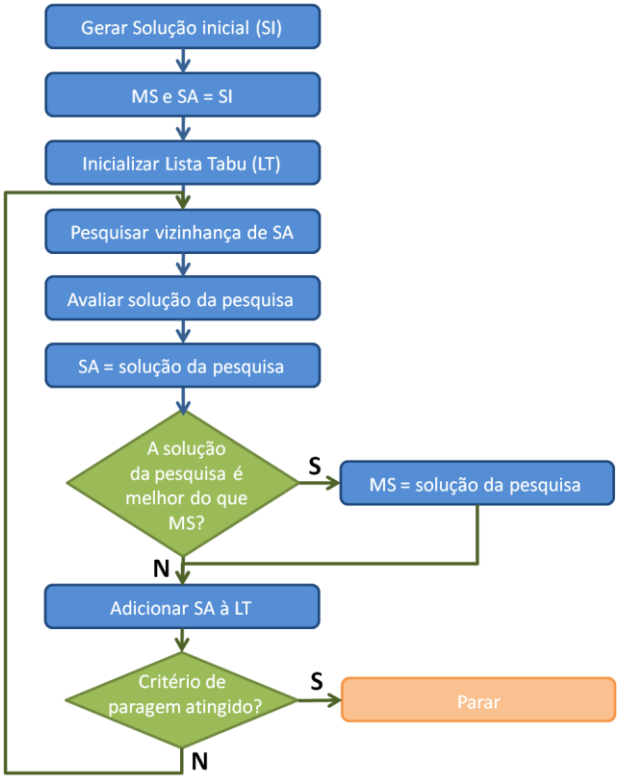

Figura 2.2 – Ilustração do funcionamento do algoritmo arrefecimento Simulado

A abordagem normal do algoritmo é pesquisar a vizinhança do ponto actual (colocando de parte os elementos que já foram visitadas nas N iterações anteriores) e escolher o elemento que contém a melhor solução. Depois de determinado o ponto, este deve ser adicionado à memória do algoritmo (para evitar ser evitado em pesquisas futuras). Caso o valor da nova solução seja superior ao melhor registo até ao momento, então o seu valor deve ser guardado. Todo este processo contínua enquanto a condição de paragem do algoritmo não se verificar. O critério de paragem verifica se a solução actual é satisfatória e previne a entrada em ciclo infinito do algoritmo [Blu02] [AL03].

Arrefecimento simulado

O arrefecimento simulado (Simulated Annealing - SA) é considerado uma das mais velhas meta-heurísticas e muito provavelmente um dos primeiros algoritmos a incluir uma estratégia para contornar os óptimos locais. O raciocínio adoptado por este algoritmo permite a adopção de movimentos que possam conduzir a soluções locais piores do que a actual, permitindo assim escapar a óptimos locais. A probabilidade de efectuar esses movimentos diminui ao longo do tempo durante a pesquisa.

O algoritmo utiliza duas estratégias bem delineadas para conseguir alcançar o seu objectivo, por um lado faz uma pesquisa aleatória, por outro lado, faz um melhoramento iterativo. A pesquisa aleatória satisfaz a necessidade de contornar óptimos locais e é caracterizada através de um parâmetro T, geralmente conhecido por parâmetro da temperatura. O parâmetro T é a probabilidade que o algoritmo tem de admitir uma solução pior do que a actual durante o processo de melhoramento iterativo. O valor do parâmetro T vai diminuindo á

medida que a pesquisa vai avançando, passando o algoritmo a ter fundamentalmente um comportamento de melhoramento iterativo. O melhoramento iterativo consiste simplesmente em pesquisar a vizinhança de uma solução e guardar a melhor solução registada até ao momento. O SA obriga a definição de uma condição de paragem, que pode ser por exemplo o número de iterações efectuadas. Na figura seguinte (Figura 2.3) é possível visualizar um diagrama ilustrativo do funcionamento do algoritmo.

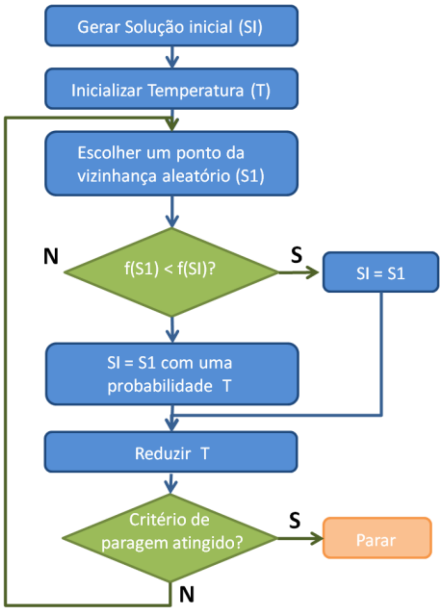

Figura 2.3 – Ilustração do funcionamento do algoritmo Arrefecimento Simulado

O nome do algoritmo provém da analogia feita entre o comportamento deste algoritmo e o processo de arrefecimento de metais e vidro. O arrefecimento de metais e vidro assume uma configuração final de baixa energia interna, caso o arrefecimento seja lento e controlado, permitindo uma reorganização homogénea dos átomos (reduz os defeitos no material) [Blu02] [MF04].

Algoritmos com meta-heurística

De entre os algoritmos identificados os que se enquadram nas meta-heurísticas são os algoritmos Meméticos e *Ant Colony Optimization*. Nos parágrafos seguintes, á semelhança dos algoritmos que foram apresentados anteriormente, é apresentada uma breve descrição do seu comportamento.

Algoritmos Meméticos

Em primeiro lugar é fundamental perceber o significado da palavra *meme*, porque só a partir dai se pode começar a perceber as raízes dos Algoritmos Meméticos (*Memetic Algorithm* - MA). R. Dawkins apresentou a sua definição através de um dos seus livros "The Selfish Gene" [Daw90], onde refere que o *meme* representa a unidade de informação que é transmitida entre gerações. Este conceito é similar ao do gene nos algoritmos genéticos, sendo a principal

diferença entre eles o facto de que o *meme* surge no contexto da evolução cultural, enquanto o gene emula a evolução biológica [Mos89].

A lógica dos MA é muito parecida com as do GA. Em ambos os casos a ideia principal passa pela evolução de uma população, transmitindo propriedades da geração anterior para a seguinte. A principal diferença reside na forma como é feita essa evolução, enquanto no caso dos GA a evolução só surge a partir do cruzamento de elementos da população, nos MA existe também um evoluir solitário de cada um dos indivíduos. Uma definição sumária dos MA é a referida por Moscato [Mos89]: "Memetic algorithms is a marriage between a population-based global search and the heuristic local search made by each of the individuals.". Na figura seguinte (Figura 2.4) é apresentado um diagrama ilustrativo do funcionamento de um MA.

Figura 2.4 – Ilustração do funcionamento de um Algoritmo Memético

Dada a representação de um problema de optimização é criado um conjunto de elementos que representam a população inicial do problema. A população inicial pode ser gerada de forma aleatória ou pode ser utilizada uma heurística para o fazer. Uma vez criada a população inicial passa-se para a fase da pesquisa local, em que cada elemento tenta melhorar a sua solução. Para efectuar a pesquisa recorre-se a um algoritmo de pesquisa local que visita a vizinhança do elemento tentando melhorar a solução existente (até um determinado nível). Terminada a fase de melhoramento local, passa-se para a fase da pesquisa global. A pesquisa global é composta por duas componentes, a interacção cooperativa e a competitiva. A interacção competitiva é similar ao processo de selecção utilizado nos GA (através do valor da aptidão), enquanto a interacção cooperativa pode ser equiparada ao mecanismo de cruzamento utilizado em GA. Na verdade a interacção cooperativa pode ser modelada através de um qualquer processo que permita a geração de novos indivíduos. No fundo a interacção cooperativa é apenas uma forma de promover a troca de informação. Todo este processo (excepto a fase inicial) é repetido iterativamente até que um critério de paragem seja satisfeito. Um exemplo de um critério de paragem valido é o valor de aptidão da população alcançar um valor esperado [Mos89] [CF08] [KG02].

Ant Colony Optimization

Ant Colony Optimization (ACO) é uma meta-heurística proposta inicialmente por Marco Dorigo [DS04A]. A ACO inspirou-se no comportamento real das formigas durante a busca de alimentos. O comportamento das formigas durante a fase de busca e recolha de alimento foi documentado por Jean Louis Deneubourg e permitiu compreender a sua estratégia na definição do caminho mais curto entre o local onde o alimento é recolhido e o ninho. Enquanto as formigas caminham entre a fonte da comida e o ninho, ou vice-versa, depositam no caminho por onde passam uma substância denominada feromona. O intuito dessa substância é indicar ao resto da colónia que alguém já passou por ali. Quando as formigas estão prestes a decidir por que caminho seguir escolhem o que tem maior concentração de feromona (decisão probabilística). Este comportamento promove a cooperação entre a comunidade e permite estabelecer mais facilmente uma rota mais curta.

Tendo em conta que as soluções calculadas pelas formigas podem não ser localmente óptimas, algumas implementações do ACO permitem que as formigas melhorem as suas soluções através de algoritmos de pesquisa local. O algoritmo básico de ACO consiste em quatro passos:

- Em primeiro lugar estabelecem-se os parâmetros e inicializam-se os caminhos de feromona.
- Em segundo lugar geram-se N soluções usando a combinação de caminhos de feromona e as restrições do problema.
- Em terceiro lugar, aplica-se um algoritmo de pesquisa local às soluções obtidas.
- Por último, se o melhor caminho definido no passo anterior é melhor do que todas as soluções geradas até ao momento, então substitui-se a melhor solução por essa. No último passo também está subjacente a actualização dos valores da feromona nos caminhos percorridos e o voltar ao passo dois para uma nova iteração do algoritmo.

Na figura seguinte (Figura 2.5) é possível ver uma ilustração do algoritmo ACO e verificar os passos enumerados anteriormente.

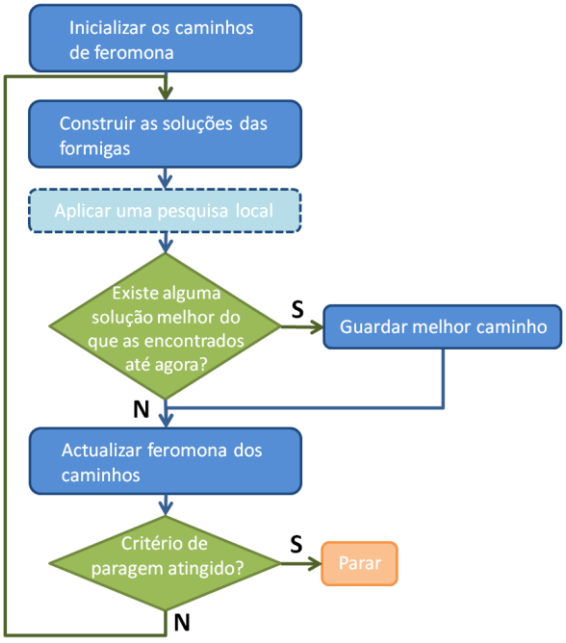

Figura 2.5 – Ilustração do funcionamento do algoritmo *Ant Colony Optimization*

Na maior parte das implementações do algoritmo as formigas são desenvolvidas de forma a assegurar que a construção das soluções para o problema só é feita com passos válidos (que respeitem as restrições), assegurando desde logo a validade da solução [Pin08] [Blu02].

2.1.2.2 Conclusões

Considerando toda a análise de algoritmos apresentada no decorrer deste subcapítulo é importante referir que não é possível identificar uma solução que fosse razoavelmente melhor que as restantes. O factor preponderante dessa incapacidade surge pelo facto de o autor não ter tido oportunidade de colocar em prática todo o conhecimento adquirido, podendo assim constatar qual era a solução que melhor se adequava ao problema.

O trabalho com algoritmos e meta-heurísticas é um trabalho ingrato, na medida em que, na maior parte dos casos é necessário testa-los para verificar a sua aptidão na resolução de problemas.

2.2 Padrões

Este capítulo pretende demonstrar a importância dos padrões e apresentar toda a pesquisa desenvolvida nesse segmento, no contexto do actual projecto. Como ponto de partida serão apresentadas algumas definições recolhidas ao longo do estudo:

- "The definition I use for pattern is an idea that has been useful in one practical context and will probably be useful in others."[Fow96]
- "As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant."[Ale79]
- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"[ISC97]
- "A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it."[BMRSS96]
- "Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context." [GHJV97]

Todas as definições do conceito de padrão apresentadas são concordantes numa questão, que é o facto de um padrão descrever uma possível solução para determinado tipo de problema. O facto de o contexto onde o problema está inserido ser relevante ou não para a sua escolha conduz a outra discussão. Segundo Fowler [Fow96], "Se eu analisar um padrão que eu penso ser útil em mais domínios, do que o original, quão abstracto devo eu considerar esse padrão? O problema de abstrair o problema para além do seu domínio original, é que eu não posso estar certo da sua validade.". Na mesma ordem de ideias surge também o seguinte comentário, "Especificar o contexto correcto de um padrão é difícil. Nós consideramos que é praticamente impossível determinar todas as situações, tanto gerais como específicas, em que um padrão pode ser utilizado. Uma aproximação mais pragmática é listar todas as situações conhecidas onde o padrão que está a ser analisado pode ser utilizado." [BMRSS96]. A questão levantada por estes autores é bastante pertinente, na medida em que os padrões advêm da experiência obtida em casos práticos e pretendem documentar o sucesso de uma abordagem. Alguns autores destacam claramente a importância desta característica, classificando-a como um dos pontos chaves para a utilização dos padrões [Fow96] [DHJV93].

No fundo os padrões têm conseguido angariar cada vez mais adeptos devido à sua característica de reutilização. Os padrões são uma forma eficaz de documentar a experiência prática dos especialistas e depois reutiliza-la. Buschmann et al. [BMRSS96] e Gamma et al. [GHJV97] também são da opinião de que não é necessário tratar um problema de raiz quando se pode aproveitar experiências passadas e aprender com os erros ou sucesso provenientes delas.

Tratar o problema de base, na maior parte dos casos, é um desperdício de tempo. Mais vale apostar na análise das soluções disponíveis para aquele tipo de problema e tentar verificar se existe alguma que se enquadre. Tal como Fowler [Fow96] afirmou, "A coisa mais importante é

estar consciente que eles são um ponto de partida e não um destino", ou seja, é preciso ter consciência que um padrão não fornece uma resolução concreta para um problema, mas antes um conjunto de orientações para o seu solucionamento.

Um dos pontos positivos de seguir um padrão, partindo do pressuposto que é adequado para ao problema, é que a solução descrita já foi implementada e testada, logo existem algumas garantias de que funciona [BMRSS96].

Ao encontrar um padrão que se pensa ser adequado ao problema em questão é necessário pô-lo em prática e verificar se realmente cobre todas as especificidades do problema. Durante o processo de implementação do padrão podem ocorrer algumas contrariedades, alguns detalhes do problema podem não ser idênticos aos descritos no padrão. Nessas situações é preciso tentar adaptar o padrão ou pesquisar um novo [Fow96].

Nem sempre os padrões existentes cobrem totalmente as necessidades, mas é possível através da sua análise extrapolar alguns conceitos para resolver problemas análogos [Fow96].

Os padrões também permitem captar algumas especificidades dos problemas que as vezes não são triviais. Ao seguir um padrão é possível, na maior parte dos casos, adoptar uma abordagem que torne os sistemas mais flexíveis, eficientes e fáceis de manter. Esta componente é uma faca de dois gomes, já que é preciso escolher convenientemente o padrão, porque no limite introduz-se uma complexidade desnecessária [Fow96].

Segundo Coplien [Cop92], "Os projectos falham por falta de soluções comuns, apesar de serem utilizadas as últimas tecnologias.". Esta expressão resume bem tudo o que dito até agora, já que ela personifica bem a importância de encontrar uma boa solução para um problema, ou seja, encontrar um padrão que se enquadre no problema proposto.

No contexto deste trabalho os padrões identificados pretendem ajudar na estruturação das diversas identidades que compõem o problema, definindo o seu papel, as suas funções e descrevendo a forma como comunicam entre si.

Um padrão é composto essencialmente por 4 elementos [GHJV97]:

- **Nome** - é usado para identificar um determinado problema, a sua solução e as consequências.
- **Problema** - descreve quando deve ser aplicado o padrão, o problema a que está associado e o seu contexto (na maior parte dos casos é difícil definir o contexto, por isso opta-se por criar uma listagem de todas as situações conhecidas em que o problema se enquadra). Na descrição do problema são definidos os requisitos que a solução deve cumprir, restrições que devem ser consideradas e propriedades que a solução deve ter.
- **Solução** - descreve os elementos que compõem o projecto, a sua relação, responsabilidades e colaboração.
- **Consequências** – este elemento é fundamental, porque permite comparar padrões diferentes, e constatar os custos e benefícios da sua adopção.

Uma vez apresentada a definição de padrão, o passo seguinte é enquadrar a sua importância no âmbito do projecto. Os padrões analisados neste subcapítulo pretendem solucionar problemas de arquitectura e funcionamento do projecto. Os padrões ponderados são enumerados a seguir:

- *Model-View-Controller* (MVC)
- *Model-View-Presenter* (MVP)
- *Observer*
- *View Navigation*
- *Broker*
- *Inversion of Control*

2.2.1 Model View Controller

As raízes deste padrão advêm da necessidade de desenvolver uma arquitectura que sustente a evolução de uma aplicação interactiva, sem que para isso seja necessário refazer grande parte do código. Através de uma estrutura mais flexível, deverá ser possível proceder quer a alterações na parte de visualização, quer a alterações no modelo de dados, sem causar implicações no resto da aplicação.

Outra característica apetecível que deverá ser suportada pela nova arquitectura é a modularidade, indo de encontro ao que o Krasner e o Pope referiram [KP88], ao afirmarem que "Na construção de aplicações interactivas, similarmente a outras aplicações, a modularidade dos componentes tem enormes benefícios. Isolar ao máximo as diversas unidades funcionais, possibilita ao arquitecto do sistema uma análise e alteração mais fácil de cada uma das unidades. O arquitecto para alterar um detalhe numa unidade não necessita de conhecer detalhadamente as restantes.". Através de uma arquitectura bem escalonada, em que cada componente tem as suas tarefas muito bem delineadas é possível dividir o processo de desenvolvimento e dividir esforços em cada uma das componentes. Para desenvolver a interface da aplicação não é necessário ter conhecimento sobre a lógica de negócio, apenas é preciso invocar um conjunto de métodos responsáveis por assegurar esse serviço.

Uma possível solução para este problema é o *Model-View-Controller*, que divide o código em três componentes (Figura 2.6) com responsabilidades distintas. A divisão nessas três componentes fomenta a reutilização e manutenção dos diversos módulos da aplicação. O *Model-View-Controller* é composto: pela *View*, que transmite a informação para o utilizador; o *Controller*, que recebe os Inputs do utilizadores; e o *Model*, que é responsável por implementar a lógica de negócio da aplicação e guardar a informação. De seguida os componentes são analisados individualmente em maior detalhe.

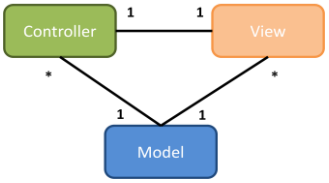

Figura 2.6 – Diagrama de classes do MVC

- **Model** (Lógica de negócio e dados) -. O *Model* gere toda a informação da aplicação e é responsável por implementar toda a lógica de negócio da *View*. Um *Model* pode estar associado a várias *Views* e *Controllers* (Figura 2.6). Assim sendo, é da sua responsabilidade informá-los quando ocorrem alterações nos dados ("change-propagation mechanism"), através do evento *StateChanged*. O "change-propagation mechanism" vem de encontro ao seguinte comentário, "Para maximizar o encapsular da informação e consequentemente a reutilização do código, *Views* e *Controllers* necessitam de ter consciência explícita do seu Model, enquanto os *Models* não devem ter conhecimento das suas Views e Controllers."[KP88]. Tendo em conta que várias *Views* ou *Controllers* podem estar dependentes de um *Model*, é necessário assegurar que a informação que mantém é actualizada. Derivado a essa necessidade surgiu o conceito de "Change-propagation mechanism", que simplesmente assegura que todas as *Views* interessadas na informação são avisadas quando há alterações. O "Change-propagation mechanism" advém de outro padrão, Observer, que também é analisado neste capítulo.
- **View** (output) - A *View* é responsável por mostrar toda a informação ao utilizador. Ela contém uma ligação para um *Model*, através da qual consegue manter actualizada a informação que detém. A ligação que estabelece com o *Model* permite-lhe consultar a informação e mantê-la actualizada. A *View* tem de se registar no *Model*, para assim poder ser informada quando ocorrem alterações nos dados ("change-propagation mechanism").
- **Controller** (input) - O *Controller* é a ponte entre a aplicação e o utilizador, sendo responsável por captar os seus inputs e transforma-los em acções. Essas acções são depois passadas para o *Model* ou para a *View*. Os inputs dos utilizadores são normalmente despoletados através de eventos, que advém normalmente de acções do rato, botões ou do teclado. Existem alguns casos em que o *Controller* depende do estado do modelo, nesses casos o *Controllor* deve registar-se no *Model* ("change-propagation mechanism") e implementar uma função para actualização. O *Controller* mantém uma ligação para uma *View*, permitindo-lhe um acesso directo quando ocorrem acções que não necessitam de passar pelo *Model*.

O diagrama seguinte (Figura 2.7) exibe a ligação entre os diversos componentes que fazem parte do padrão e identifica as relações que existem entre eles. Através da análise do diagrama é possível verificar os serviços que os componentes trocam entre si.

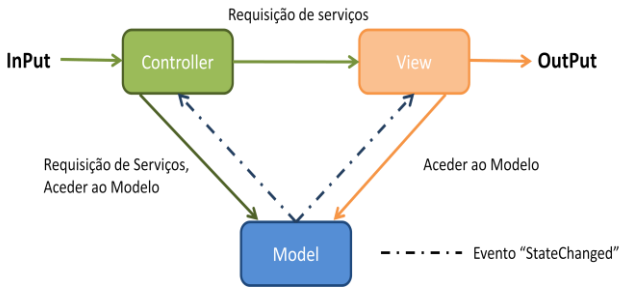

Figura 2.7 - Diagrama de interacção dos componentes do MVC

2.2.2 Model View Presenter

O *Model-View-Presenter* (MVP) é um padrão que foi elaborado com base no MVC, sendo que uma das principais diferenças entre eles é que no caso do MVC a *View* também tem conhecimento do *Model* [Mal98]. Segundo Mike Potel [Pot96], "Existem diversos benefícios em generalizar o conceito do *Model*. O primeiro é que permite uma separação limpa entre o *Model* e a apresentação", ou seja, o MVP permite uma divisão clara entre a informação e a forma como é tratada. Segundo esta filosofia é possível isolar o modelo de dados do resto da aplicação, promovendo a separação e facilitando uma possível evolução futura. Jean-Paul Boodhoo [Boo06] acrescenta outra ideia, ao afirmar que, "Sem uma clara separação das responsabilidades, a camada de interface usualmente contém lógica, que na realidade pertence a outras camadas da aplicação.", reforçando a perspectiva de que cada componente deve ter a sua tarefa bem delineada, facilitando a manutenção/teste das aplicações produzidas.

O MVP é composto por três componentes, similarmente aos MVC, mas existem algumas diferenças no nome e nas respectivas funcionalidades. A *View* desempenha o mesmo papel que no MVC, embora com o acréscimo da recepção dos Inputs do utilizador. O *Controller* no MVP não existe, sendo substituído pelo *Presenter*, que tem a obrigação de implementar toda a lógica de negócio. O *Model* fica apenas encarregue de estabelecer a ponte com a informação, sendo possível através dele efectuar algumas operações sobre os dados.

Uma *View* está ligada apenas a um *Presenter* e vice-versa. Um *Presenter* por sua vez apenas têm um *Model*, mas um *Model* pode estar ligado a vários *Presenters*. As relações entre as classes podem ser verificadas na ilustração seguinte (Figura 2.8).

Figura 2.8 – Modelo de classes do MVP

- **View** - é responsável pela visualização da informação. A *View* contém uma referência para o *Presenter*, através da qual pode encaminhar todos os *Inputs* que recebe.
- **Presenter** - é responsável por implementar a lógica de negócio. O *Presenter* é a única classe com acesso ao *Model*, através dessa ligação pode aceder à informação, processa-la e actualiza-la. A comunicação entre o *Presenter* e a *View* é feita através de uma interface, desta forma o *Presenter* torna-se independente da *View* (promove a reutilização do código). O facto de o *Presenter* aceder à *View* através de uma interface também agiliza bastante a fase de testes, já que se pode simular o comportamento de uma *View* e verificar a reacção do *Presenter*. O Presenter é responsável por actualizar a View, fornecendo-lhe toda a informação necessária.
- **Model** - apenas serve de ponte entre os dados e o *Presenter*, facilitando uma interface para o seu acesso. Através desta abordagem é possível alterar o modelo de dados ou mesmo a plataforma que os sustentam, sem ter de alterar a lógica de negócio.

Na figura seguinte (Figura 2.9) pode-se verificar que o *Presenter* não acede directamente à *View*, o *Presenter* utiliza uma interface para operar sobre ela.

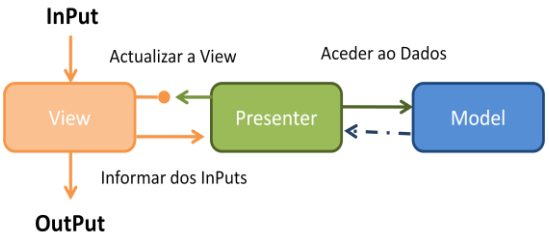

Figura 2.9 – Diagrama de interacção dos componentes do MVP

A utilização da interface permite facilitar a validação da lógica de negócio, implementada no *Presenter*, porque facilmente é possível substituir a View por um *MockObject* e correr alguns testes [Mcr08].

Existe outra variante do MVP, que apenas difere na forma como a *View* é actualizada, *Supervising Controller*. Essa variante será apresentada mais em detalhe de seguida.

2.2.3 Supervising Controller

Ao contrário do MVP tradicional, o *Supervising Controller* destaca-se na forma como a *View* recebe a informação que necessita para manter a interface actualizada. Enquanto no MVP o *Presenter* é responsável por assegurar todo esse serviço, no caso do *Supervising Controller* essa função é repartida entre o *Model* e o *Presenter*. A *View* mantém uma ligação para o *Model*, através da qual é assegurada a actualização da maior parte da informação (todas as operações que são possíveis mapear através de *data-binding* declarativo). Nos casos em que não é possível recriar a actualização, através de *data-binding* declarativo, então o *Presenter* fica encarregue de actualizar os dados da *View*.

Na escolha entre o padrão original e esta variante é preciso ponderar entre a simplicidade do código e uma maior facilidade de teste. Caso se opte pelo padrão original, MVP, então privilegia-se a capacidade de testar os módulos de forma isolada, embora o *Supervising Controller* também o permita. Por outro lado, caso a decisão se incline para o *Supervising Controller* então está-se a privilegiar a simplicidade de código. Esta surge ao nível das pequenas alterações na *View*, porque não implicam alterar o *Presenter* para suportar essas actualizações.

De seguida é apresentada uma figura (Figura 2.10) que ilustra a forma como os diversos componentes, estabelecidos no padrão, se relacionam entre si.

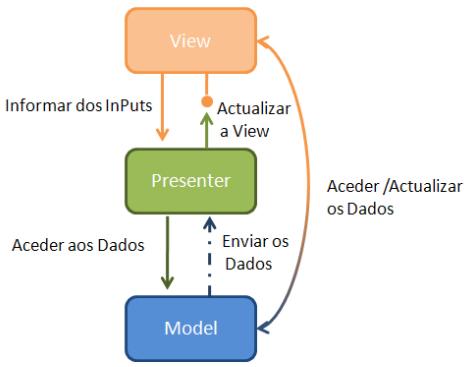

Figura 2.10 - Diagrama de interacção dos componentes do Supervising Controller

2.2.4 Observer

Em primeiro lugar vai-se expor alguns dos motivos que levaram a construção do padrão *Observer*.

- "Changing the internal state of a component may introduce inconsistencies between cooperating components. To restore consistency, we need a mechanism for exchanging data or state information between such components." [BMRSS96]
- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [GHJV97]

Como se pode verificar através dos comentários recolhidos, o padrão *Observer* tem como intuito fornecer uma solução para o problema de sincronização de informação/estado entre objectos distintos. A descrição do padrão será feita no contexto da sincronização de dados, mas também é válida no contexto da sincronização de estado entre objectos. Esta problemática é crucial porque se um objecto depende da informação residente noutro, é fundamental que seja informado quando os dados se alteram. Caso não seja avisado quando a informação se altera, a cópia que detém pode tornar-se obsoleta. Se este problema se coloca numa relação unitária entre objectos, imagine-se quando existe uma relação de um para muitos, em que vários objectos podem aceder à origem da informação e altera-la. Nesses casos, esta questão torna-se ainda mais crítica, porque é difícil assegurar que cada um dos componentes dependentes da informação esteja actualizado.

Um efeito colateral conhecido do particionamento de uma aplicação em classes é a necessidade de manter a consistência de estado entre elas. Uma possível solução para esse problema seria assegurar que as classes estavam firmemente acopladas, obrigando-as a propagar explicitamente as alterações que ocorrem para os seus parceiros. Essa solução não é a mais apetecida, porque limita de antemão a reutilização dessas classes [GHJV97]. Uma solução mais apetecível deveria respeitar as seguintes características: os componentes devem ser fracamente ligados; a fonte da informação (*Model*) não deve depender dos detalhes dos seus colaboradores e os componentes que dependem da informação não devem ser conhecidos à priori [BMRSS96].

Uma das possíveis abordagens que pretende contornar as adversidades referidas e respeitar as características mencionadas é a adopção do padrão *Observer*, também conhecido por *Publish-Subscriber*. O padrão *Observer* pretende criar um mecanismo que assegure a propagação das alterações entre a entidade que guarda a informação/disponibiliza um serviço (*Provider*) e os componentes que dependem dela (*Observers*). Esse mecanismo deve possibilitar aos *Observers* registarem-se e cancelarem o seu registo para um determinado *Provider*. Quando o estado de um *Provider* é alterado o mecanismo deve propagar as alterações por todos os *Observers* registados, assegurando assim a consistência da informação que detêm. Quando ocorrem alterações, o mecanismo lança um evento que é comum a todas as entidades registadas para aquele Provider.

A Figura 2.11 é uma boa representação da forma como deve funcionar o mecanismo. No caso descrito nessa figura existe um conjunto de *Views* que depende do modelo de dados, sendo que cada uma das *Views* pode aceder-lhe e efectuar alterações. Quando ocorrem alterações no modelo, todas as *Views* que dependem dele recebem um evento a anunciar que foi alterado.

Depois de receberem a notificação, cada uma das *Views* é responsável por aceder ao modelo e actualizar a informação que lhes interessa.

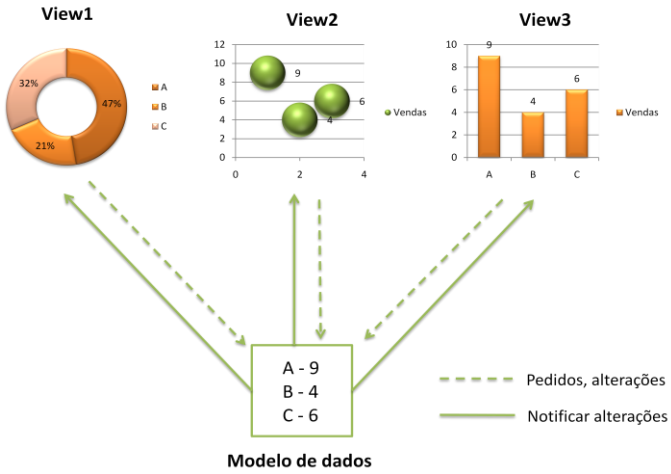

Figura 2.11 – Diagrama de comunicação entre *Views* e o *Model*

2.2.5 View Navigation

Partindo do pressuposto que se está a trabalhar num ambiente composto por várias *Views*, que podem comunicar entre si, é necessário estabelecer uma norma de comunicação. Se para além do factor anterior se pretender promover a modularidade e facilitar futuras alterações na aplicação, então é preciso adoptar estratégias que permitam aumentar a flexibilidade da aplicação.

Um dos contextos onde é preponderante promover a flexibilidade, pelo menos ao nível das interfaces, é nas aplicações interactivas. Essa necessidade surge pelo facto de regularmente ser necessário introduzir novas funcionalidades, que podem ser mapeadas em novas *Views*, na alteração das existentes, ou na introdução de novas rotinas entre as existentes. Para promover a flexibilidade na comunicação entre *Views* é necessário adoptar um mecanismo que permita sustentar a comunicação sem saber de antemão os seus intervenientes. Caso se consiga recriar esse mecanismo, a alteração dos intervenientes de uma mensagem pode ser alterada facilmente. Na prática, isso significa que é fácil alterar a *View* que é chamada para desencadear determinada acção.

O padrão pressupõe as seguintes condições:

- A cada *View* está subjacente um *Presenter*, ou seja, a utilização do padrão MVP.
- As mensagens que são trocadas entre *Views* são traduzidas em eventos.
- Os destinatários dos eventos têm de se registar para recebê-los (lógica *Publisher Subscriber*).
- Existe uma entidade, *Event Broker*, que é responsável por coordenar todos os eventos.

Assim sendo, se todas as condições pré-estabelecidas forem respeitadas, a comunicação através do padrão torna-se trivial. A *View1* se pretender enviar uma mensagem para a *View2*, deve informar a sua intenção ao seu *Presenter*, que por sua vez lança o evento respectivo para o *Event Broker*. A *View2* deve informar antecipadamente o *Event Broker* da sua intenção de

receber eventos daquele tipo. Uma vez subscritos os eventos por parte da *View2* resta-lhe apenas esperar que eles sejam lançados. A figura seguinte (Figura 2.12) é uma boa ilustração do cenário descrito.

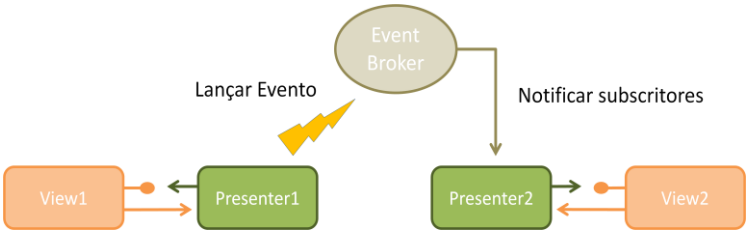

Figura 2.12 – Modelo de comunicação entre *Views*

2.2.6 Inversion of Control

Segundo Miller [Mil05], "a injecção de dependências simplesmente significa que uma determinada classe ou sistema deixa de ser responsável pelo instanciar das suas próprias dependências. Neste caso, "Inversion of Control" refere-se ao mover da responsabilidade de localizar e acrescentar as dependências dos objectos de outras classes". O que Miller quis realçar através da sua definição do padrão *Inversion of Control* (IoC), foi a capacidade do padrão de desacoplar dos objectos a tarefa de encontrar e inicializar as suas dependências. Para que isso seja possível é necessário fornecer uma plataforma capaz de registar os objectos existentes e as suas dependências. Quando um objecto novo é criado a plataforma deve fornecer-lhe as suas dependências e distribuir a sua referência pelas entidades que dependem dele. Hellesøy [Hel06] ajuda a sustentar a definição enunciada até ao momento, ao afirmar que, "Não basta construir um componente segundo o padrão IoC para que ele vá buscar os componentes que necessita para concluir o seu trabalho. Ao invés, ele deve declarar as suas dependências e o repositório fornece-as.". A figura seguinte (Figura 2.13) mostra um conjunto de classes e as suas dependências.

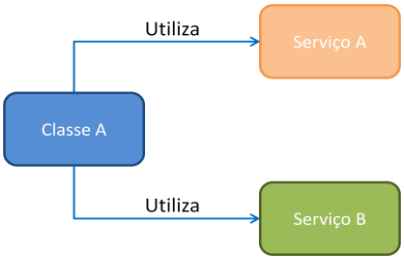

Figura 2.13 – Diagrama de dependências entre elementos

Na figura anterior (Figura 2.13) pode-se verificar a relação entre uma classe e dois serviços. A classe "A" necessita dos serviços para cumprir as funções que lhe foram instituídas. Enquanto numa abordagem usual a classe A teria de saber onde se encontravam os serviços e procurar obter uma referência para eles, segundo o padrão IoC ela apenas teria de declarar as suas dependências e ser-lhe-iam atribuídas as referências para os serviços. A declaração das dependências de uma classe implica informar a entidade responsável por implementar o IoC das

suas relações, só depois essa entidade será capaz de resolver as suas necessidades. A entidade responsável pelo IoC só conseguirá devolver referências para as classes que forem previamente registadas.

Os factores que favorecem a utilização deste padrão são:

- Permite criar classes que sustentam mais facilmente a criação de testes unitários isolados.
- Promove um maior desacoplamento entre classes e subsistemas.
- Aumenta a flexibilidade do código, facilitando a manutenção/evolução das funcionalidades existentes.
- Possibilita separar a responsabilidade de localizar e manter o ciclo de vida das dependências dos objectos, dos mesmos [Mic09a].

2.2.7 Conclusões

Tendo em conta os padrões analisados é importante realçar a sua importância no âmbito deste projecto, uma vez que permitem endereçar algumas problemáticas importantes na definição do esquema da aplicação desenvolvida.

Todos os padrões analisados foram adoptados, à excepção do MVC e do *Supervising Controller*. O MVC e o *Supervising Controller* não foram adoptados porque pretendem solucionar as mesmas questões do MVP, que no intuito deste projecto é considerado o mais adequado. O MVP foi escolhido uma vez que privilegia a modularidade, a manutenção, facilita a fase de testes e incute uma divisão clara das competências de cada entidade de uma interface (o *Model*, a *View* e o *Presenter*).

A utilização dos padrões é feita de forma implícita ou explícita, já que alguns estão enraizados nas tecnologias ponderadas. No subcapítulo seguinte será possível constatar esse facto e verificar quais são os padrões que estão agregados às tecnologias.

2.3 Revisão Tecnológica

Neste subcapítulo são apresentadas todas as tecnologias ponderadas ao longo da fase de análise do problema. É necessário desde já frisar que as tecnologias analisadas foram um requisito da empresa, não tendo ficado ao cargo do autor a sua selecção. Ao estagiário apenas coube a tarefa de as analisar e verificar a viabilidade de as utilizar no decorrer do seu estágio.

As tecnologias estão divididas em duas categorias, de um lado as *Frameworks*, do outro lado as tecnologias que facilitam e dão suporte à construção de interfaces complexas.

2.3.1 Framework

Em primeiro lugar serão apresentadas algumas definições de *Framework* foram recolhidas através de uma pesquisa sobre o tema.

- "A set of cooperating classes that makes up a reusable design for a specific class of software." [GHJV97]
- "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes." [JF88]
- "A framework is an incomplete design that factors commonalities and variabilities of a specific domain of application." [LTMS07]
- "An application framework is a collection of classes implementing the shared architecture of a family of applications." [HHKPVV01]
- "A framework, then, is a semicomplete application that contains certain fixed aspects common to all applications in the problem domain, along with certain variable aspects unique to each application generated from it." [Sav99]
- "An object-oriented framework is defined as a set of related classes and object interactions, that models a common behavior and functionality in certain domain."[SCKR99]

Segundo as definições apresentadas, uma *Framework* resume-se a uma abstracção de classes que pretende evidenciar o comportamento de um tipo de aplicações, para um dado contexto. A abstracção de classes consiste na definição de um conjunto de classes que modelam um género de problemas, assim como, a definição da interacção entre os diversos elementos do modelo. Uma *Framework* não pretende apenas ser uma abstracção teórica, pelo contrário, uma *Framework* é um esqueleto de uma aplicação, capta as decisões de *design* que são comuns para o domínio onde está inserida. No fundo uma *Framework* fornece um guia da arquitectura da aplicação, promovendo a reutilização do *design* face ao código [GHJV97].

Segundo Shin et al. [SCKR99], "Uma *Framework* é disponibilizada sobre a forma de uma aplicação "Semi-completa", e serve de núcleo para a nova aplicação". Esta perspectiva é muito interessante e vai de encontro à verdadeira essência das *Frameworks*, já que elas são o ponto de partida do desenvolvimento de uma aplicação e pretendem que os seus utilizadores apenas se foquem nos detalhes do problema [Raz07].

Uma *Framework* normalmente é constituída por uma mistura de classes abstractas e concretas, sendo que as classes abstractas normalmente residem na *Framework*, enquanto as

classes concretas estão alojadas na aplicação. Tal como já foi referido, uma *Framework* é uma aplicação semi-completa que é composta por alguns detalhes fixos, inerentes ao tipo de problema. Por outro lado, a *Framework* também deixa alguns aspectos em aberto, fruto da especificidade dos problemas. Os detalhes que são deixados em aberto normalmente são chamados de "Hot Spots" e pretendem aumentar a flexibilidade da *Framework*, permitindo abranger um maior leque de aplicações. O que diferencia duas aplicações desenvolvidas pela mesma *Framework*, que partilham o mesmo domínio do problema, é a forma como implementa esses "Hot Spots" [Sav99].

Uma boa *Framework* deve conseguir prever as necessidades das aplicações que serão implementadas por ela, fornecendo um modelo suficientemente expansível. Esse modelo deve permitir adicionar novas funcionalidades e serviços que irão de encontro às necessidades das aplicações. Uma boa *Framework* fomenta um esquema modular, facilitando a tarefa de manutenção das aplicações [Raz07].

Embora as *Frameworks* prometam uma maior produtividade de desenvolvimento, reduzam o tempo para lançar novos produtos e aumentem a qualidade das soluções, esses benefícios só podem ser alcançados a longo prazo e implicam um maior investimento inicial. Numa fase inicial é necessário despender muito tempo/esforço na adaptação à *Framework*, investindo tempo na análise da sua arquitectura, dos seus princípios de arquitectura e na aprendizagem de como personaliza-la [AD06]. Não obstante do facto de as *Frameworks* facilitarem um grande potencial de reutilização, essa característica apenas pode ser aproveitada caso a equipa de desenvolvimento consiga adaptar-se convenientemente à *Framework*. A complexidade do esquema e da implementação da *Framework* pode tornar-se um forte entrave ao sucesso da tarefa, conduzindo a uma curva de aprendizagem acentuada. O esforço da aprendizagem pode ser substancialmente reduzido caso a documentação existente seja boa e acompanhada de material para treino [FHLS97].

As *Frameworks* normalmente são um caso difícil de documentar, uma vez que representam um esquema reutilizável com um elevado nível de abstracção [Sav99]. Para tentar contornar o problema associado à falta de documentação, ou à qualidade da mesma, Johnson [Joh92] propôs que os seguintes documentos devessem acompanhar a *Framework*:

- **O propósito da *Framework*** - uma descrição do domínio da *Farmework*, os requisitos que pretende satisfazer e as suas limitações.
- **A utilização da *Framework* -** uma descrição da forma como a *Framework* deve ser utilizada.
- **O esquema da *Framework*** - uma descrição da estrutura e do comportamento da *Framework*.

Caso as suas directivas sejam seguidas então deverá ser mais fácil analisar uma *Framework* e verificar se é a mais adequada para o problema em questão. Por outro lado, também deverá ser mais fácil elaborar uma estimativa do tempo necessário de adaptação à *Framework* e consequentemente fazer uma análise mais capaz do custo benefício da sua adopção para um projecto.

Existe ainda quem defenda que o sucesso das *Frameworks* no futuro não passará por uma documentação mais eficaz, mas sim pelo fornecimento de ambientes específicos de

programação. Esses ambientes simultaneamente guiaram e controlaram os programadores das aplicações através do processo de criação, assegurando que as convenções estipuladas pela *Framework* são seguidas [HHKPVV01].

Depois desta pequena introdução ao conceito de *Framework*, passar-se-á à apresentação das *Frameworks* ponderadas no intuito deste projecto. As *Frameworks* analisadas pertencem ambas à empresa Microsoft e são conhecidas por *Composite UI Application Block* (CAB) e *Composite Applications* (PRISM). O objectivo de ambas as *Frameworks* é fornecer uma plataforma que permita desenvolver aplicações interactivas de uma forma mais ágil, respeitando um requisito muito importante e intrínseco a este tipo de aplicações, que é a volatilidade dos requisitos.

As aplicações interactivas implicam uma evolução contínua da forma como a aplicação comunica com o utilizador, tentando melhorar o canal de comunicação com os utilizadores. A interacção entre o utilizador e a aplicação é um domínio difícil de endereçar, pois é preciso eliminar possíveis ambiguidades e definir um mecanismo que permita uma comunicação simples. Segundo Deighton [Dei96], a interacção tem duas componentes fundamentais: a capacidade de transmitir uma mensagem a uma pessoa e a aptidão de recolher e lembrar a resposta dessa pessoa. Os profissionais que se dedicam à construção de interfaces normalmente tentam seguir uma máxima muito conhecida no meio, que é, KISS – "Keep It Simple, Stupid". Quanto mais simples for a interface disponibilizada aos utilizadores menor é a probabilidade de estes errarem ou confundirem a mensagem e mais rápida será a sua adaptação ao sistema.

No desenvolvimento de aplicações interactivas é necessário assegurar que a estrutura escolhida é suficientemente flexível para responder a novos requisitos do utilizador, sem que para isso seja necessário um grande esforço por parte da equipa de desenvolvimento. Espera-se que a *Framework* escolhida permita desenvolver aplicações que facilitem a reutilização/troca de componentes. Para que essa funcionalidade seja possível é necessário que os componentes desenvolvidos sejam desacoplados e facilmente integráveis. Nos pontos seguintes é apresentada uma comparação entre o conceito de *Framework* e Biblioteca, seguido da apresentação das *Frameworks* analisadas.

2.3.1.1 Framework Vs Library

Para começar é importante definir o conceito de Biblioteca (*Library*). Uma Biblioteca é essencialmente um conjunto de funcionalidades que estão organizadas segundo classes. Uma Biblioteca existe para permitir a reutilização de funcionalidades que são utilizadas recorrentemente [Atk97].

Partindo da definição de Biblioteca fornecida, não existem grandes diferenças face ao conceito de *Framework*, embora uma *Framework* seja muito mais do que um conjunto de funcionalidades. Segundo Embassy [Emb06], "*Framework* é um esquema abstracto que representa a forma como uma aplicação funciona e tem "locais" onde é possível "injectar" um módulo ou uma componente", ou seja, uma *Framework* não guarda apenas algumas funcionalidades que são reutilizáveis, capta também o domínio do problema onde as funcionalidades estão inseridas e condiciona o comportamento da aplicação que as utilizar. Segundo Fowler [Fow96], "A framework embodies some abstract design, with more behavior

built in.”. Como se pode verificar, mais uma vez foi dado maior ênfase ao comportamento da aplicação onde vai ser reutilizada a funcionalidade, do que propriamente à funcionalidade.

Com uma Biblioteca o código do utilizador controla a estrutura do programa e apenas recorre a algumas das suas funcionalidades quando é necessário. Neste contexto a Biblioteca é apenas uma ferramenta auxiliar para contornar algumas dificuldades pontuais. No caso de uma *Framework* esta é que controla a aplicação, o utilizador apenas complementa alguns espaços que lhe permitem adaptar a solução [Bru05].

Por último, mas não menos importante, fica um comentário que capta claramente a essência das duas realidades. Segundo Embassy [Emb06], “Uma Biblioteca é acerca de reutilizar funcionalidades, enquanto uma *Framework* pretende reutilizar comportamentos”.

2.3.1.2 CAB

CAB é a sigla porque é conhecida a *Framework Composite Application Block* desenvolvida pela Microsoft. A palavra *Composite* provém do facto de a *Framework* facilitar a permutação de diversos componentes, tendo em conta os requisitos que se pretende satisfazer. No fundo é possível desenvolver um conjunto de componentes e depois conjuga-los de acordo com as necessidades da aplicação que se pretende desenvolver. O resto do nome sugere a forma como a *Framework* é facultada. O código fonte da *Framework* é disponibilizado sobre a forma de um bloco aos utilizadores, que depois apenas necessitam compilá-lo e incluí-lo nos seus projectos. [Raz07a]

Esta *Framework* pretende incutir boas práticas de programação no desenvolvimento de aplicações interactivas complexas, tendo por base padrões que já foram devidamente testados e reconhecidos. Um bom exemplo de um padrão que é seguido pela *Framework* é o *Composite.* O padrão *Composite* consiste em desenvolver interfaces mais complexas através da conjugação de interfaces mais simples, proporcionando a possibilidade de testar, desenvolver e usar cada uma das componentes de forma independente.

Esta *Framework* pretende possibilitar: a interacção com múltiplos componentes, facilitando a sua inclusão e configuração; oferecer uma experiência mais rica para os utilizadores; utilizar mecanismos mais complexos de interacção com a informação que permitem melhorar a resposta das interfaces.

2.3.1.2.1 Principais Características

Esquema modular das aplicações

A *Framework* CAB pretende que as aplicações desenvolvidas segundo os seus princípios permitam construir aplicações modulares, em que os seus componentes são independentes e reutilizáveis. Segundo esta filosofia as aplicações são construídas através da alocação dos módulos que são necessários, utilizando um princípio semelhante ao “Plug&Play”. Embora a analogia possa parecer um pouco descabida, no fundo é esse o conceito que a *Framework* pretende difundir. A integração de um novo módulo numa aplicação deve ser trivial, tendo em conta que a *Framework* disponibiliza mecanismos para a comunicação entre módulos distintos e possibilita obter referências para os serviços pretendidos.

A *Framework* encarrega-se de transmitir as mensagens entre módulos (*Event Broker*) segundo uma filosofia, *Publish-Subscribe*, em que os módulos indicam os serviços que disponibilizam e os interessados devem subscrevê-los. A partir daí, a *Framework* fica responsável por difundir as mensagens, eliminando a necessidade de os módulos terem consciência uns dos outros [Pla07].

Dependecy Injection

A *Framework* CAB segue a filosofia do padrão *Inversion-of-Control*, descrito na secção de padrões. Segundo este padrão os módulos não necessitam de se preocupar em obter as suas dependências, a *Framework* fica responsável por assegurar esse serviço. Esta funcionalidade permite fortalecer o desacoplar dos módulos e promover o esquema modular, já que os módulos não necessitam conhecer à priori onde se encontram as suas dependências. O *Object Builder* é a *Dependency Injection Framework* por trás de CAB [Ber07].

Facilita o desenvolvimento distribuído de um projecto

O desenvolvimento de projectos de média ou grande dimensão envolve a alocação de equipas de grande dimensão, cujos elementos têm valências em áreas distintas. Considerando a dimensão dos projectos e o facto de envolverem diversas áreas de conhecimento, é necessário na maior parte dos casos dividir os projectos por partes e atribuir a equipas mais restritas. A interacção entre as diversas equipas continua a ser fundamental, porque no final do projecto é esperada uma solução coesa e que consiga integrar efectivamente as diversas funcionalidades. À medida que o projecto cresce a sua complexidade aumenta, assim como a dificuldade de o coordenar, sendo fundamental escolher logo à partida uma arquitectura que suporte efectuar mudanças facilmente. Caso essa situação não seja ponderada e a arquitectura da aplicação não seja flexível, as tarefas de actualização e manutenção podem estar comprometidas, já que os esforços necessários para as efectuar podem tornar-se inviáveis. A *Framework* em causa pretende endereçar todas estas problemáticas e, tal como já foi referido, facilitar o desenvolvimento das aplicações de forma modular, promovendo a construção de módulos e a sua reutilização [Mic05].

O facto de a *Framework* disponibilizar um mecanismo que permite guardar os serviços existentes e injectar as dependências que os módulos necessitam, facilita o desenvolvimento dos módulos de forma independente. As equipas no decorrer do projecto apenas necessitam de acordar as dependências entre módulos e os serviços que disponibilizam. A *Framework* fica responsável de resolver as devidas referências.

Diminui a complexidade da fase de testes

O facto de as aplicações serem construídas por módulos, que mais tarde são integrados, permite diminuir a complexidade inerente à fase de testes. É possível testar cada um dos módulos independentemente, emulando os dados provenientes dos módulos de que dependem. Ao elaborar os testes desta forma a detecção de situações anómalas é mais fácil, filtrando a maior parte dos erros antes da fase de integração [Mic05].

Suporte de raiz para a tecnologia Windows Forms

A *Framework* CAB também foi desenvolvida com o intuito de facilitar o desenvolvimento de aplicações interactivas, que utilizassem componentes da tecnologia *Windows Forms*, que facilita a construção de interfaces. A tecnologia *Windows Forms* disponibiliza alguns componentes visuais que permitem criar uma interface mais facilmente, sem que para isso seja necessário desenvolver todos os controlos essências para interagir com os utilizadores. Essa tecnologia será abordada em maior detalhe numa fase posterior.

2.3.1.2.2 Apresentação da arquitectura disponibilizada em CAB

O desenvolvimento de aplicações com base em CAB subentende a adopção dos paradigmas de estruturação definidos pela *Framework*. Na figura seguinte (Figura 2.14) é possível constatar os principais componentes de uma aplicação.

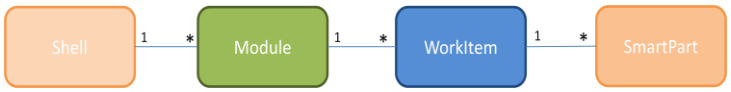

Figura 2.14 – Componentes base da *Framework* e respectiva ligação

A *Shell* é o componente base de uma aplicação, disponibilizando os serviços mínimos necessários para correr uma aplicação. As aplicações em CAB, tal como já foi referido, estão divididas em módulos, sendo que cada módulo é responsável por implementar um conjunto de funcionalidades. A *Shell* disponibiliza um conjunto de mecanismos que permitem carregar os módulos necessários para uma determinada aplicação, assim como, as áreas visuais (*Workspaces*) onde os módulos podem expor as suas componentes gráficas. No fundo a *Shell* é um simples repositório que inicializa a aplicação, alberga os módulos e cria o *WorkItem* inicial da aplicação.

Os módulos por sua vez são compostos por *WorkItems*, sendo que, cada *WorkItem* é responsável por implementar um caso de uso da aplicação. Os módulos agrupam um conjunto de funcionalidades que fazem sentido enquanto um todo. Esta organização permite a construção de componentes mais modulares e reutilizáveis, facilitando a construção e manutenção das aplicações.

Os *WorkItems* albergam um conjunto de componentes que colaboram com o ímpeto de satisfazer um caso de uso. A sua responsabilidade é coordenar o desenrolar de um caso de uso, fornecendo as ferramentas necessárias para a sua concretização. Resumidamente, os *WorkItems* são o elo de ligação entre os componentes desacoplados da aplicação, fornecendo um porto onde é possível encontrar os recursos necessários para levar a cabo as suas tarefas.

Por fim, surgem as *SmartParts*, mais conhecidas por *Views*, que representam os componentes visuais de uma aplicação em CAB. As *SmartParts* são apresentadas através das *WorkSpaces* e permitem estabelecer uma ponte entre o utilizador e a aplicação. As *SmartParts* são um componente fundamental nesta estrutura, já que o sucesso de uma aplicação está intrinsecamente ligado à sua usabilidade e à sua capacidade de apresentar a informação.

De seguida serão apresentados em maior detalhe os componentes referidos anteriormente, assim como, outros mecanismos que sustentam o seu bom funcionamento.

Shell

A classe inicial de um projecto em CAB é a *Shell*. A *Shell* tem de derivar de um dos tipos de aplicações suportados pela *Framework*. A figura seguinte (Figura 2.15) apresenta os tipos disponíveis, assim como a hierarquia que existe entre eles:

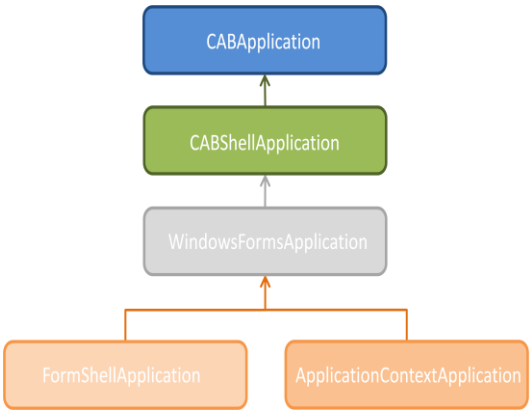

Figura 2.15 – Diagrama hierárquico dos tipos de aplicações suportados por CAB

De seguida são apresentados os detalhes sobre cada uma das classes apresentadas no diagrama, sendo importante referir que normalmente só se utilizam as duas últimas classes da hierarquia.

- ***CabApplication*** – Inicializa os requisitos mínimos necessários para tirar partido da *Framework* CAB.
- ***CabShellApplication* –** Usada em aplicações que têm de mostrar a *Shell* no período de inicialização.
- ***WindowsFormsApplication*** – Esta classe permite a utilização de *Shells* baseadas em *Windows Forms* e possibilita a inclusão de novos componentes para melhorar a aplicação.
- ***FormsShellApplication*** – Esta classe é uma é uma das mais comuns na criação de aplicações em CAB. A classe deriva da *WindowsFormsApplications* e na sua inicialização lança uma janela em *Forms*.
- ***ApplicationContextApplication*** – Esta classe é similar à anterior, a única diferença é que na sua inicialização não lança uma janela em *Forms*. Normalmente esta classe é utilizada para interfaces complexas ou quando uma aplicação é uma componente de outros sistemas [Scu08] [Adl06].

Na figura seguinte (Figura 2.16) pode-se constatar os elementos que compõem o projecto da *Shell*. Todos os elementos serão convenientemente explicados e contextualizados no decorrer da explicação da *Shell*.

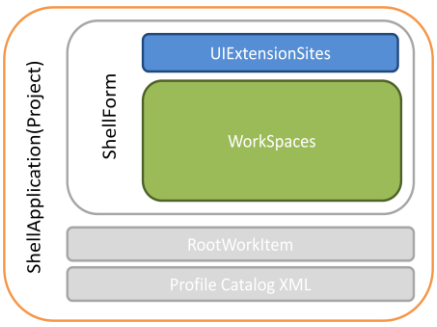

Figura 2.16 – Componentes da Shell

A *Shell* é responsável por carregar os módulos que compõem a aplicação. Para desempenhar essa tarefa recorre ao *Module Loader.* O *Module Loader* é a entidade responsável por carregar os módulos, inicializá-los e depois adicioná-los à Shell. O *Module Loader* descobre os módulos que necessita de carregar através *do Module Enumerator*, que por sua vez acede a um ficheiro de configurações (“ProfileCatalog.Xml”) que indica os módulos que pertencem à aplicação.

Para além de carregar os módulos, a *Shell* também é responsável pela criação do *RootWorkItem*. O *RootWorkItem* disponibiliza todos os mecanismos necessários para dar suporte às componentes da aplicação.

Normalmente a *Shell* fica responsável por lançar a janela inicial da aplicação, onde para além dos controlos usuais de *.Net* (*UIElements*), podem ser adicionados *Containers* que permitem mostrar *SmartParts*. Os *Containers* advêm de CAB, são conhecidos por *Workspaces* e permitem mostrar, esconder, activar e monitorizar as *SmartParts* que contêm, independentemente da forma como estão implementadas. Existem diversos tipos de *Workspaces*, que se distinguem na forma como mostram as suas *SmartParts*. Os *Workspaces* padrão disponíveis são [Pla07]:

- ***DeckWorkspace*** - Sobrepõe todas as *SmartParts* que contém na sua colecção e apenas mostram a que esta no topo. A *SmartPart* que é mostrada ocupa toda a área disponível.
- ***MdiWorkspace*** - Permite mostrar diversas *SmartParts* ao mesmo tempo, sendo que, cada uma das *SmartParts* é mostrada em janelas separadas.
- ***TabWorkspace*** - Segue a filosofia de *Tabs*, cada uma das *SmartParts* é inserida num *Tab*, sendo depois possível escolher o *Tab* que se pretende ver.
- ***WindowWorkspace*** - O conceito deste *Workspace* é similar ao do *MidWorkspace*, possibilitando mostrar várias *SmartParts* ao mesmo tempo. Cada uma das *SmartParts* é mostrada numa janela flutuante individual.
- ***ZoneWorkspace*** – Permite definir diversas zonas dentro de uma área. As zonas definidas ficam disponíveis para mostrar *Smartparts*, uma por zona.

A *Shell* fornece ainda outra funcionalidade que pretende facilitar o acesso dos módulos carregados á interface principal, conhecida por *UIExtensionSites*. O acesso dos módulos à interface é feita segundo algumas regras, que são preestabelecidas à priori pelo programador. O programador deve definir as zonas que podem ser alteradas e as respectivas permissões, caso

seja necessário. Usualmente esta funcionalidade é utilizada na construção de menus, barras de ferramentas e barras de estado que são partilhadas por toda a aplicação.

Cada um dos *UIElements* que a *Shell* pretende que sejam acessíveis/modificáveis pelos módulos existentes, devem ser registados no *RootWorkItem* como *UIExtensionSites*. Para conseguir distinguir os diversos elementos que são registados é necessário registá-los com um nome único. Os módulos que pretendam aceder aos *UIExtensionsSites* registados, apenas necessitam de aceder ao seu *WorkItem* e procurar pelo nome pretendido.

Module

Para começar é importante referir que os módulos implementados são obrigados a derivar da classe *ModuleInit*, presente na *Framework* de CAB.

Um módulo representa um conjunto de funcionalidades que estão distribuídas por *WorkItems*. Todas as operações efectuadas no *Module* estão dependentes do *RootWorkItem*, que ele recebe por *Dependency Injection*. A partir do *RootWorkItem* o *Module* pode criar os seus próprios *WorkItems*, que serão responsáveis por implementar os casos de uso. A figura seguinte (Figura 2.17) indica o conjunto de tarefas da responsabilidade do Module.

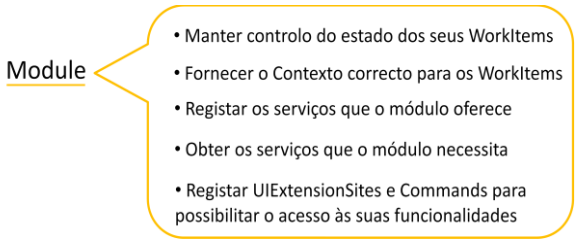

Figura 2.17 – Principais funções do Module

De entre as responsabilidades enumeradas na figura anterior (Figura 2.17), é importante salientar o registo dos serviços que o módulo oferece, assim como, a adição à interface principal das opções que permitem aceder às funcionalidades do módulo (registar *UIExtensionSites* e *Commands*). Através destas duas opções o módulo abre as portas para a comunicação com o utilizador e com os outros módulos.

Tendo em conta que os diversos módulos que compõem a aplicação são desacoplados e não têm conhecimento uns dos outros, é necessário encontrar um mecanismo que permita a comunicação entre eles. A forma encontrada foi através de Eventos, que são geridos através de um *EventBroker*. O *EventBroker* segue uma filosofia *Publish-Subscribe*, em que os módulos registam os eventos que pretendem lançar e subscrevem os eventos que pretendem receber. O elemento que faz a ponte entre o módulo e o *EventBroker* é o *WorkItem*, permitindo registar/subscrever eventos. O *EventBroker* pertence aos *Core Services* de CAB [New09].

WorkItem

Todos os *WorItems* criados são obrigados a derivar da classe *WorkItem* implementada na *Framework* de CAB. Para que seja possível a um *WorkItem* levar a cabo as suas funções, ele necessita de um conjunto de propriedades que o sustentem. A figura seguinte (Figura 2.18) é uma boa ilustração das propriedades que compõem o *WorkItem*.

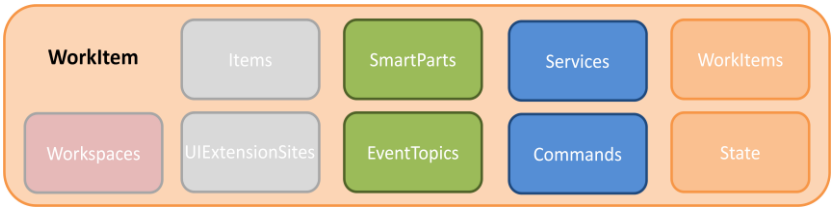

Figura 2.18 – Propriedades de um WorkItem

De seguida serão apresentadas cada uma das propriedades presentes na figura anterior (Figura 2.18), acompanhadas de uma breve explicação.

Services

Em CAB existe um conceito chamado *Service*, que permite implementar um conjunto de funcionalidades autónomas, que depois podem ser partilhadas e utilizadas por qualquer classe (promove o desacoplamento das classes). Para isso, o *WorkItem* dispõe de um repositório onde podem ser registados todos os serviços disponíveis na aplicação. O repositório chama-se *Services* e é através dele que se podem registar novos serviços ou aceder aos já existentes.

Os serviços implementados devem conter dois ficheiros, um com a implementação da classe que disponibiliza os serviços e outra que define a sua interface. Ao registar um serviço no *WorkItem* deve ser passado o nome da classe e a respectiva interface.

UIExtensionSites

Cada um dos *WorkItems* tem um repositório que guarda todas as referências para os *UIElements* registados na *Shell*. Através deste repositório é possível aceder e manipular a colecção de *UIElements* existentes, através dos respectivos adaptares que estão registados. O *Adapter* é a entidade que define os métodos existentes para manipular um tipo de *UIElements*. Depois de obtido o *Adapter* correspondente ao tipo de *UIElement* que se pretende aceder, é preciso saber o seu identificador. O identificador de um elemento normalmente é conhecido por *extension site* e é representado por uma *string*.

EventTopics

O *WorkItem* dispõe de um repositório para os eventos. Esse repositório não é mais do que uma colecção de todos os eventos registados até ao momento. Através desta lista é possível aceder aos eventos existentes e subscrevê-los ou executá-los. Essa lista também permite registar novos eventos criados.

Para cada evento da lista é mantido o seu conjunto de subscritores, assim como os métodos que devem ser executados aquando do seu lançamento (um por subscritor), Figura 2.19.

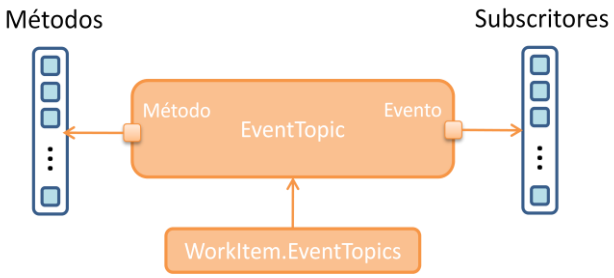

Figura 2.19 – Digrama do mecanismo de eventos

Commands

Um comando é um método que pode ser invocado por mais do que um controlo. Em CAB esse conceito é emulado através da Classe *Command*, que permite criar um *Handler* para um evento que pode estar associado a mais do que um *UIElement*. Esta classe também permite associar múltiplos *Handler*s de comandos a um *UIElement*.

Os comandos permitem desacoplar o código que implementa a funcionalidade das entidades que pretendem invocá-lo. Essa separação permite dividir claramente os deveres das diversas entidades e torna o código mais fácil de manter. Um exemplo banal da utilização de comandos é nos menus, onde a cada botão lhe é associada a respectiva acção. Apenas é necessário criar o comando e associa-lo ao menu, a partir daí a acção implementada será sempre desencadeada quando o menu é acedido.

O *WorkItem* dispõe de um repositório onde estão acessíveis todos os comandos registados, facilitando assim o seu acesso. Esse repositório é conhecido por *Commands* e também permite adicionar novos comandos.

State

A *Framework* de CAB disponibilizou uma funcionalidade que permite a partilha de algumas propriedades entre *WorkItems* pais e filhos. Através desta funcionalidade é possível ao *WorkItem* pai fornecer informação ao seu filho, sem que para isso seja necessário criar uma referência entre os dois. Ao eliminar esta referência as classes tornam-se reutilizáveis.

SmartParts

O *WorkItem* disponibiliza um repositório onde podem ser guardados todos os elementos visuais que compõem a interface com o utilizador.

WorkItems

É uma colecção de elementos que representa os *WorkItems* filho de um determinado *WorkItem*. Este conceito existe para que seja possível criar uma hierarquia de *WorkItems*, que partilham alguns recursos. Na figura seguinte (Figura 2.20) pode-se ser um exemplo de uma hierarquia de *WorkItems*.

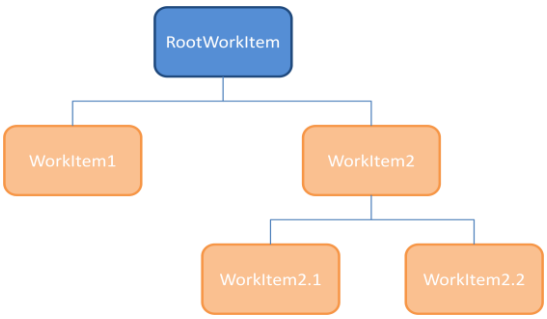

Figura 2.20 – Exemplo da estrutura hierárquica dos WorkItems

Tal como já foi indicado durante a apresentação da *Shell*, o *WorkItem* principal é criado na *Shell* e é dele que derivam todos os outros (Figura 2.20). Quando um *WorkItem* cria um novo filho, ele deve acrescentá-lo à sua lista de filhos (*WorkItems*). Na ligação entre pai e filho nem todas as propriedades são partilhas. Na verdade, os *WorkItems* filho só têm acesso a um conjunto restrito das propriedades do pai. As propriedades do pai a que o filho tem acesso são as seguintes [Pla07]:

- Commands
- EventTopics
- Services
- UIExtensionSites
- WorkSpaces

Caso seja preponderante herdar mais alguma informação, isso deve ser feito através da propriedade *State*.

Items

É uma colecção que o *WorkItem* facilita onde é possível guardar todo o tipo de objectos. Esta colecção é do tipo *object*[2].

Workspaces

Esta propriedade guarda a colecção dos *Workspaces* que foram registados até ao momento. Esta propriedade para além de possibilitar aceder às *Workspaces* já registadas também permite registar novas.

SmartPart(Views)

Normalmente as *Smartparts* são identificadas como *User Controls*, numa aplicação de CAB, contudo podem ser definidas como qualquer elemento visual numa interface de CAB. A construção de *SmartParts* é feita através da alteração de *User Controls* Standard. A *Framework* de CAB foi desenhada para suportar a utilização de raiz de *User Controls* de *Forms*.

[2] Object é um tipo genérico de objectos em C#

A *Framework* pressupõe que normalmente as interfaces estão construídas segundo o padrão *Model-View-Presenter*, embora isso não seja uma restrição.

2.3.1.3 PRISM

"The Composite Application Library provides the foundation for building WPF and Silverlight applications"[Mic09b]

Composite Application é o nome da biblioteca desenvolvida pela Microsoft que pretende agilizar a construção de aplicações em *Windows Presentation Foundation* (WPF) e *Silverlight*. Os ideais desta biblioteca são similares aos que sustentam a *Framework* de CAB. Em ambos os casos pretende-se facilitar a construção de aplicações modulares, em que os seus componentes são desacoplados e cooperam entre si para cumprir tarefas.

A *Composite Application*, mais conhecida por PRISM, rege-se por um conjunto de padrões já testados, que ajudam no decorrer do desenvolvimento de aplicações com interfaces para utilizadores. A utilização desses padrões diminui o tempo de desenvolvimento, uma vez que os padrões definem linhas orientadoras para a resolução daquele tipo de problemas.

Na fase que antecedeu a construção da biblioteca PRISM foram identificados alguns objectivos que seriam fundamentais para o seu sucesso. Os objectivos foram divididos em duas categorias, os de arquitectura e os de desenho. No contexto da arquitectura consideraram-se os seguintes objectivos:

- Incitar o encapsulamento de serviços comuns disponíveis para os diversos módulos, aumentando assim a qualidade das aplicações.
- Promover a reutilização de funcionalidades, recorrendo a uma arquitectura que estimule/facilite esta prática.
- Utilizar uma arquitectura que promova a separação de funções, possibilitando às diversas equipas de desenvolvimento focarem-se em áreas de conhecimento distintas. O ideal seria dividir as aplicações por áreas, como por exemplo, desenho de interfaces, lógica de negócio, construção da infra-estrutura base.
- Impulsionar a construção de aplicações por módulos. Esses módulos devem ser independentes uns dos outros e possibilitar a sua integração em aplicações de WPF ou *Silverlight*.
- A manutenção das aplicações deve ser facilitada, a inclusão de novas funcionalidades não deve implicar um grande esforço.

Já no contexto do desenho as principais metas definidas foram:

- Permitir que os módulos que compõem uma aplicação possam partilhar e aceder a componentes existentes na infra-estrutura.
- Facilitar a adopção de padrões na construção de interfaces para utilizadores em WPF ou *Silverlight*. Esses padrões devem promover a separação da interface da lógica de negócio.
- Equipas distintas devem poder criar novos módulos que depois são facilmente integrados na aplicação do cliente.

2.3.1.3.1 Principais características

Todas as principais características enunciadas para CAB (2.3.1.2.1) também são validas para PRISM, com a excepção do suporte de raiz para *Views* em *Windows Forms*, que no caso de PRISM é para *Views* em WPF ou *Silverlight*.

Para além das características enunciadas em CAB, PRISM oferece um conjunto de novos paradigmas, tornando-o mais versátil e adaptada às necessidades dos desenvolvedores. Na citação seguinte é possível descortinar um dos principais pontos de ruptura entre CAB e PRISM:

"The Composite Application Library was designed in such a way that you can take the services you need to help solve your problem without having to use the entire library." [Mic09b]

Maior flexibilidade da arquitectura

Ao contrário de CAB, em PRISM não existe uma arquitectura tão rígida para as aplicações. É possível utilizar algumas das funcionalidades existentes em PRISM sem necessitar de adoptar toda a filosofia que lhe está subjacente. Por exemplo, caso uma equipa apenas necessitasse de introduzir a lógica de módulos desacoplados numa aplicação, então poderia recorrer ao *EventBroker* existente em PRISM para promover a comunicação. Em CAB não existe essa hipótese, porque o elemento que estabelece a ponte entre o *EventBroker* e a aplicação é o *WorkItem*, que subentende a construção de outras entidades à sua volta.

Extensibilidade

A arquitectura de PRISM promove a extensibilidade de várias formas diferentes [Mic09b]. A estrutura de PRISM é tão flexível que até permite trocar os serviços ou as estratégias definidos por defeito. Desta forma é possível adaptar a estrutura/serviços que ela fornece às necessidades do projecto onde está a ser usada. Este comportamento distingue-se da abordagem seguida em CAB, em que existe uma estrutura rígida e difícil de contrariar.

Suporte para aplicações em ambiente Web e aplicacional

PRISM possibilita a construção de aplicações que funcionam tanto em ambiente *Web* como aplicacional (*Silverlight* Vs WPF). A única diferença entre os dois contextos está no projecto inicial, sendo que num caso tem de ser orientado a *Silverlight* e no outro a WPF. A construção do Core da aplicação fica delegada para os módulos que são carregados, e esses sim são independentes do tipo do projecto.

Existe apenas um detalhe que pode atrapalhar a compatibilidade de um módulo tanto para WPF como para *Silverlight*. Esse detalhe deriva do facto de os controlos existentes numa tecnologia não serem exactamente iguais à outra, havendo casos em que não há correspondência.

2.3.1.3.2 Apresentação da arquitectura disponibilizada em PRISM

Nesta secção é apresentada a arquitectura de PRISM e as suas principais características. Na figura seguinte (Figura 2.21) é possível visualizar os principais componentes de uma aplicação em PRISM.

Figura 2.21 – Componentes de uma aplicação em PRISM

Como se pode verificar na figura anterior (Figura 2.21), em PRISM não existe o conceito de *WorItem*, as suas funções foram distribuídas por outras entidades. A delegação das competências do *WorkItem* em diversas entidades faculta a possibilidade de apenas utilizarmos os serviços necessários. Nos pontos seguintes pode-se ver um paralelismo entre a realidade de CAB e PRISM em relação às competências do *WorkItem*.

- ***SmartParts, Items*** – Em PRISM a entidade responsável por guardar todos os objectos é o *IUnityContainer*, não se faz distinção entre as *Views* e os restantes *items*.
- ***Workspaces*** – Em PRISM o conceito de *Workspace* passou a chamar-se *Region* e a entidade responsável por coordená-los é o *RegionManager*.
- ***EventTopics*** – Em PRISM os eventos não estão por defeito acessíveis, caso seja necessário utilizá-los é necessário requisitá-los/activá-los através da inclusão da classe *IEventAggregator*.
- ***Commands*** – Os comandos em PRISM são obtidos através da infra-estrutura de WPF. Para complementar a infra-estrutura de comandos disponibilizada em WPF PRISM disponibiliza duas novas classes, CompositeCommand e o DelegateCommand. Os comandos não têm nenhum repositório em especial, ao contrário de CAB, é preciso criar uma colecção caso seja necessário.

Em PRISM já não existe nenhuma classe com a clara responsabilidade de coordenar os casos de uso de um módulo, dai que existam duas hipóteses, ou se cria uma classe para satisfazer essas necessidades ou delega-se no *Module* essas funções.

Shell

"The shell is the main window of the application where the primary user interface (UI) content is contained." [Mic09b]

Em PRISM a Shell não representa a aplicação base do projecto, mas apenas o *layout* inicial da interface da aplicação. Na figura seguinte (Figura 2.22) é possível verificar a estrutura da aplicação inicial e a forma como está interligada com os restantes componentes.

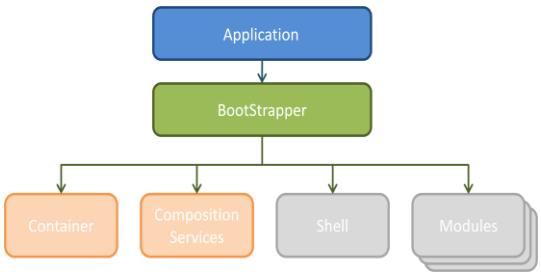

Figura 2.22 – Estrutura de uma aplicação em PRISM

Em primeiro lugar é preciso criar um objecto do tipo *Application*, que é proveniente do *System Windows*. A *Application* por sua vez cria uma instância do tipo *BootStrapper*, que é a entidade responsável pela inicialização de uma aplicação que utilize PRISM. O *BootStrapper* deriva da classe *UnityBootstrapper*, que lhe permite efectuar as inicializações da aplicação recorrendo ao *Unity Container.* O papel do *Container* é preponderante neste contexto, porque é graças a ele que é possível implementar o mecanismo de injecção de dependências. As responsabilidades do *BootStrapper* são as seguintes:

- Configurar o *Container*
- Registar os *Regions Adapters* necessários para cada controlo. Os *Region Adapters* são responsáveis por criar uma região e associá-la a um controlo.
- Criar a *Shell*
- Finalmente inicializar os módulos.

A razão pela qual não é possível criar a *Shell* logo numa primeira instância prende-se com o facto de a *Shell* estar dependente de alguns serviços que só são inicializados pelo *BootStrapper*. Na fase inicial de configuração do *Container* o *BootStrapper* regista um conjunto de serviços que são fundamentais para o bom funcionamento da *Shell* (residentes no *Composition Services*), como por exemplo, o *RegionManager*.

A *Shell* em PRISM representa a interface principal da aplicação, facilitando a injecção das *Views* dos módulos. Os módulos podem injectar *Views* na *Shell* através das regiões que são definidas no *layout* da *Shell*. Na *Shell* é também possível definir elementos da interface gerais a toda a aplicação, como por exemplo, menus ou barra de ferramentas. Caso seja necessário definir um estilo geral a toda a aplicação, a *Shell* é a entidade mais propícia para essa tarefa. A *Shell* tem a capacidade de propagar os estilos, templates e temas definidos para as *Views* que aloja.

Module

"Modules can communicate with other modules and access services in a loosely coupled fashion. They reduce the friction of maintaining, adding, and removing system functionality." [Mic09b]

O conceito de módulo em PRISM é similar ao conceito em CAB. O principal benefício de adoptar uma arquitectura deste género surge ao nível da manutenção do código, já que a tarefa de adicionar, alterar e remover funcionalidades é simplificada. A principal diferença entre CAB

e PRISM neste prisma reside no facto de em PRISM não existir *WorkItems*. A inexistência de *WorkItems* em PRISM faz com que o módulo passe a ter de coordenar os diversos casos de uso que implementa, a menos que se crie uma entidade intermédia para desempenhar essas funções. Em PRISM os módulos implementam a interface *IModule*.

O serviço que permite trabalhar com os módulos encontra-se definido à parte, tal como *EventAggregator* ou o *RegionManager*. Para lidar com os módulos é necessário carregar o serviço *ModuleManager*; através dele é possível carregar os módulos e inicializá-los.

Antes de poder começar a utilizar os módulos é necessário passar por um ciclo de inicialização, as principais etapas desse processo são as seguintes:

- **Descobrir os módulos** - existem três formas de procurar os módulos que compõem uma aplicação. Os módulos podem ser adicionados manualmente em código ou no XAML. Os módulos podem ser procurados num directório predefinido, ou pode ser criado um ficheiro de configurações, à semelhança de CAB, onde estão definidos os módulos que compõem a aplicação.
- **Carregar os módulos** - Depois de preenchido o *ModuleCatalog* com os módulos pertencentes à aplicação, chega a hora de carregar para a memória as *assemblies* que contêm os módulos.
- **Inicializar os módulos** - Por último é necessário criar as instâncias dos módulos e executar o método *Inicialize* de cada um.

O processo de inicialização dos módulos subentende as seguintes rotinas:

- Registar as *Views* e Serviços que o módulo disponibiliza. O registo destes elementos no *Container* possibilita utilizar o mecanismo de *Dependecy Injection*, sendo possível partilhar com outros módulos os elementos registados.
- Registar as *Views* nas respectivas regiões. Através deste processo é possível injectar nas regiões definidas as interfaces dos módulos.
- Integração dos módulos na aplicação. Os módulos precisam de subscrever os serviços/eventos que possibilitam levar a cabo as suas funções.

O diagrama seguinte (Figura 2.23) mostra em maior detalhe os passos envolvidos na procura, carregamento e inicialização dos módulos.

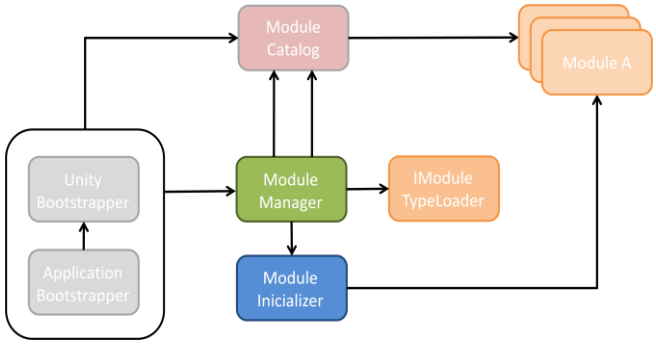

Figura 2.23 – Diagrama de construção de um módulo

Todo o processo começa no *UnityBootstrapper*, esta entidade é responsável por preencher o *ModuleCatalog* com as referências dos módulos que compõem a aplicação. De seguida o *ModuleManager* acede à lista de módulos a carregar e carrega as suas *assemblies*/dependências através do *IModuleTypeLoader*. Por fim é executado o *ModuleInicializar*, que apenas cria uma instância dos módulos e depois executa a operação Inicialize de cada um.

View

"A view encapsulates a portion of your user interface that you would like to keep as decoupled as possible from other parts of the application." [Mic09b]

As *Views* são o componente base da construção de uma interface. A construção de uma interface pode ser realizada através da conjugação de várias *Views*, reutilizando alguns componentes quando é possível. Uma *View* pode ser definida como um *User Control*, um Data Template, ou mesmo um controlo normal. Os *User Controls* utilizados em PRISM normalmente derivam ou de WPF ou *Silverlight*, dependendo do tipo de aplicação que se pretende desenvolver. Toda a lógica de criação, registo e interacção com as *Views* é fornecida por WPF ou *Silverlight*, a única funcionalidade adicional que PRISM forneceu foi o C*ontainer* das *Views*, conhecido por *Regions*. As regiões permitem definir zonas onde é possível injectar outras *Views*, um bocado à semelhança de CAB onde são utilizadas os *Workspaces*.

Framework ou Biblioteca?

Em que filosofia se enquadra PRISM, será que se pode considerar uma *Framework* ou é antes uma Biblioteca? Segundo o que foi dito sobre PRISM e relembrando a comparação entre o conceito de Biblioteca e *Framework*, considera-se que PRISM é um anfíbio, ou seja, pode desempenhar os dois papéis.

Caso se utilize PRISM como um todo, tirando partido de todas as suas funcionalidades e aceitando os seus princípios, então considera-se que PRISM é uma *Framework*. Por outro lado, se apenas se utilizar algumas das funcionalidades de PRISM, como por exemplo o serviço de eventos, então já se considera que é uma Biblioteca.

2.3.2 Tecnologias orientadas à construção de interfaces

Neste subcapítulo são apresentadas as duas tecnologias ponderadas para a construção de interfaces ao longo da fase de análise: WPF e *Windows Forms*.

2.3.2.1 WPF

"The Windows Presentation Foundation (WPF) is an entirely new graphical display system for Windows." [Mac08].

Windows Presentation Foundation (WPF), formalmente conhecido por *Avalon*, é o novo subsistema gráfico do Windows Vista. Este sistema tira partido do investimento já feito pela

Microsoft na *Framework* .NET e possibilita aos programadores habituados à tecnologia .NET uma rápida adaptação ao novo ambiente.[ALMV07]

A tecnologia WPF pretende agilizar a construção de interfaces mais complexas e ricas interactivamente. WPF tem por objectivo combinar as melhores características de sistemas como o *DirectX* (3D e aceleração por hardware), *Windows Forms* (produtividade da equipa de desenvolvimento), *Adobe Flash* (suporte para a criação de animações complexas), e HTML (anotação declarativa e fácil implementação) [Nat06].

WPF introduziu algumas alterações radicais na forma como são abordadas as interfaces gráficas em *Windows*. De entre todas as alterações que foram introduzidas existem cinco princípios base que se destacam, na medida em que são um ponto de ruptura face aos sistemas existentes (por exemplo *Windows Forms*). Os princípios base são os seguintes [ALMV07]:

- **Ampla integração** – WPF integra um conjunto de tecnologias que até então teriam de ser utilizadas independentemente. WPF, para além dos normais controlos de 2D possibilita a utilização de 3D, vídeo e linguagem. Todas estas áreas são disponibilizadas através de um modelo de programação consistente, que permite mapear muitas das técnicas utilizadas num domínio para os restantes (integração forte).
- **Aceleração por Hardware** – Todos os desenhos efectuados em WPF são mapeados através de DirectX. O facto de ser utilizada a tecnologia DirectX permite tirar partido das mais recentes capacidades das placas gráficas.
- **Controlos ricos e adaptáveis** – os controlos de WPF são extremamente flexíveis, permitindo criar configurações nunca antes ponderadas. Por exemplo, é possível criar uma *ComboBox* preenchida com botões animados, ou ainda um menu composto por videoclips. WPF também torna bastante acessível a criação de um conjunto de estilos distintos para uma aplicação.
- **Gráficos vectoriais** – Para tirar partido dos componentes gráficos de hardware, WPF implementou um motor gráfico baseado em vectores. Este motor permite redimensionar as imagens para a resolução específica dos monitores sem ocorrerem perdas de qualidade.
- **Programação declarativa** –WPF apresenta uma nova linguagem baseada em XML, que permite representar interfaces e a interacção com os utilizadores. Esta linguagem é conhecida por XAML e é considerada uma linguagem de anotação.
- **Fácil implantação** – WPF providencia opções para implantar aplicações tradicionais Windows ou para alojar aplicações em browsers de net. Esta funcionalidade já não é recente, em *Windows Forms* já era possível, mas não deixa de ser um componente muito importante da tecnologia. Um novo aspecto interessante é que WPF é construído sobre o conceito de *ClickOnce*, este conceito suporta a integração directa entre um browser de net e o seu sistema de navegação.

Nos subcapítulos seguintes é apresentada a arquitectura de WPF e uma linguagem declarativa (XAML) que permite agilizar a construção de interfaces em WPF.

2.3.2.1.1 Arquitectura de WPF

A arquitectura de WPF assenta essencialmente em três camadas distintas (Figura 2.24), com competências distintas [Mac08].

- No topo da arquitectura está a camada que disponibiliza um conjunto de serviços que possibilitam o desenvolvimento de interfaces. É uma camada de alto nível e está escrita em C#. De entre os vários serviços que disponibiliza pode-se enfatizar: a definição dos tipos de controlos usados em WPF (PresentationFramework.dll) e os componentes visuais a partir dos quais todos os controlos e formas visuais derivam (PresentationCore.dll).
- A camada intermédia é o núcleo do sistema de desenho de WPF. Esta camada é composta por duas componentes: milcore.dll - é responsável por traduzir todos os elementos visuais para a linguagem que o Direct3D espera; WindowsCodecs.dll - é uma API de baixo nível que providencia suporte para imagens.
- Por último surge a camada responsável por concretizar o desenho dos componentes. O Direct3D é uma API de baixo nível através da qual todos os gráficos de WPF são desenhados.

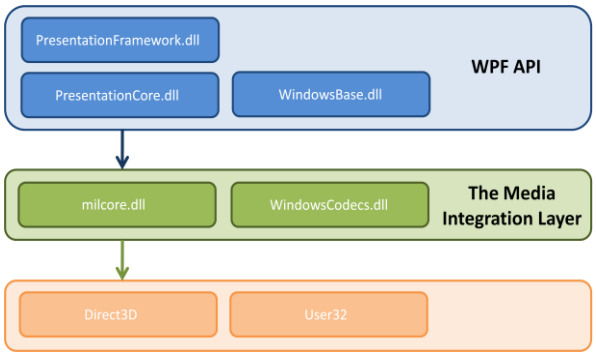

Figura 2.24 – Diagrama da arquitectura de WPF

2.3.2.1.2 eXtensible Application Markup Language

A *eXtensible application Markup Language*, mais conhecida por XAML, é uma linguagem declarativa orientada à construção e inicialização de objectos em .NET. XAML consiste num conjunto de regras que regulam a forma como os parsers/compiladores devem tratar XML, assim como, algumas palavras-chave. XAML sozinho não define nenhum tipo de elementos interessantes, no fundo é apenas uma forma de utilizar as APIs de .NET. Falar de XAML sem uma *Framework* como WPF é como falar de C# sem a .NET *Framework* [Nat06].

Factores que encorajam a utilização de XAML [Nat06]:

- XAML é normalmente a forma mais concisa de representar uma interface ou uma hierarquia de objectos.
- O XAML encoraja a separação da interface da lógica de negócio, permitindo desenvolver aplicações que são mais facilmente mantidas.

A codificação das interfaces produzidas em XAML é facilmente verificável. É possível utilizar uma ferramenta do género *XamlPad* que mostra o resultado da codificação sem ser necessário compilar a interface.

2.3.2.2 Windows Forms

"Windows Forms is a very cleanly designed class hierarchy for building Windows applications." [Whi05]

Windows Forms foi o nome dado à nova API de desenvolvimento de aplicações gráficas construída para complementar a Microsoft .Net *Framework* (Figura 2.25). *Windows Forms* são a tecnologia *Smart Client* para a *.Net Framework*. *Smart Clients* são aplicações ricas graficamente que são fáceis de implantar e actualizar.

Os *Windows Forms* são a unidade base de uma aplicação, sendo essencial dedicar algum tempo ao seu funcionamento e arquitectura. Em última análise *Windows Forms* é a base que os programadores utilizam para construir uma interface. A construção de uma interface é feita através da adição de novos controlos e do código que permite manipular a informação. Um controlo é um elemento discreto de uma interface, que permite mostrar informação ou aceita a sua inserção. O *Windows Forms* contém um conjunto de controlos predefinidos que facilitam a tarefa de construção de interfaces, já que implementam muitos dos componentes que normalmente são necessários (como por exemplo, *Label*, *DataGrid* ou *MenuStrips*). Caso os controlos disponibilizados não sejam do agrado do programador é possível construir novos. Os novos controlos são obrigados a derivar da classe *UserControl* [Mic09d].

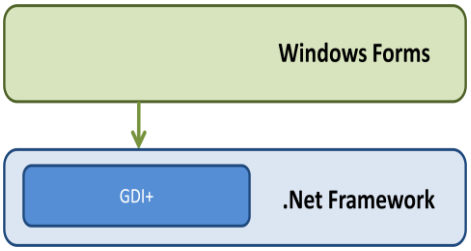

Figura 2.25 – Diagrama da arquitectura de Windows Forms

Microsoft Windows GDI+ é a nova biblioteca gráfica da Microsoft e é baseada na .NET *Framework*. A *GDI+* é uma interface de um dispositivo gráfico que permite aos programadores escrever aplicações que utilizem gráficos e texto formatado [Whi05]. Este dispositivo gráfico permite fazer output da informação quer para o monitor como para a impressora. As aplicações baseadas na *Windows Win32* API não têm acesso directo ao hardware gráfico, por isso a *GDI+* serve de intermediário entre as aplicações e os dispositivos gráficos. Os serviços da *GDI* estão disponibilizados através de um conjunto de classes em *C++*[Mic09c].

2.3.3 Conclusões

O intuito deste subcapítulo não é escolher qual a tecnologia que melhor se adequa ao projecto, pelo contrário, apenas se pretende analisar as propostas existentes. O capítulo 3 é que trata esse assunto e identifica quais foram as opções tomadas a nível tecnológico e o porquê.

Da análise efectuada existem algumas conclusões importantes que se podem retirar, a utilização das *Frameworks* propostas é uma mais-valia para o projecto, na medida em que permitem diminuir o tempo de desenvolvimento e incitam à adopção de boas práticas e padrões.

Por sua vez, as tecnologias propostas para a construção de interfaces também são preponderantes para o sucesso do projecto, já que permitem a construção de interfaces interactivas mais ricas e complexas com menor esforço. Ao utilizar essas tecnologias é possível oferecer ao utilizador uma experiência mais agradável e completa.

Capítulo 3

Análise Tecnológica

Neste capítulo é apresentada a Infra-estrutura de migração que surgiu no decorrer da fase de análise tecnológica. O capítulo está dividido em 7 partes: apresentação do conceito da Infra-estrutura desenvolvida; contextualização do seu surgimento; documentação da Infra-estrutura; documentação de como fazer *refactoring* a uma solução em CAB para usar a Infra-estrutura; explicação de como migrar uma solução em CAB para PRISM usando a Infra-estrutura e conclusões que se podem extrair do seu desenvolvimento.

3.1 Infra-estrutura de Migração

A Infra-estrutura de migração foi desenvolvida com o objectivo de criar uma plataforma que permita construir aplicações independentemente da *Framework* base, seja esta CAB ou PRISM. Através desta nova plataforma é possível desenvolver aplicações para as duas *Frameworks* em simultâneo, assegurando que os programas são compatíveis para os dois casos. Há apenas uma situação que não é resolvida pela Infra-estrutura: a Shell está directamente dependente da *Framework*, assim sendo, é necessário criar entidades distintas. Excepto a situação referida o desenvolvimento das aplicações é efectuado de forma transparente, deixando para a Infra-estrutura todos os detalhes inerentes às duas *Frameworks* base.

Outro ponto importante a salientar é que a Infra-estrutura também permite uniformizar a utilização de *Views*, quer em *Windows Forms* quer em WPF. A Infra-estrutura fornece mecanismos que possibilitam utilizar os dois tipos de *Views*, tanto em CAB como em PRISM. Mais uma vez a sua utilização é transparente, facilitando a reutilização de *Views* já existentes.

Para a construção da Infra-estrutura foi efectuado um estudo prévio das diferenças entre as duas *Frameworks* base. Desse estudo resultou a elaboração de um manual (D) onde é feito um paralelismo entre acções similares nas duas *Frameworks*. Esse manual serve de guia no caso de ser necessário realizar uma migração entre as duas *Frameworks* base. A criação do manual e a construção da Infra-estrutura foi produzida por uma equipa composta pelo autor deste documento e pelo colega Luís Ponte.

3.2 Contextualização da Infra-estrutura de Migração

A Infra-estrutura de migração surgiu no contexto do estudo tecnológico realizado na empresa. O estudo consistiu na análise da viabilidade de adoptar duas *Frameworks* distintas para a construção de aplicações. As duas *Frameworks* propostas foram CAB ou PRISM. No desenrolar desse estudo foi necessário ponderar alguns detalhes importantes, que condicionam directamente a qualidade da solução adoptada e que são enumerados nos pontos seguintes:

- A aplicação a criar não é estanque e deve ser integrada com os restantes produtos da empresa.
- Os restantes produtos desenvolvidos na empresa têm por base a *Framework* CAB.
- As interfaces da nova aplicação devem ser desenvolvidas usando WPF.
- Caso se opte por desenvolver em CAB é preciso testar a eficiência das *Views* em WPF.
- No caso de se optar por PRISM, é preciso avaliar o esforço necessário para migrar os produtos existentes. Será que essa migração pode ser faseada? É possível continuar a utilizar as interfaces existentes em *Windows Forms* (pelo menos numa fase inicial)?

Nos próximos subcapítulos são apresentadas algumas conclusões em relação aos pontos indicados anteriormente.

3.2.1 Inclusão de Views

Neste capítulo é aprofundada a possibilidade de incluir *Views* de WPF em CAB e *Views* de *Windows Forms* em PRISM. Estas questões são abordadas porque CAB foi feito para utilizar *Views* em *Windows Forms* e PRISM foi construído a pensar na tecnologia WPF. Pretende-se assim verificar em primeira instância se é possível concretizar as possibilidades que foram enumeradas e só depois é feita uma análise à sua performance.

Inclusão de *Views* em WPF dentro de CAB

Para mostrar *Views* de WPF dentro de *Windows Forms* é necessário utilizar um software específico, *The Smart Client Software Factory*, que inclui uma nova camada de integração com WPF que estende CAB. Com esta nova camada é possível utilizar os controlos de WPF da mesma forma que se usam as *SmartParts*.

Inclusão de Views em *Windows Forms* dentro de PRISM

Para injectar uma *View* em *Windows Forms* dentro de uma *View* em WPF é necessário utilizar o controlo WinFormsHost, que está disponível em WPF. A biblioteca que se tem de incluir para poder aceder ao controlo é “WindowsFormsIntegration.dll”, que pertence às *assemblies* de WPF.

3.2.1.1 Testar a eficiência de CAB e PRISM a carregar as *Views*

Uma questão essencial na selecção da *Framework* é a eficiência de carregar *Views*, quer em *Windows Forms* quer em WPF. Este subcapítulo apresenta um teste efectuado em ambas as *Frameworks* propostas e analisa os resultados obtidos. A finalidade deste teste é complementar o estudo de selecção de uma das *Framework*.

O teste foi elaborado usando duas plataformas base, uma em CAB e outra em PRISM, que partilham as mesmas *Views*. De seguida é apresentada a estrutura do teste e as *Views* utilizadas.

Composição do teste

O teste é composto por 4 *Views*:

- **Menu** – é uma *View* em *Windows Forms* e contém 3 botões para lançar as restantes *Views*. Os botões são controlos de *Windows Forms*.
- **Opção1** – é uma *View* em WPF e contém 5 controlos. Quatro dos controlos pertencem a uma biblioteca da *Infragistics*[Inf09], enquanto o outro é um botão normal. Os controlos da *Infragistics* são bastante pesados e normalmente demoram mais tempo a carregar que os restantes.
- **Opção2** – também é uma *View* em WPF mas é composta por mais controlos, nomeadamente vinte e três. Existem seis tipos distintos de controlos e todos pertencem à biblioteca da *Infragistics*.
- **Opção3** – é uma *View* em *Windows Forms* e contém um *Container* do tipo *TabControl,* onde estão guardados os restantes controlos.

As *Views* são inseridas dentro de *WorkSpaces* no caso de CAB e em *ContentControl* no caso de PRISM. O objectivo deste teste é medir o tempo que a plataforma demora a mostrar cada uma das *Views*.

De seguida são apresentadas as imagens que ilustram as *Views* usadas neste teste.

A figura seguinte (Figura 3.1) esboça o ecrã inicial da aplicação correspondente à *View* Menu.

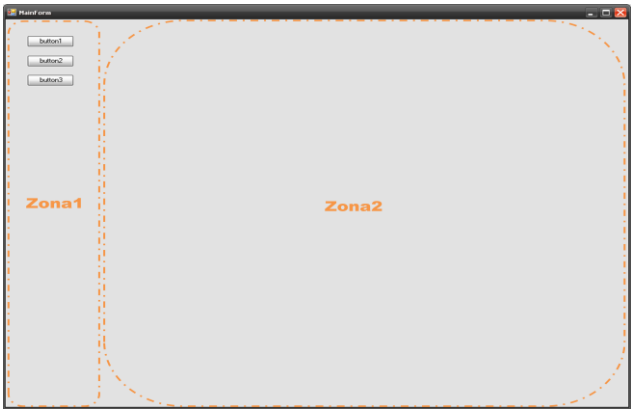

Figura 3.1 – *View* do Menu

Na zona 1 encontra-se a *View* do Menu, enquanto na zona 2 aparecem as restantes *Views*.

A Figura 3.2 apresenta a *View* da Opção1. O objectivo desta *View* é testar a eficiência de WPF com poucos controlos.

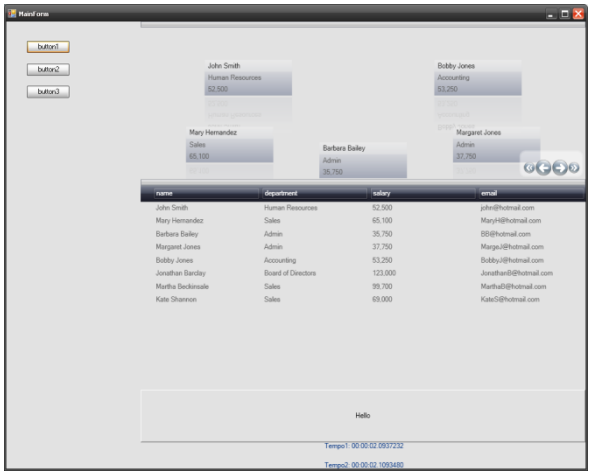

Figura 3.2 – *View* da Opção1

A *View* da Opção2 pode ser deslumbrada na Figura 3.3. O conceito é similar ao da *View* anterior (da Opção1), a diferença é que neste caso o teste é efectuado com um número elevado de controlos.

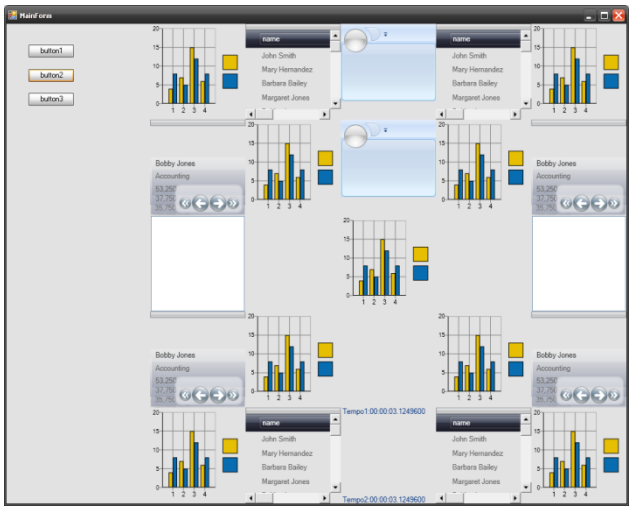

Figura 3.3 – *View* da Opção2

A Figura 3.4 contém a *View* da Opção3. Enquanto as opções anteriores testam a eficiência de WPF, agora o objectivo é testar *Windows Forms*. No caso de *Windows Forms* não é necessário comparar a eficiência entre uma situação com muitos controlos e outra com poucos, porque a grandeza dos tempos é pequena (na ordem das centésimas de segundo).

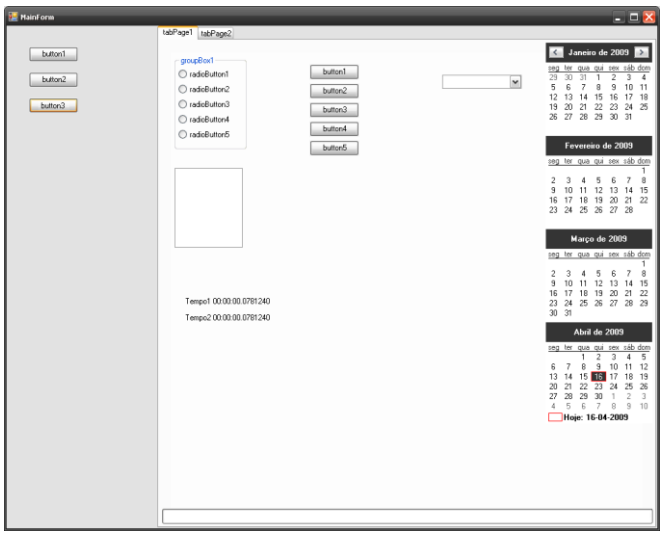

Figura 3.4 – *View* da Opção3

Resultados obtidos

Para análise e discussão dos testes efectuados, são apresentados gráficos relativos aos tempos recolhidos durante a fase de testes. Os tempos são calculados desde o instante em que se carrega no botão, até que o *Load* da página é terminado. Ao medir estes tempos, verificou-se existir uma pequena variação, por isso optou-se por recolher três tempos para cada situação e trabalhar com a média.

A Figura 3.5 apresenta um gráfico onde é efectuada a comparação entre CAB e PRISM no contexto da abertura das *Views* na Opção1 e na Opção2

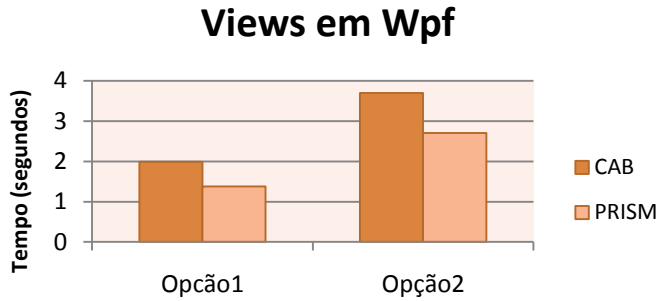

Figura 3.5 – Gráfico de comparação do tempo necessário para abrir uma *View* de WPF em CAB e PRISM

Como se pode verificar, em ambos os casos PRISM é mais rápido. Os valores obtidos são os esperados, já que PRISM foi desenhado para utilizar WPF, enquanto CAB não.

Outra característica que pode influenciar o desempenho de CAB face a PRISM é o facto de em CAB ser possível efectuar atribuições declarativas. O facto de as atribuições serem declarativas obrigam a plataforma de CAB a percorrer todo o código à procura das atribuições efectuadas com *Tags* (palavras reservadas), e só depois é possível concretizar essas atribuições.

A Figura 3.6 apresenta a comparação entre CAB e PRISM em relação ao carregar de uma *View* em *Windows Forms* (Opção 3).

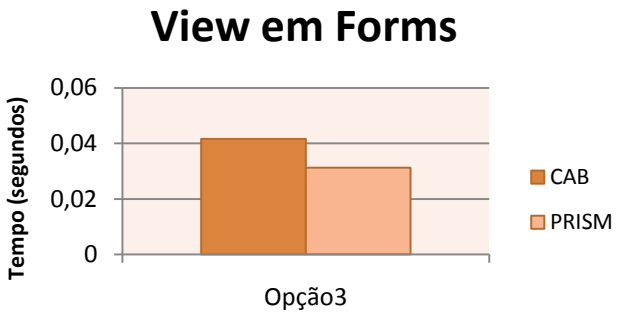

Figura 3.6 - Gráfico de comparação do tempo necessário para abrir uma View de Forms em CAB e PRISM

Surpreendentemente, PRISM consegue ser mais rápido que CAB, mesmo sabendo de antemão que não foi desenhado para desempenhar essa função. Uma das possíveis justificações para esse facto já foi referida no teste anterior; as atribuições declarativas usadas em CAB introduzem uma complexidade adicional, podendo conduzir a perdas na eficiência.

3.2.2 Migração de uma solução em CAB para PRISM

Com a finalidade de verificar a viabilidade de migrar uma solução de CAB para PRISM, foi proposta a elaboração de um teste que consiste na migração de uma pequena amostra de um projecto em CAB, disponibilizada pela Microsoft, para PRISM. A equipa que elaborou o teste foi composta por dois elementos e cada um dos elementos fez a conversão de forma isolada, a fim de conseguir obter uma estimativa mais real e diversificada. Embora já exista um pequeno manual (D) que apresenta os pontos a ser alterados e a forma como se deve proceder, nem sempre o processo é linear. O manual desenvolvido parte do pressuposto que os projectos seguem uma determinada estrutura, mas no exemplo proposto tal não se verifica, pelo que a equipa perdeu algum tempo adicional para contornar essas situações. Houve também algumas situações que não estavam convenientemente pensadas no manual, tendo sido necessário repensá-las e corrigir o manual.

A estimativa temporal final de conversão da amostra proposta foi de dois dias e meio, ficando dentro dos limites estabelecidos inicialmente. Considera-se que depois de eliminar a barreira inicial de adaptação ao processo, a equipa destacada para a tarefa consiga aumentar o ritmo de conversão.

3.2.3 Conclusões do estudo

Os pontos anteriores foram bastante esclarecedores em relação a qual deverá ser o melhor caminho a seguir. Os testes de inclusão de WPF em CAB e de *Windows Forms* em PRISM foram aceitáveis, validando as duas abordagens inicialmente propostas. Contudo, a análise da viabilidade de migrar as soluções existentes na empresa de CAB para PRISM não foi muito

favorável, pelo menos a curto prazo; o esforço e custos serão enormes e implicará congelar a evolução dos projectos durante o processo.

Paralelamente a isto estudou-se também a viabilidade de criar uma *Framework* que permita criar um projecto independentemente da *Framework* base (CAB ou PRISM). Através desta *Framework* é possível migrar de CAB para PRISM, ou vice-versa, de forma quase instantânea. Um dos pontos positivos de uma abordagem deste género é a possibilidade de migrar os projectos existentes para a nova *Framework* de forma faseada. Uma migração faseada permite manter o sistema funcional e ter um melhor controlo sobre todo o processo, possibilitando elaborar testes à medida que o processo vai progredindo (maior confiança na solução final). Outro aspecto relevante é que os produtos migrados para a nova *Framework* continuam a funcionar em CAB, eliminando os problemas de compatibilidade com clientes antigos e simultaneamente podem ser compilados e usados em PRISM.

No âmbito deste projecto, a construção da *Framework* é a solução mais apetecível, já que consegue conjugar duas necessidades muito importantes: a curto prazo funciona com a *Framework* CAB e é possível integrar com os projectos existentes na empresa; a longo prazo, quando os projectos da empresa estiverem disponíveis em PRISM, a solução criada já estará funcional.

3.3 Documentação da infra-estrutura de migração

Neste capítulo é apresentada a Infra-estrutura criada para suportar a migração da *Framework* de CAB para PRISM. O objectivo desta Infra-estrutura é permitir automatizar essa migração, fornecendo uma plataforma que permite emular em PRISM alguns dos conceitos existentes em CAB, que entretanto foram alterados. Ao utilizar esta plataforma torna-se transparente o desenvolvimento de aplicações, independentemente da *Framework* base.

A Figura 3.7 apresenta um diagrama de alto nível da Infra-estrutura.

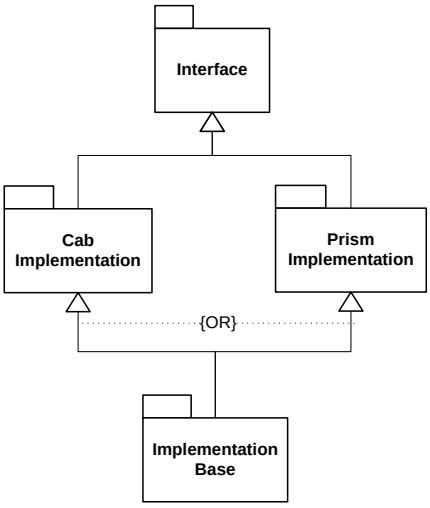

Figura 3.7 – Diagrama de alto nível da Infra-estrutura de migração

Como se pode verificar na figura anterior, foi definida uma interface base (*Interface*) para as duas *Frameworks*, onde é descrita a estrutura base que ambas devem respeitar. Ao derivar as

implementações em CAB e em PRISM dessa interface, é possível assegurar que definem todas as estruturas/procedimentos necessários para assegurar o funcionamento das funcionalidades declaradas na interface. Nos pacotes de CAB e PRISM é onde estão realmente implementadas as funcionalidades, respeitando a *Framework* em questão e as suas particularidades. Por fim, existe o pacote *Implementation Base*, cuja tarefa é permitir incluir as duas *Framework*s em simultâneo, podendo mais tarde escolher em que *Framework* se pretende compilar a aplicação.

Nos subcapítulos seguintes são apresentadas em maior detalhe as principais classes da Infra-estrutura, evidenciando as distinções entre CAB e PRISM.

3.3.1 ModuleInit

As diferenças entre CAB e PRISM neste contexto surgem principalmente em dois aspectos: o *ModuleInit* deriva de classes distintas para as duas vertentes; as componentes que têm de incluir para suportar a realização das acções que lhe estão inerentes são diferentes. Esses dois aspectos podem ser facilmente constatados no diagrama apresentado na Figura 3.8.

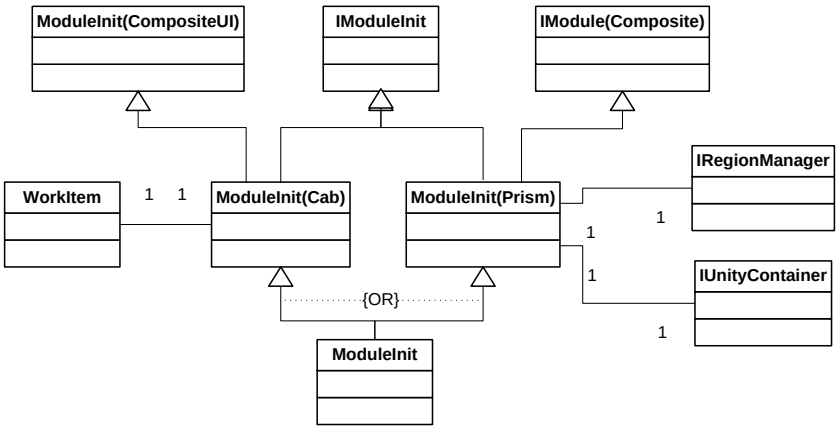

Figura 3.8 – Diagrama da estrutura de um ModuleInit

O *ModuleInit* de CAB deriva da classe *ModuleInit* da *CompositeUI*, enquanto o módulo de PRISM deriva da classe *IModule* do *Composite*. Em relação às componentes que incluem, o *ModuleInit* de CAB apenas necessita de um *WorkItem*, que suporta todas as operações necessárias, enquanto que o *ModuleInit* em PRSIM precisa de um *IRegionManager* e de um *IUnityContainer*.

3.3.2 Controller

A entidade *Controller* foi criada porque em PRISM não existe o conceito de *WorkItem*, que existe em CAB. Para tornar o processo de migração transparente foi necessário criar uma classe que desempenhasse o mesmo papel. A implementação do *Controller* em PRISM implica reunir todas as entidades responsáveis por executar as acções inerentes ao *WorkItem*, enquanto em CAB é essencialmente derivar do *WorkItem* (ver Figura 3.9).

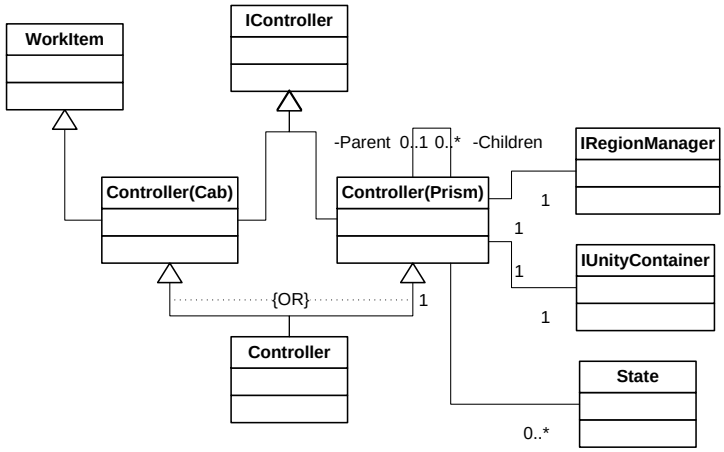

Figura 3.9 – Diagrama da estrutura de um Controller

Ao observar o diagrama anterior pode-se verificar que o controlo de PRISM inclui: um conjunto de estados (*state*), funcionalidade inerente ao conceito de *WorkItem*, em que é possível guardar um conjunto de propriedades; um *IUnityContainer*, onde é possível registar as *Views*, novos *Controllers* e outros dados que sejam necessários; *IRegionManager*, onde são registadas todas as regiões onde é possível injectar *Views*; um conjunto de *Controllers* que derivam de si (os seus filhos). Para além do que foi referido, o *Controller* de PRISM também inclui uma lista de *WorkSpaces*, já que essa classe é uma propriedade de *Windows Forms* e não existe nenhuma entidade responsável por esse domínio em PRISM. De forma análoga também existe no *Controller* de CAB uma lista de regiões, promovendo assim a transparência no momento de inserir uma *View* num local (*WorkSpace* ou *Region*).

A Figura 3.10 ilustra a hierarquia de *WorkItems*, propriedade que é necessário simular no *Controller* de PRISM.

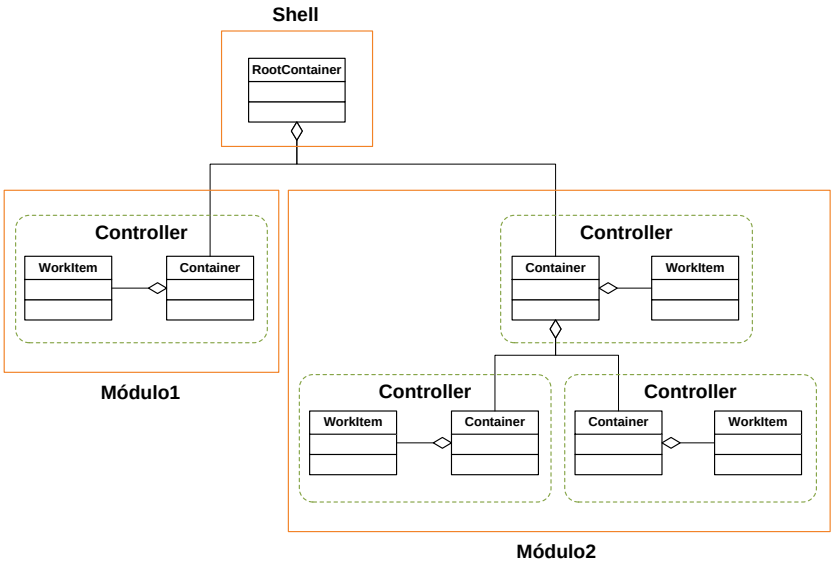

Figura 3.10 – Simulação da hierarquia de *WorkItems* usando *Containers*

Sempre que um *WorkItem* cria um filho este fica dependente dele (pai) e pode herdar algumas das suas propriedades, como por exemplo, estados e *WorkSpaces*. A forma encontrada

para transpor a propriedade de hierarquia para PRISM foi através do uso de um *Container*, sendo que sempre que é criado um novo *Controller* é-lhe associado um novo *Container* que deriva do seu pai.

Ao nível das duas implementações da Infra-estrutura foi necessário desenvolver uma funcionalidade que permitisse guardar todos os *handlers* de comandos que um *Controller* instalou. Essa funcionalidade foi criada para permitir remover todas as referências para os comandos quando o *Controller* é eliminado. Em CAB esta funcionalidade já está assegurada caso se use atribuição declarativa.

O *Controller* também faculta um conjunto de funções que permitem adicionar *Views* em *Windows Forms* ou WPF de forma transparente, inserindo-as dentro de uma *Region* ou *WorkSpace* (previamente registado). Fruto dessa funcionalidade é possível utilizar as *Views* de forma transparente, potenciando a interoperabilidade dentro das duas *Frameworks*.

3.3.3 Presenter

A função do *Presenter* é implementar toda a lógica de negócio da *View*, assim sendo este necessita de uma referência para a interface da *View* para poder proceder a sua actualização. O Presenter necessita também de uma ligação para o *Controller*, permitindo-lhe assim instalar comandos e eventos. O diagrama seguinte (Figura 3.11) permite visualizar estas características.

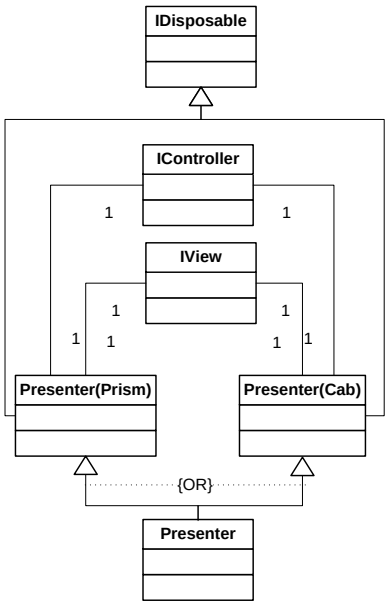

Figura 3.11 – Diagrama da estrutura de um Presenter

A interface *IDisposable* é implementada tanto em PRISM como em CAB para fornecer a possibilidade de os utilizadores efectuarem alguma operação antes de o *Presenter* ser completamente eliminado. A implementação em CAB por si só já remove o *Presenter* do *Controller* onde foi criado.

3.3.4 WinFormsUserView e WPFUserView

Um dos principais requisitos desta plataforma é conseguir trabalhar em CAB ou em PRISM com *Views* em WPF e em *Windows Forms*. É importante realçar que *Windows Forms* é orientado para CAB e WPF é orientado para PRISM. Para conseguir atingir essa meta foi necessário criar uma generalização das Views em *Windows Forms* e WPF, para poder trabalhar as suas especificidades nas duas *framework*s. A figura seguinte (Figura 3.12) ilustra o que foi referido anteriormente.

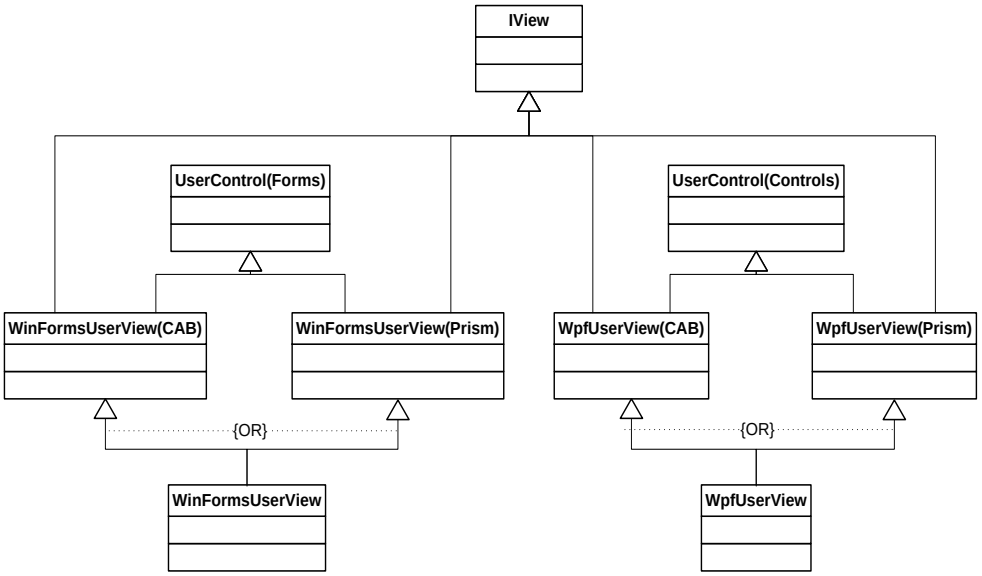

Figura 3.12 – Diagrama da estrutura de uma WinFormsUserView e uma WPFUserView

A principal diferença digna de destaque na comparação da implementação em CAB e em PRISM de uma WpfUserView é a forma como são registadas as regiões existentes na *View*. Em ambos os casos é feita manualmente no código do *Controller*, desaparecendo a sua sintaxe do XAML. No caso das *WinFormsUserView* ocorre uma situação similar, a única diferença é que só no caso de PRISM é que é preciso registar manualmente as *WorkSpaces*.

Uma propriedade da *View* que não pode ser aferida no diagrama é o facto de no momento da sua criação gerar um novo *Presenter* e ficar com uma referência para ele.

3.3.5 CommandBroker

O objectivo do *CommandBroker* é criar uma entidade que fique totalmente responsável por trabalhar com comandos e permita funcionar com eles de forma fácil e transparente, independentemente da *Framework*. O diagrama presente na Figura 3.13 ilustra as diferenças entre a implementação do *CommandBroker* em CAB e PRISM.

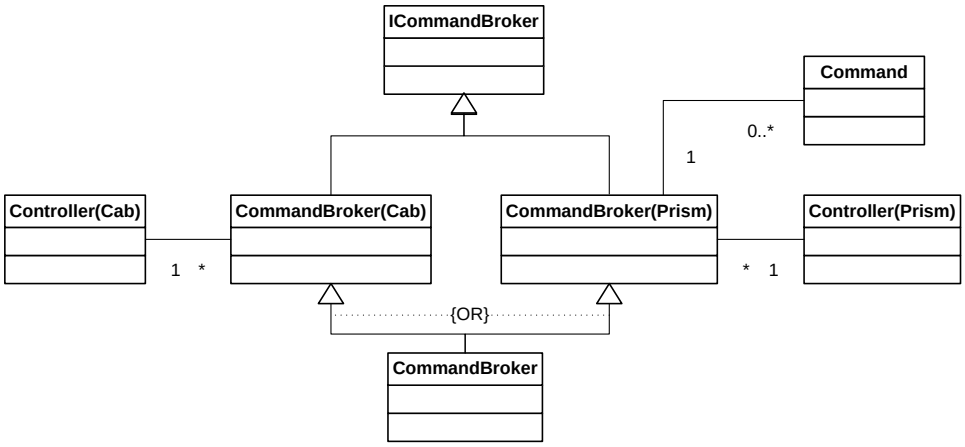

Figura 3.13 – Diagrama da estrutura de um CommandBroker

Enquanto a implementação em CAB contém um *Controller* (*WorkItem*) que facilita todos os mecanismos necessários para funcionar com comandos, em PRISM isso não se passa. No caso de PRISM, para além do *Controller* é necessário criar uma lista de comandos, global a toda a solução, onde vão sendo guardados os comandos criados.

3.3.6 Event

A classe *Event* foi criada porque em PRISM os eventos são obrigados a implementar uma interface padrão. Assim, para promover a transparência e possibilitar uma conversão directa entre as duas *Frameworks*, procedeu-se a esta estruturação. O diagrama seguinte (Figura 3.14) identifica facilmente esta questão.

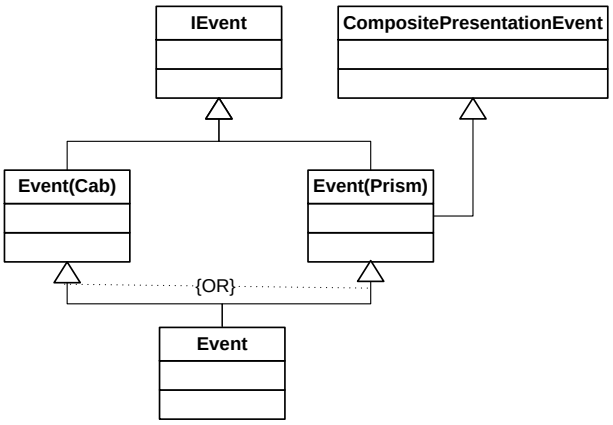

Figura 3.14 – Diagrama da estrutura de um Event

3.3.7 EventBroker

Tal como o próprio nome sugere esta classe foi criada com o intuito de fornecer uma entidade que permita trabalhar com os eventos de forma transparente, camuflando as diferenças entre as duas *Frameworks*. A Figura 3.15 mostra algumas diferenças entre as duas implementações.

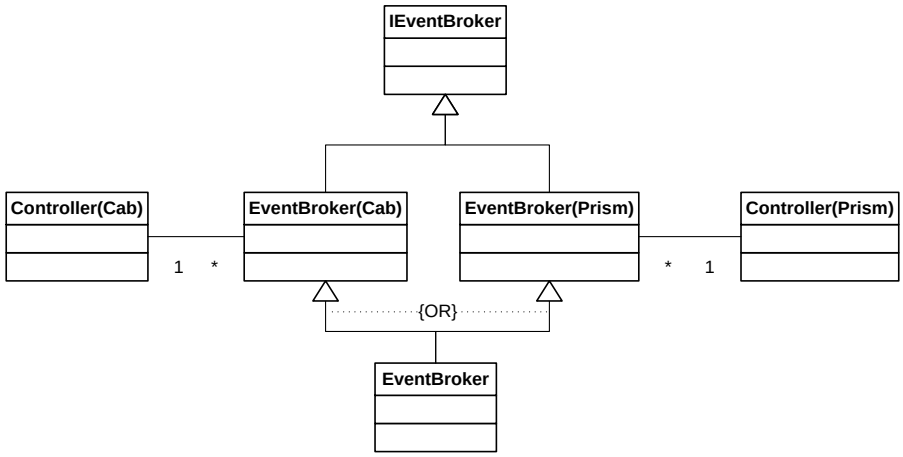

Figura 3.15 – Diagrama da estrutura de um EventBroker

A grande diferença entre as duas implementações é simplesmente o local onde estão registados os eventos, cada uma das *Frameworks* tem uma identidade distinta para tratar desta funcionalidade.

3.4 Refactoring de CAB para a Infra-estrutura

Neste capítulo é demonstrado todo o processo de *Refactoring* necessário num projecto de CAB para passar a utilizar a nova Infra-estrutura. A demonstração está dividida pelos seguintes tópicos:

- *ModuleInit*
- *WorkItem*
- *Commands*
- *Views*
- *Events*

Em cada um dos tópicos são apresentados extractos de código que pretendem evidenciar as transformações a efectuar ao código em CAB. Esses extractos de código são acompanhados por alguns comentários, de modo a enfatizar as principais diferenças entre a implementação em CAB e a implementação na nova Infra-estrutura (designado como *refactoring* nos subcapítulos que se seguem).

3.4.1 ModuleInit

A tabela seguinte (Tabela 3.1) apresenta uma analogia entre a declaração de um módulo em CAB e recorrendo à Infra-estrutura.

Tabela 3.1 – Paralelismo entre código do *ModuleInit* em CAB e na Infra-estrutura

CAB	`public class ExampleModuleInit : ModuleInit`
	`{`
	`  public override void Load()`
	`  {`
	`  MyWorkItem NewWorkItem =`
	`  rootWorkItem.WorkItems.AddNew<MyWorkItem>();`
	`  NewWorkItem.Run(ProjectConstants.CONTENT_WORKSPACE);`
	`  }`
	`}`

REFACTORING	`public class ExampleModuleInit : ModuleInit`
	`{`
	`  public override void Load()`
	`  {`
	`  MyWorkItem NewWorkItem =`
	`this.createWorkItem<MyWorkItem>();`
	`  NewWorkItem.Run(ProjectConstants.CONTENT_WORKSPACE);`
	`  }`
	`}`

Em primeiro lugar é importante salientar que nos dois extractos de código fornecidos na tabela anterior (Tabela 3.1), embora o *ExampleModuleInit* aparente derivar da mesma classe, fruto de o nome ser o mesmo, isso não é verdade. Nos dois casos a classe *ModuleInit* pertence a implementações distintas.

Ao analisar o método *Load* pode-se verificar que a única diferença reside na forma como é gerado o novo *WorkItem*. Em CAB é necessário recorrer ao *RootWorkItem* para obter uma nova instância, já na nova Infra-estrutura essa função passa a ser delegada para o próprio módulo.

3.4.2 WorkItem

No caso do *WorkItem* é necessário analisar três situações em concreto:

- Como criar/lançar um *WorkItem*
- Como criar/mostrar uma *View* a partir de um *WorkItem*
- Como injectar estado num *WorkItem*

Considera-se que o código fornecido nos pontos seguintes está a ser executado dentro de um *WorkItem*.

Como criar/lançar um *WorkItem*

A Tabela 3.2 exemplifica como migrar a criação e o lançamento de um WorkItem.

Tabela 3.2 – Paralelismo entre o código de criação/lançamento de um WorkItem em CAB e na Infra-estrutura

CAB	`MyWorkItem myWorkItem = this.WorkItems.AddNew<MyWorkItem>();` `myWorkItem.Run("tabWorkspace1");`
REFACTORING	`MyController myWorkItem = this.createNewController<MyController>();` `myWorkItem.Run("tabWorkspace1“);`

O objectivo da tabela anterior (Tabela 3.2) é responder à questão de como criar/lançar um novo *WorkItem*. Tal como se pode verificar, existem essencialmente duas diferenças, o tipo do *WorkItem* a criar e a forma como é criada a nova instância. Em CAB o tipo do *WorkItem* tem de derivar da classe *WorkItem* (pertence à Microsoft.Practices.CompositeUI), enquanto na Infra-estrutura tem de derivar do *Controller* (pertence à GlinttHS.MigrationInfrastructure.Implementation).

Como criar/mostrar uma *View* a partir de um *WorkItem*

A Tabela 3.3 demonstra como migrar a criação e apresentação de uma *View* a partir de um *WorkItem*.

Tabela 3.3 - Paralelismo entre o código que permite criar/mostrar uma *View* a partir de um *WorkItem* em CAB e na Infra-estrutura

CAB	`IMyView view = this.Items.AddNew<MyView>();` `Workspaces[place].Show(view);`
REFACTORING	`IMyView view = this.CreateView<MyView>();` `this.ShowView(view, place);`

A forma como é gerada uma nova instância de uma *View* em CAB é bastante similar à da Infra-estrutura. A única coisa que se altera no código é a sintaxe. Enquanto em CAB é necessário aceder aos *Items* do *WortItem* e executar a função *AddNew*, na Infra-estrutura basta aceder ao *Controller* e executar o *CreateView*.

Para mostrar a *View* já existem alterações mais significativas. Em CAB é preciso aceder ao *Container* onde estão guardados os *WorkSpaces*, seleccionar o local que nos interessa (recorrendo ao nome) e só depois é possível mostrar a *View*. Recorrendo à Infra-estrutura, apenas é necessário aceder ao *Controller* e executar o método Show (recebe a *View* e o nome do local onde se pretende injectar).

Como injectar estado num *WorkItem*

A Tabela 3.4 apresenta a analogia entre CAB e a Infra-estrutura em relação à injecção de estado num *WorkItem*.

Tabela 3.4 – Paralelismo entre a injecção de estado num *WorkItem* em CAB e na Infra-estrutura

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

REFACTORING	`[NewState("mensagem")]`
	`public string StateMsg`
	`{`
	`  get { return stateMsg; }`
	`  set`
	`  {`
	`  stateMsg = value;`
	`  }`
	`}`
	`//No construtor...`
	`public myWorkItem()`
	`{`
	`  fatherWorkItem.InjectMyState(this);`
	`}`

A injecção de estados num *WorkItem* altera-se ligeiramente. Enquanto em CAB bastava apenas definir uma propriedade no *WorkItem* e indicar que essa propriedade derivava de um atributo do estado do seu pai (através da *tag State*, seguida do nome do estado), na Infra-estrutura isso não basta. Na Infra-estrutura para além do processo inicial enunciado em CAB, em que a única diferença é a alteração da *tag* de *State* para *NewState*, também é necessário executar o procedimento *InjectMyState* no construtor do WorkItem. O *InjectMySate* é responsável por resolver as dependências das propriedades assinaladas com a *tag NewState*.

A classe onde é injectado o estado não está apenas restringido aos *WorkItems*, é possível injectar estados em qualquer classe. Por exemplo, caso se pretende herdar numa *View* algumas propriedades do *WorkItem* é possível utilizar o mesmo procedimento.

3.4.3 Commands

No âmbito dos comandos (*commands*) é necessário ponderar três situações:

- Como registar um *handler* para um comando?
- Como associar uma acção de um objecto a um comando?
- Como executar um comando?

Para executar acções sobre comandos recorrendo à nova Infra-estrutura é necessário criar um objecto do tipo *CommandBroker*. Ao longo das situações enumeradas anteriormente poder-se-á constatar este facto, já que todas as operações sobre comandos são executadas por intermédio do *CommandBroker*. Ao criar uma nova instância desse objecto é necessário fornecer um *WorkItem*, isso deve-se ao facto de internamente ser através dele que se consegue aceder aos comandos.

Como registar um *handler* para um comando

A Tabela 3.5 apresenta a forma de migrar um handler de um comando.

Tabela 3.5 – Paralelismo entre registar o *handler* de comando em CAB e na Infra-estrutura

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

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

Na tabela anterior (Tabela 3.5) pode-se constatar as alterações necessárias para migrar um comando de CAB para a Infra-estrutura. O passo inicial é remover a *tag* que antecede a declaração do *handler*, já que na Infra-estrutura é necessário registar o *handler* de forma manual. De seguida é preciso criar o CommandBroker.Uma vez criado o *CommandBroker* já é possível proceder à instalação, para isso basta apenas executar a acção *InstallHandler* e passar-lhe o nome do comando e o respectivo *handler*.

A assinatura do *handler* também sofre algumas alterações. Depois de efectuado o *refactoring*, o *handler* deve apenas receber um parâmetro do tipo *EventArgs*.

Como associar uma acção de um objecto a um comando

A Tabela 3.6 apresenta a correspondência entre CAB e a Infra-estrutura em relação à associação de um comando a um botão.

Tabela 3.6 – Paralelismo entre a associação de um comando a um botão em CAB e na Infra-estrutura

CAB	`workItem.Commands["button1Command"].AddInvoker(button1, "Click");`
REFACTORING	`ICommandBroker cmdBroker = new CommandBroker(workItem);` `cmdBroker.BindInvoker(button1, "Click", "button1Command");`

O mapeamento do procedimento que associa uma acção de um objecto a um comando é trivial, como se pode verificar na tabela anterior (Tabela 3.6). Após a criação do *CommandBroker* só é preciso executar a função *BindInvoker* e passar-lhe o objecto ao qual se pretende associar o comando, o nome da acção e o nome do comando.

Como executar um comando

A Tabela 3.7 exibe como migrar a execução de um comando.

Tabela 3.7 – Paralelismo entre a execução de um comando em CAB e na Infra-estrutura

CAB	`workItem.Commands["button1Command“].Execute();`
REFACTORING	`ICommandBroker cmdBroker = new CommandBroker(workItem);` `cmdBroker.ExecuteCommand("button1Command“);`

Tal como foi referido na introdução dos comandos, a entidade *CommandBroker* é responsável por executar todas as operações sobre comandos. Partindo desse pressuposto, para executar um comando basta apenas chamar a função *ExecuteCommand* e passar-lhe o nome do comando.

3.4.4 Views

Este ponto abrange dois tipos de *Views*, as que têm origem em *Windows Forms* e as de WPF. Em ambos os casos, ao criar uma nova *View* está subjacente a criação de um novo *Presenter*. O tipo do *Presenter* que a *View* cria é definido no cabeçalho da *View*.

Definição da classe (Forms)

A Tabela 3.8 apresenta o paralelismo entre CAB e a Infra-estrutura na declaração de uma *View* de *Windows Forms*.

Tabela 3.8 – Paralelismo entre a declaração de uma *View* de *Windows Forms* em CAB e na Infra-estrutura

CAB	`[SmartPart]`
	`public partial class MyView : UserControl, IMyView`
	`{ ...`

REFACTORING	`public partial class MyView: WinFormsUserView< MyViewPresenter >,`
	`IMyView`
	`{ ...`

Ao nível das *Views* em *Windows Forms* existem duas alterações que é necessário elaborar. Em primeiro lugar retira-se a *tag* existente antes do cabeçalho da função. Em segundo lugar altera-se o nome da classe da qual a *View* descende (*WinFormsUserView*).

Definição da classe (WPF)

A Tabela 3.9 apresenta como migrar a declaração de uma *View* em WPF.

Tabela 3.9 – Paralelismo entre a declaração de uma *View* de WPF em CAB e na Infra-estrutura

CAB	`public partial class MyView : UserControl, IMyView` `{ ...`
REFACTORING	`public partial class MyView: WpfUserView< MyViewPresenter >, IMyView` `{ ...`

Tal como nas *Views* de *Windows Forms*, em WPF também é necessário alterar o nome da classe da qual a *View* descende. A diferença é que neste caso a *View* descende de *WpfUserView*, como se pode verificar na tabela anterior (Tabela 3.9).

Para além das alterações na definição da classe, no caso das *Views* em WPF ainda é preciso proceder a algumas alterações ao nível do XAML da *View* (ver Tabela 3.10).

Tabela 3.10 – Alterações efectuadas ao XAML da *View* em CAB para usar a Infra-estrutura

CAB	`<UserControl x:Class="Module.Views.MyView" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">`
REFACTORING	`<my:WpfUserView x:Class="Module.Views.MyView" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:my="clr-namespace:GlinttHS.MigrationInfrastructure.Implementation;assembly=GlinttHS.MigrationInfrastructure.ImplementationBase" xmlns:z="clr-namespace:Module" x:TypeArguments="z:MyViewPresenter">`

As alterações necessárias são as seguintes:

1. Alterar o tipo do controlo, passa de "`UserControl`" para "`my:WpfUserView`"
2. Estabelecer o caminho para a definição do "`my:WpfUserView`", "`xmlns:my="clr-namespace:GlinttHS.MigrationInfrastructure.Implementation;assembly=GlinttHS.MigrationInfrastructure.ImplementationBase"`"
3. Definir o tipo do *Presenter* que está associado à *View* "`x:TypeArguments="z:MyViewPresenter"`"
4. Indicar onde está guardada a definição do *Presenter* "`xmlns:z="clr-namespace:Module"`".

3.4.5 Events

À semelhança dos comandos, os eventos (*events*) também têm uma entidade responsável por tratar de todas as suas operações, o *EventBroker*. Fruto dessa abordagem é necessário transpor todos os eventos definidos em CAB para a nova filosofia. Os tópicos que têm de ser revistos são os seguintes:

- Como publicar um evento?
- Como subscrever um evento?
- Como tratar o *StateChanged*?

Como publicar um evento

A Tabela 3.11 apresenta como migrar a publicação de um evento.

Tabela 3.11 – Paralelismo entre a publicação de um Evento em CAB e na Infra-estrutura

CAB	`WorkItem.EventTopics["MyEvent"].Fire(this, new EventArgs(), null, EventPublication.Global);`

REFACTORING	`// Defining the event`
	`public class MyEvent : Event`
	`{`
	`  //...`
	`}`
	`// Publishing`
	`EventBroker broker = new EventBroker(WorkItem);`
	`broker.fireEvent<MyEvent>("MyEvent", sender, new EventArgs());`

Ao analisar a tabela anterior (Tabela 3.11) é possível apurar as diferenças entre a implementação em CAB e na nova Infra-estrutura. A primeira diferença surge ao nível da publicação do evento. Em CAB essa operação é efectuada recorrendo ao *WorkItem*, enquanto na Infra-estrutura é feita utilizando o *EventBroker*. Na Infra-estrutura é preciso definir uma classe para o evento que derive de *Event*, já que na opção que despoleta o evento ("fireEvent") é preciso passar o tipo do evento, o nome, a origem, e os argumentos.

Como subscrever um evento

A Tabela 3.12 mostra como migrar a subscrição de um evento.

Tabela 3.12 – Paralelismo entre a subscrição de um evento em CAB e na Infra-estrutura

CAB	`// Subscribing`
	`[EventSubscription("MyEvent")]`
	`public void OnMyEvent(object sender, EventArgs e)`
	`{`
	`  //...`
	`}`

PRISM	`// Subscribing`
	`EventBroker broker = new EventBroker(WorkItem);`
	`broker.subscribeEvent<CopyEvent>(ProjectConstants.COPY_EVENT, OnCopyEvent);`
	`// Defining the handler`
	`public void OnCopyEvent(object sender, EventArgs e)`
	`{`
	`  //...`
	`}`

Para subscrever um evento na nova Infra-estrutura é necessário criar uma instância do *EventBroker* e só depois executar a operação *subscribeEvent*. O *subscribeEvent* recebe o tipo do evento, o seu nome e o *handler* do evento.

Como tratar o *StateChanged*

A Tabela 3.13 mostra como migrar a subscrição do evento *StateChanged*.

Tabela 3.13 – Paralelismo entre a subscrição do evento *StateChanged* em CAB e na Infra-estrutura

CAB	`// Subscribing`
	`[StateChanged("mensagem")]`
	`public void OnMessageChanged(object sender, StateChangedEventArgs e)`
	`{`
	`//...`
	`}`

REFACTORING	`// Subscribing`
	`FatherWorkItem.StateChanged += new EventHandler<StateChangedEventArgs>(OnMessageChanged);`
	`public void OnMessageChanged(object sender, StateChangedEventArgs e)`
	`{`
	`//...`
	`}`

Quando um objecto herda uma propriedade (um estado) de um *WorkItem* e pretende ser informado quando ocorrem alterações a essa propriedade, então necessita de subscrever o evento *StateChanged* desse *WorkItem*. A diferença entre a subscrição desse evento em CAB e na Infra-estrutura dá-se ao nível da instalação do *handler*. Enquanto em CAB basta colocar uma *tag* antes do *handler*, a informar que se pretende subscrever o *StateChanged*, na Infra-estrutura é preciso aceder ao *WorkItem* e adicionar-lhe um novo *EventHandler*. O novo *EventHandler* recebe o *handler* que se pretende instalar.

3.5 Migrar um projecto de CAB para PRISM usando a Infra-estrutura

Uma vez efectuado o passo descrito no subcapítulo anterior, a migração de um projecto em CAB para PRISM torna-se numa tarefa simples. A figura seguinte (Figura 3.16) é uma boa ilustração dos passos necessários para efectuar o processo.

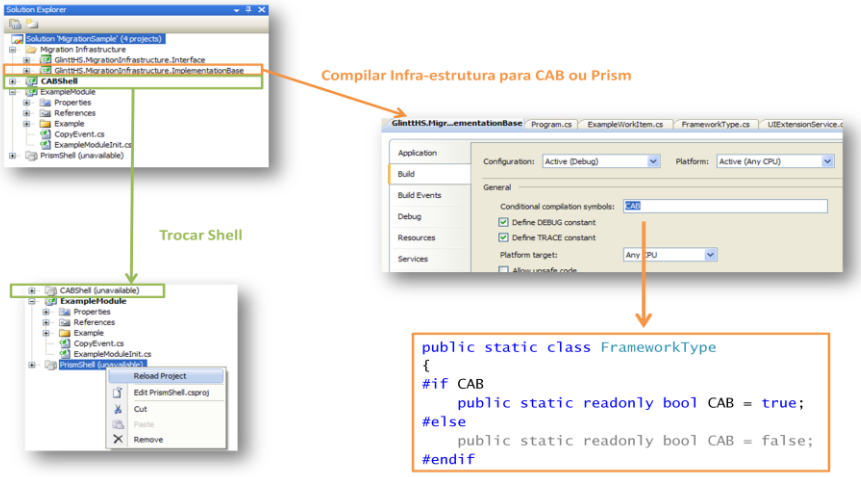

Figura 3.16 – Como migrar um projecto de CAB para PRISM usando a Infra-estrutura

Em primeiro lugar é necessário trocar a *Shell* que está activa, já que essa é uma das propriedades do projecto que não é coberta pela Infra-estrutura. Como já foi referido anteriormente, é necessário criar uma *Shell* independente para CAB e para PRISM. Para alternar a *Shell* activa é necessário desactivar (*Unload*) a de CAB e activar (*Reload*) a de PRISM.

Em segundo lugar é necessário trocar a flag de compilação do projecto "ImplementationBase" da Infra-estrutura. Caso a flag "CAB" pertença à lista dos símbolos condicionais de compilação, então toda a solução será compilada para CAB, caso contrário, será compilada para PRISM. No projecto "ImplementationBase" foi definida uma classe (*FrameworkType*) que é responsável por captar o valor dessa propriedade, assim, as decisões podem ser tomadas em relação a essa classe.

3.6 Testar a eficiência da Infra-estrutura

Uma vez criada a Infra-estrutura é necessário testá-la e compará-la face a soluções criadas de raiz com as duas *Frameworks*. Estes testes têm por objectivo verificar a viabilidade de utilizar a Infra-estrutura de migração, já que não é suficiente que funcione, é preciso assegurar que não introduz ineficiências consideráveis no processo.

Os testes efectuados estão divididos em duas categorias, numa primeira fase compara-se um projecto elaborado com a Infra-estrutura de migração face a projectos elaborados com as *Frameworks* originais de raiz (utilizando as mesmas *Views* nos dois casos). Numa segunda fase, utiliza-se o mesmo projecto construído com a Infra-estrutura e compara-se a versão compilada para PRISM com a de CAB.

Nos pontos seguintes são apresentados todos os valores recolhidos ao longo dos testes efectuados. As *Views* utilizadas para os testes seguem a mesma estrutura das apresentadas no subcapítulo0. A *View WpfSimples* corresponde à Opção1, a *View WpfComplexa* corresponde à Opção2 e a *View* View*Forms* corresponde à Opção3. Os valores apresentados nos gráficos são uma média de três amostragens, já que os resultados apresentavam alguma variabilidade (embora baixa).

3.6.1 Comparar a Infra-estrutura com as soluções de raiz

O teste em questão está dividido em duas partes, a comparação da Infra-estrutura com uma solução CAB e a comparação da Infra-estrutura com uma solução PRISM. Em ambos os casos é comparado um projecto construído com a *Framework* de raiz em cada uma das plataformas (CAB e PRISM), face ao seu homólogo construído usando a Infra-estrutura (compilado para a mesma *Framework)*.

3.6.1.1 CAB

A Figura 3.17 apresenta os resultados obtidos ao calcular o tempo necessário para mostrar uma *View* em WPF.

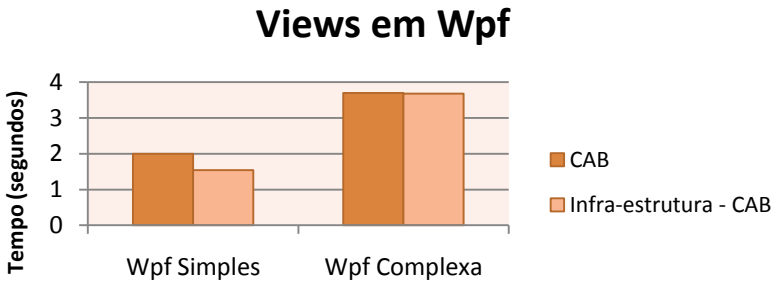

Figura 3.17 – Comparação da abertura de *Views* de WPF com a Infra-estrutura ou com um projecto de raiz em CAB

A figura anterior (Figura 3.17) demonstra que a utilização da Infra-estrutura para a produção de aplicações em CAB com *Views* em WPF é uma solução viável. Ao analisar o gráfico presente na Figura 3.17, pode-se verificar que existe um ligeiro aumento de eficiência no uso da implementação CAB. Esse aumento é mais notório quando o número de controlos na *View* é menor. O facto de o ganho de eficiência ser menos notório quando se usam mais controlos, explica-se porque em tal situação consegue-se diluir melhor as perdas de eficiência causadas pela atribuição declarativa usada na *Framework* de raiz de CAB.

A Figura 3.18 apresenta os resultados obtidos ao calcular o tempo necessário para mostrar uma *View* em *Windows Forms*.

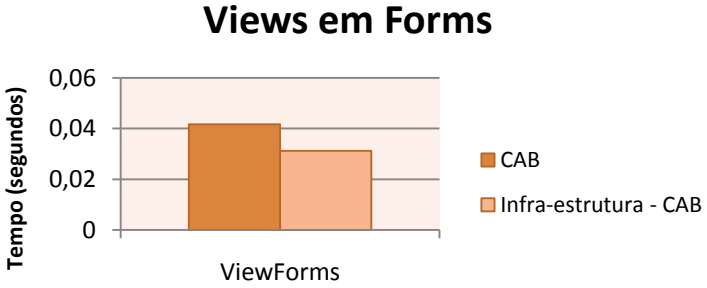

Figura 3.18 - Comparação da abertura de *Views* de *Windows Forms* com a Infra-estrutura ou com um projecto de raiz em CAB

No caso das *Views* em *Windows Forms* a situação é similar, o comportamento apresentado no gráfico anterior (Figura 3.18) segue o mesmo padrão descrito para as *Views* de WPF.

3.6.1.2 PRISM

A Figura 3.19 apresenta os resultados obtidos ao calcular o tempo necessário para mostrar uma *View* em WPF.

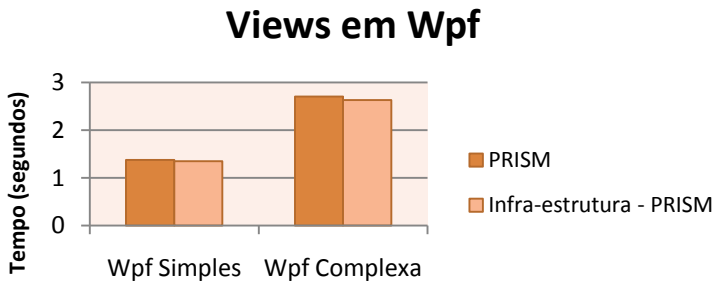

Figura 3.19 - Comparação da abertura de *Views* de WPF com a Infra-estrutura ou com um projecto de raiz em PRISM

O gráfico da figura anterior (Figura 3.19) mostra que não existe uma diferença significativa entre a solução criada com a Infra-estrutura ou com a *Framework* de raiz de PRISM. O facto de a solução criada com a Infra-estrutura de migração ser subtilmente melhor pode dever-se ao facto de a organização do código em módulos ter sido melhor conseguida nesse projecto.

A Figura 3.20 apresenta os resultados obtidos ao calcular o tempo necessário para mostrar uma *View* em *Windows Forms*.

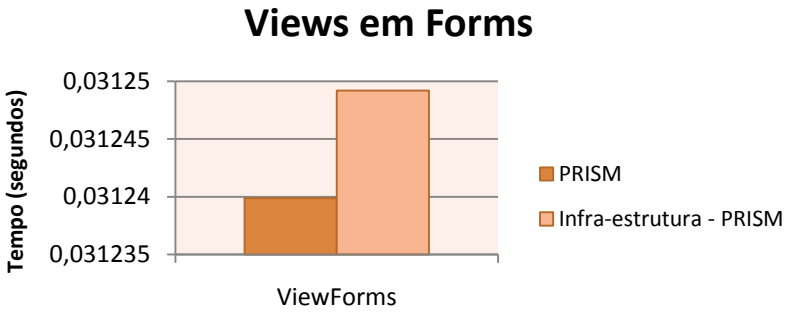

Figura 3.20 - Comparação da abertura de *Views* de *Windows Forms* com a Infra-estrutura ou com um projecto de raiz em PRISM

No caso das *Views* em *Windows Forms* a diferença entre as duas abordagens é praticamente nula. A única explicação encontrada para o valor não ser nulo, pode ser pelo facto de os resultados apresentados derivarem da média de três amostragens.

3.6.2 Comparar soluções CAB e PRISM

Depois da comparação efectuada no ponto anterior pode-se afirmar que a Infra-estrutura de migração é uma boa opção para o desenvolvimento de aplicações. A sua utilização não traz custos adicionais, pelo contrário, na maior parte dos casos aumenta ligeiramente o desempenho e permite uma maior flexibilidade das soluções (funcionam tanto em CAB como em PRISM).

Falta agora comparar o desempenho da Infra-estrutura quando compilada para PRISM ou CAB. A figura seguinte (Figura 3.21) apresenta um gráfico que reporta os valores recolhidos ao longo dos testes efectuados com *Views* em WPF.

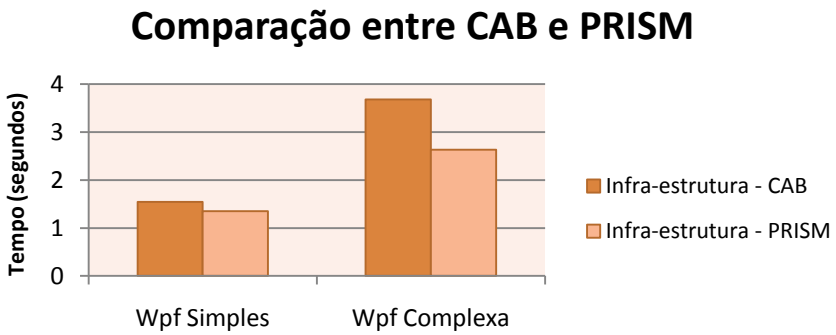

Figura 3.21 – Comparar a abertura de *Views* em WPF usando a Infra-estrutura compilada em PRISM ou em CAB

Os resultados ilustrados no gráfico anterior vão de encontro ao esperado, já que a *Framework* PRISM foi desenvolvida para alojar *Views* em WPF. À medida que o número de controlos da *View* aumenta, a diferença de desempenho entre as duas *Frameworks* torna-se mais evidente, em consequência da complexidade da *View*.

A figura seguinte (Figura 3.22) apresenta um gráfico que reporta os valores recolhidos ao longo dos testes efectuados com uma *View* em *Windows Forms*.

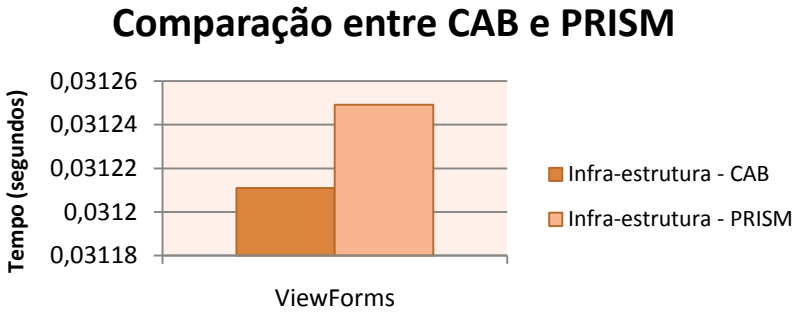

Figura 3.22 - Comparar a abertura de *Views* em *Windows Forms* usando a Infra-estrutura compilada em PRISM ou em CAB

Também neste caso, as conclusões que se podem extrair do gráfico anterior são as esperadas. A Infra-estrutura quando compilada para CAB consegue maior desempenho a carregar *Views* em *Windows Forms*. Embora o desempenho em CAB seja maior que em PRISM

a diferença entre elas é pouco significativa, encorajando a migração de todos os projectos em CAB para PRISM.

3.7 Conclusões

A Infra-estrutura construída é uma componente fundamental para o desenrolar do projecto na empresa, uma vez que permite a integração da solução com os restantes componentes da empresa. Esta Infra-estrutura permite dar uma resposta capaz a uma das principais questões levantada no inicio do estágio: "qual será a *Framework* mais indicada (entre CAB e PRISM) para a construção do projecto, tendo por base a situação actual da empresa (soluções em CAB)?". A resposta não foi trivial e implicou uma longa análise, já que se pretendia aproveitar o desempenho e as facilidades de uma nova *Framework* e ao mesmo tempo não se podia esquecer o compromisso com as soluções existentes (a componente de integração era essencial).

A Infra-estrutura pode ser vista numa perspectiva bem mais alargada do que somente o âmbito do projecto, já que na óptica da empresa será um forte aliado na evolução tecnológica. Através desta Infra-estrutura vai ser possível migrar todos os projectos existentes em CAB para PRISM de uma forma sustentada e faseada. O facto de a Infra-estrutura suportar a compatibilidade com CAB permite que todo o processo de migração possa ser feito por etapas e testado, permitindo que as soluções continuem funcionais no decorrer de toda a operação.

A Infra-estrutura de migração também permite numa primeira fase reutilizar as *Views* existentes em *Windows Forms*, possibilitando aproveitar as sinergias da empresa na migração da lógica de negócio dos produtos. Numa fase posterior o objectivo da empresa é actualizar as interfaces dos produtos, aproveitando todas as potencialidades fornecidas pela tecnologia WPF.

Paralelamente ao processo de migração, as equipas de desenvolvimento podem continuar as suas rotinas de trabalho habituais, devendo nesta fase já recorrer à Infra-estrutura. Embora não seja óbvio as vantagens recorrentes deste facto são enormes, na medida em que é possível continuar a incrementar funcionalidades aos sistemas actuais, estando essas funcionalidades já preparadas para ser integradas no futuro.

Os gastos estimados em formação das equipas para utilizar a nova Infra-estrutura são baixos, já que as diferenças de sintaxe entre o sistema actual e o sistema agora criado são mínimas (tal como foi discutido no subcapítulo 3.4). Por outro lado, toda a formação necessária será fornecida por equipas internas, reduzindo bastante os custos inerentes.

Capítulo 4

Descrição detalhada do projecto

Este capítulo apresenta em detalhe as características do projecto desenvolvido. O capítulo está dividido em cinco partes: apresentação detalhada do problema; definição dos requisitos funcionais e não funcionais do projecto; apresentação dos casos de uso; descrição da arquitectura lógica e física do projecto e por último a integração com a solução da empresa.

4.1 Apresentação detalhada do problema

Tal como já foi referido na introdução, este projecto consiste na construção de uma aplicação que permita agendar um protocolo médico. Para começar é importante entender o conceito de protocolo, na medida em que é a base de toda a lógica da aplicação. Na sua vertente mais abstracta um protocolo é simplesmente um plano de combate a uma patologia. Na verdade a definição mais fidedigna de protocolo, passa por definir a sua composição.

Um protocolo é composto por um conjunto de actividades e restrições, que permitem condicionar o tratamento de um paciente. As restrições definidas no protocolo pretendem evidenciar a forma como as actividades se relacionam e permitem restringir o seu comportamento. No fundo são as restrições que traçam o plano de tratamento, indicando quando e como deve ser tratado o paciente.

As actividades definidas num protocolo podem ser de vários tipos: medicação, exames, consulta, radiação, procedimentos hospitalares, etc. As actividades estão organizadas num protocolo segundo sessões, que por sua vez estão organizadas por ciclos de tratamento. Um protocolo é assim composto por vários ciclos de tratamento. Na figura seguinte (Figura 4.1) é possível constatar a relação entre o protocolo, os ciclos, as sessões e as actividades.

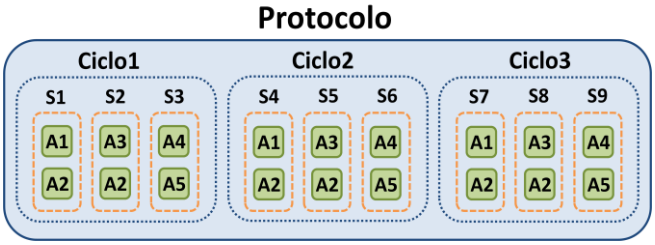

Figura 4.1 – Constituição de um protocolo

Na figura anterior (Figura 4.1) as sessões estão representadas por "S" e as actividades por "A". No contexto de um protocolo um ciclo de tratamento representa uma fase do mesmo. Após o doente cumprir um ciclo de tratamento é feito um ponto de situação para verificar o seu estado. Caso o estado seja favorável o tratamento avança, caso contrário é cancelado e opta-se por outra alternativa. Normalmente os ciclos de um protocolo são iguais, embora não exista nenhuma restrição a esse nível. Como já foi dito, um ciclo é composto por um conjunto de sessões, sendo que cada uma delas representa um dia de tratamento. Cada sessão esta relacionada com um dia e contém as actividades a realizar.

Uma vez compreendidos os principais conceitos do projecto é chegada a hora de começar a detalhar o problema do agendamento. O problema do agendamento deriva da necessidade dos hospitais em ter um mecanismo que permita agilizar a tarefa de marcação de protocolos. O agendamento de um protocolo implica alocar um conjunto de recursos que satisfaçam o conjunto de actividades que compõem esse protocolo. Nem sempre a tarefa de encontrar um conjunto de vagas, nos recursos do hospital, que respeitem as restrições do protocolo é fácil. A maior dificuldade deve-se ao facto de uma vaga num hospital, na maior parte das vezes, ser composta pela conjugação de um recurso material com um recurso humano. Os recursos de um hospital estão divididos em duas categorias, os recursos humanos e os materiais. Na categoria dos recursos humanos, enquadram-se os médicos, os enfermeiros ou os técnicos de saúde. Na categoria dos materiais incluem-se as camas, macas, cadeiras, salas, entre muitos.

Para facilitar a tarefa de agendamento de protocolos foi planeado um conjunto de componentes que pretendem simplificar essa tarefa. O mecanismo de agendamento de protocolos proposto é composto por três componentes:

- Um módulo de agendamento de protocolos
- Um algoritmo que permite agendar automaticamente um protocolo
- Um módulo de gestão de conflitos.

O módulo de agendamento de protocolos é vocacionado para os médicos e permite marcar automaticamente um protocolo. Para levar a cabo esta tarefa, o módulo de agendamento recorre a um algoritmo de agendamento automático, que permite verificar se existe algum conjunto de vagas que respeitem as restrições do protocolo a agendar. Antes de o médico pedir para agendar um protocolo necessita de especificar um conjunto de propriedades que restringem o seu panorama de marcação. Essas propriedades definem o espectro temporal da marcação e o número de ciclos que precisam de ser marcados logo de inicio (por defeito deve ser apenas agendado o primeiro ciclo). Caso o algoritmo de agendamento automático não consiga retornar

uma reposta válida para o pedido, então deve encaminhar a marcação do protocolo para o módulo de gestão de conflitos.

O algoritmo que permite agendar automaticamente um protocolo, isto é, procurar um conjunto de vagas que satisfaçam as restrições do protocolo, deve conjugar um conjunto de heurísticas que lhe permitam encontrar rapidamente uma solução. O tempo de resposta deste algoritmo é crucial, porque no contexto onde vai ser utilizado de nada serve o facto de se conseguir encontrar uma solução, se não o fizer em tempo útil. Quando o médico está a marcar um protocolo precisa saber de imediato se tem vaga ou não, pois por um lado tem o paciente à espera de resposta e por outro lado o seu tempo é demasiadamente valioso. Durante a pesquisa de uma solução, o algoritmo deve ter em conta alguns factores que se pretendem optimizar: minimizar o tempo que os pacientes passam num hospital, melhorando a qualidade de vida dos pacientes; e aumentar a taxa de ocupação dos recursos, aumentando as receitas do hospital.

Por último, surge o módulo de gestão de conflitos. O módulo de gestão de conflitos é responsável por marcar os protocolos que numa primeira instância não conseguiram ser marcados no módulo de agendamento de protocolos. Este módulo deve disponibilizar uma lista de todos os protocolos em espera para agendamento. A partir dessa lista o utilizador selecciona arbitrariamente um protocolo e é efectuada a sua marcação manual. Para efectuar a marcação o módulo deve fornecer um mecanismo onde são visualizados os recursos e a sua disponibilidade, possibilitando depois ao utilizador efectuar a ligação entre as tarefas e as vagas (marcação). Durante o processo de afectação das tarefas às vagas, o sistema deve garantir que as restrições estabelecidas pelo protocolo são cumpridas. No caso de ser realmente impossível agendar o protocolo, devido á falta de vagas, o utilizador deve ter a possibilidade de forçar a sua marcação, contrariando assim as indicações do sistema.

4.2 Requisitos do projecto

Este subcapítulo pretende dar a conhecer todo o processo de análise e levantamento de requisitos efectuado, assim como o seu produto final, a lista de requisitos. Tal como se esperava o processo foi turbulento e bastante complexo, muito devido ao facto de o cliente ainda não ter uma ideia concreta do que realmente pretendia. Embora se esperasse que o processo fosse complexo, a estimativa temporal efectuada falhou redondamente. A fase de análise e validação de requisitos ocupou grande parte do projecto.

4.2.1 Processo de análise e validação

Na fase inicial do projecto foram estabelecidas algumas metas que com o decorrer da fase de análise se foram alterando. No início do projecto foi definido como objectivo o desenvolvimento de três componentes: o módulo de agendamento de protocolos; o algoritmo de agendamento e o módulo de gestão de conflitos.

No decorrer do projecto, e tendo em conta as prioridades definidas pelo cliente, a lista de objectivos alterou-se. O algoritmo de agendamento de protocolos foi colocado em segundo plano, uma vez que a empresa já tinha uma solução testada. A solução existente não cumpre todos os requisitos estabelecidos, no entanto, consegue responder aos mínimos exigidos.

Entende-se por mínimos a capacidade do algoritmo fornecer um conjunto de vagas que respeitem as restrições, sem ter de se preocupar com questões de optimização. Pretende-se que no futuro em vez de construir um algoritmo de agendamento de raiz, seja melhorado o existente. Para esse efeito já existe um plano que define as características a melhorar. Entre estas características, a mais relevante diz respeito à necessidade de obter soluções que optimizem um determinado critério (diminuição do tempo de espera do paciente ou aumento da taxa de ocupação dos recursos).

Para além da componente relativa ao algoritmo de agendamento, também os outros dois módulos sofreram uma reestruturação completa. Para facilitar a compreensão da dimensão das alterações, é apresentada uma pequena descrição destes dois módulos, no início do projecto e na fase actual (perspectivando-se mais alterações no futuro).

Início do Projecto

- **Módulo de agendamento** – o módulo de agendamento deve permitir especificar um conjunto de características fundamentais para o processo de agendamento de um protocolo. A principal característica é o tempo. Antes de o médico executar a acção de agendamento, deve estipular a data, ou o período temporal, que pretende para a marcação. Caso o módulo não consiga encontrar uma solução com as restrições definidas, então o médico tem duas opções: ou redefinir as características e voltar a tentar a marcação, ou forçar a marcação e enviar um pedido para o módulo de gestão de conflitos.
- **Módulo de gestão de conflitos** – O módulo de gestão de conflitos recebe todos os pedidos de agendamento que não foram resolvidos numa primeira instância pelo módulo de agendamento. O módulo disponibiliza uma listagem de todos os protocolos em lista de espera e permita efectuar a sua marcação individual. Para cada um dos módulos, o utilizador, neste caso os Administrativos ou os enfermeiros, deve proceder à marcação do protocolo de forma manual. O utilizador deverá ter acesso a um ecrã onde é possível verificar as actividades do protocolo e as vagas disponíveis. Esse mesmo ecrã deve permitir estabelecer a associação entre os elementos, concretizando assim o agendamento. Caso o sistema indique que não existem vagas para uma determinada actividade, o utilizador dever ter o poder de forçar a sua marcação, ficando esta à sua responsabilidade.

Fase actual do projecto

- **Módulo de agendamento** – o módulo de agendamento de protocolos passou a designar-se módulo de prescrição e agendamento de protocolos. O módulo passou a ter a responsabilidade de prescrever e depois agendar um novo protocolo a um paciente. Este facto alterou radicalmente a lógica de negócio, já que agora é necessário efectuar todos os mecanismos necessários para prescrever um protocolo e validar a sua atribuição. A prescrição de um protocolo subentende a selecção de um paciente e depois a escolha de um protocolo. Uma vez escolhido o protocolo é necessário preencher um conjunto de características que o caracterizam, como por exemplo: a data pretendida para agendamento; o posto de tratamento e o contexto do doente. Antes de ser possível

agendar um protocolo é necessário validar um conjunto de critérios, que permitam verificar se o paciente está apto para ser submetido ao protocolo. A validação existe por questões de segurança e permite alertar para possíveis falhas técnicas, como por exemplo, o médico não reparar que o paciente é alérgico a determinada substância. Quando todos os detalhes referidos anteriormente estiverem devidamente concluídos, então o médico pode finalmente requisitar a marcação do protocolo. A partir desta fase o módulo segue um comportamento análogo ao definido na fase inicial.

- **Módulos de gestão de conflitos** – também neste módulo ocorreram alterações significativas a nível conceptual. O módulo passou de um simples componente de resolução de conflitos, para um componente em que é possível também marcar e alterar os restantes ciclos de um protocolo. Resumindo, o módulo de gestão de conflitos passou a ser o centro das marcações, sendo agora conhecido por Módulo de marcação. Uma vez feito o agendamento do período inicial do protocolo (que pode ocorrer neste ou no módulo anterior), todos os restantes ciclos serão marcados neste módulo. Assim sendo, este módulo ganha outro relevo, já que se torna numa aplicação central no contexto hospitalar. O ecrã inicial do módulo contém todos os protocolos em lista de espera, quer por estarem em estado pendente, como à espera da marcação dos ciclos sucessivos. Depois de escolhido um protocolo é apresentado um ecrã de marcação similar ao descrito na fase inicial.

Um conceito importante que foi introduzido no decorrer do projecto é a noção de tarefa mestre. Cada um dos protocolos tem à partida definida uma tarefa mestre que condiciona toda a marcação de um protocolo. Uma vez encontrada disponibilidade para marcar a tarefa mestra, todas as outras são marcadas, mesmo que para isso seja necessário forçar a sua marcação. Este conceito foi introduzido porque normalmente num protocolo existe uma tarefa que necessita de um conjunto de recursos mais escassos, logo, a prioridade é agendá-la.

Técnicas usadas para identificar e validar os requisitos

Muitas vezes a troca de ideias com o cliente é difícil sendo necessário utilizar um conjunto de ferramentas que as permitam esclarecer. Nesse âmbito foi adoptada a construção de protótipos. Os protótipos numa primeira fase são deita-fora e depois evolutivos. A negociação com o cliente foi morosa e conduziu a uma panóplia de protótipos (D). Embora a construção de protótipos implique gastar muito tempo e congelar a evolução do produto, mostrou ser a única forma eficaz de conseguir levar o projecto a bom porto.

A etapa de prototipagem foi composta por duas fases distintas: os protótipos deita-fora, e os protótipos evolutivos [Som04]. Os protótipos deita-fora não são funcionais, mas apenas ilustrativos, normalmente são concretizados em papel ou num formato de apresentação (por exemplo power point). Os protótipos evolutivos são funcionais e pretendem reaproveitar todo o esforço realizado, podendo ser aproveitados para o desenvolvimento da aplicação.

Numa primeira fase foram validados os conceitos mais gerais da aplicação, como as funcionalidades a inserir, o ciclo de funcionamento da aplicação, a composição das interfaces e alguns detalhes de usabilidade do produto. Na segunda fase foram validados exaustivamente os detalhes definidos na primeira, verificando assim se estavam realmente em concordância com o

pretendido. Para além disso também foram aprofundadas as questões de usabilidade da aplicação. O cliente foi convidado a interagir com o protótipo e a criticá-lo, estabelecendo assim um elo de ligação com o projecto, fazendo-o sentir parte integrante. O facto de o cliente estar embrenhado no projecto é um factor importante para o seu sucesso, porque diminui a probabilidade de haver alterações de última hora. O cliente tem conhecimento das últimas evoluções e no extremo poderá ser um dos seus principais impulsionadores e defensores.

Para além da prototipagem também foram utilizados nesta fase de análise e validação de requisitos os convencionais diagramas de casos de uso, que permitem documentar de forma eficaz e clara o que ficou acordado com o cliente.

4.2.2 Requisitos funcionais

Nesta secção são apresentados os requisitos funcionais da aplicação. Os requisitos serão agregados por módulos, facilitando assim a sua percepção e leitura.

Módulo de prescrição e agendamento

A tabela seguinte (Tabela 4.1) contém o conjunto de requisitos funcionais do módulo de prescrição e agendamento, acompanhados de uma pequena descrição.

Tabela 4.1 – Requisitos funcionais do módulo de prescrição e agendamento

	Nome	Descrição
01	Adicionar novo protocolo ao paciente	O sistema permite que o médico escolha um novo protocolo (de uma lista predefinida) e o adicione ao paciente.
02	Validar os critérios de inclusão e exclusão	Antes de o médico poder concretizar o agendamento de um protocolo, necessita confirmar os critérios de inclusão e exclusão que o protocolo define.
03	Validar se o médico pode prescrever o protocolo	O médico só pode prescrever protocolos para os quais está autorizado. Durante o processo de adição de novos protocolos, o médico só tem acesso aos que tem permissão.
04	Validar se o novo protocolo é compatível	Depois de seleccionado um novo protocolo para um paciente, o sistema tem de validar se o paciente está apto para o cumprir. O paciente pode não estar apto por duas razões: incompatibilidade com outros protocolos em curso, ou conflito com alguma característica do paciente (alergias, estados clínicos, etc.).
05	Preencher as características do protocolo	Antes de agendar um protocolo há um conjunto de características associadas ao protocolo que precisam ser preenchidas, nomeadamente: data pretendida; tempo útil; posto de tratamento e contexto do doente.
06	Agendar protocolo	Depois de todas as validações feitas e de as características do protocolo estarem preenchidas, o sistema fornece a opção de agendar o protocolo. Ao agendar um protocolo o sistema procura

		um conjunto de vagas que o satisfaçam e caso existam reserva-as.
07	Fazer pedido de marcação	Caso a opção de agendar o protocolo não tenha sucesso, o médico terá a opção de registar um pedido no sistema, requisitando que o protocolo seja marcado na data pretendida.
08	Guardar Protocolo	Esta opção permite guardar o novo protocolo associado ao cliente. Só quando o protocolo é gravado é que as vagas reservadas no agendamento são marcadas.
09	Cancelar novo Protocolo	Durante a prescrição de um novo protocolo, o sistema permite que seja cancelada a sua atribuição, desde que ocorra antes da operação de guardar protocolo.

Módulo de Marcação

A marcação de uma tarefa mestre equivale à marcação de uma sessão, porque uma sessão só tem uma tarefa mestre e existe sempre uma tarefa mestre por sessão.

A tabela seguinte (Tabela 4.2) contém o conjunto de requisitos funcionais do módulo de marcação, acompanhados de uma pequena descrição.

Tabela 4.2 - Requisitos funcionais do módulo de marcação

	Nome	Descrição
01	Listar os protocolos em espera	O sistema facilita uma listagem de todos os protocolos em espera para marcação. Os protocolos em lista de espera, tanto podem ser para marcar o ciclo inicial, como para marcar os ciclos subsequentes.
02	Filtrar a listagem dos protocolos	O sistema permite que o utilizador possa filtrar a listagem dos protocolos pelo campo que pretender, utilizando a restrição correspondente. Por defeito o sistema deve apresentar um conjunto de filtros padrão. Os filtros padrão existentes são os seguintes: a iniciar tratamento; sem nada agendado; doentes de hoje e doentes em fim de ciclo.
03	Agregar a listagem dos protocolos por um campo	O utilizador tem a possibilidade de agrupar as linhas da tabela pelo campo que pretender, facilitando assim a visualização da informação.
04	Escolher campos da tabela	O utilizador pode escolher os campos a visualizar na listagem dos protocolos.
05	Escolher protocolo para realizar a marcação	Ao efectuar duplo clique sobre uma linha da tabela, o sistema abre a funcionalidade de marcação do protocolo.
06	Mostrar grelha de actividades	A grelha de actividades mostra uma perspectiva geral do agendamento, indicando os dias em que as várias tarefas já estão marcadas. A grelha de actividades obtém essa informação através

		da conjugação da marcação das sessões e as restrições do protocolo.
07	Mostrar grelha de recursos	O sistema mostra uma grelha com a disponibilidade de cada um dos recursos disponíveis para a tarefa mestre. Essa grelha possibilita a marcação das tarefas (sessões).
08	Mostrar resumo das sessões marcadas	O sistema mostra uma grelha que contém os dias em que foi marcada a tarefa mestre (sessões). Para cada dia marcado apresenta a hora da marcação e o tipo do recurso associado.
09	Sincronizar grelha de recursos com as restantes	Quando ocorre uma alteração na grelha de recursos é necessário actualizar as restantes grelhas.
10	Marcar uma tarefa na grelha de recursos	O sistema possibilita ao utilizador marcar uma tarefa mestre num dia à escolha, partindo do pressuposto que cumpre as restrições do protocolo. Uma tarefa mestre está directamente associada a uma sessão.
11	Desmarcar tarefa na grelha de recursos	O utilizador pode desmarcar uma tarefa (sessão) na grelha. Ao desmarcar uma tarefa, desmarca consequentemente todas as sessões posteriores (desmarcação em bloco).
12	Escolher horário do recurso na grelha de resumo	Ao clicar sobre uma das colunas da grelha de resumo, o utilizador pode alterar o horário seleccionado naquele recurso.
13	Mover marcação de uma tarefa (simples)	O utilizador pode mover a marcação de uma tarefa mestre, para isso basta arrastar a célula marcada para o sítio pretendido. Antes de concretizar a operação, o sistema valida se a operação cumpre as restrições do protocolo.
14	Mover marcação de uma tarefa em bloco	Esta acção é similar ao mover simples, com a diferença de que neste caso não se move apenas a tarefa seleccionada, mas também todas as tarefas que estão agendadas para a frente.
15	Alterar a legenda das grelhas	As grelhas da tabela contêm uma legenda com a correspondência entre o dia e a célula. A informação do dia pode aparecer de duas formas, sobre a forma de dia do mês, ou de dia de um ciclo. O sistema tem de disponibilizar uma forma de comutar entre as duas vistas.
16	Alterar modo de visualização da grelha de recursos	A grelha de recursos tem de ter dois modos de visualização, um modo por tipo de recursos e outro por recursos. O sistema tem de fornecer uma opção onde se comuta entre as duas vistas.
17	Cancelar marcação do protocolo	O sistema dispõe de uma opção que permite cancelar a marcação que o utilizador fez. Depois de cancelar a marcação, o sistema mostra novamente o ecrã de listagem.
18	Voltar à listagem de protocolos	Ao seleccionar a opção de voltar, o sistema pergunta se pretende guardar as alterações, rejeitar as alterações ou cancelar. Nas duas

		primeiras opções, o sistema executa a acção correspondente e depois volta ao ecrã de listagem.
19	Guardar marcação do protocolo	Ao utilizar esta funcionalidade, o sistema guarda todas as marcações efectuadas no protocolo e depois volta ao ecrã da listagem.

4.2.3 Requisitos não funcionais

Os requisitos não funcionais identificados ao longo da fase de levantamento de requisitos são enumerados nos pontos seguintes:

- **Usabilidade** – a aplicação deve ser fácil de utilizar e intuitiva. A sucessão de passos necessária para executar uma acção tem de ser clara e única, dissipando ambiguidades. A interface da aplicação deve ser atractiva e o número de funcionalidades disponibilizadas deve ser o estritamente necessário, mas estas devem ser bem pensadas.
- **Disponibilidade** - o sistema deve ser capaz de fornecer uma resposta quando tal for requisitado. O médico não pode estar demasiado tempo à espera que o sistema funcione, pois o tempo de uma consulta normalmente é escasso.
- **Eficiência** – quando requisitado, o sistema deve responder prontamente. Se o seu tempo de resposta for alto o sistema torna-se inútil, porque os médicos não o vão poder usar durante as consultas (que é um dos principais objectivos, pelo menos do módulo de prescrição e agendamento).
- **Confiança** – a classe médica, por norma, é um pouco céptica em relação às tecnologias, por isso é necessário um grande esforço para conseguir a sua integração. Nesse contexto é preciso assegurar que existe um baixo índice de erros e que na eventualidade de ocorrer uma falha, as suas repercussões são ínfimas. Caso contrário, será difícil convencer os médicos da sua utilidade.
- **Robustez** – mesmo que eventualmente ocorram erros, o sistema deve camuflar e tentar minimizar o seu impacto. Caso contrário, os médicos podem não saber o que fazer e começam a depositar pouca confiança no sistema. Por outro lado, o sistema também deve ser construído de forma a filtrar o maior número de erros, limitando logo à partida as situações que possam conduzir a um estado instável.
- **Segurança** – dado que o sistema vai conter os tratamentos dos pacientes, é extremamente importante que a informação seja confidencial e inacessível para utilizadores sem permissões para tal. Caso a alteração de um protocolo de um paciente fosse elaborada por uma pessoa sem autorização os efeitos poderiam ser trágicos, podendo conduzir ao agravar do estado médico do paciente.
- **Flexibilidade** – durante o tempo de vida de um produto é pouco provável que este seja estático, podendo ser necessário alterar as suas funcionalidades ou acrescentar novas. Por esse motivo é desejável que o esforço necessário para produzir essas eventuais alterações seja mínimo, diminuindo assim o seu custo e tempo de desenvolvimento.
- **Resolução** – a aplicação deve ser optimizada para funcionar na resolução de 1440x900 , embora também deva ser garantido que funciona nas restantes.

4.3 Casos de uso

Neste subcapítulo são apresentados os casos de uso extraídos da lista de requisitos definida no subcapítulo anterior (4.2), bem como os actores que interagem com eles. O subcapítulo está dividido em três partes: na primeira parte são apresentados os diagramas de casos de uso de cada um dos módulos; na segunda parte são apresentados os actores; na terceira parte é descrito um caso de uso completo, a título de exemplo. Tanto na primeira como na segunda parte, os conteúdos estão divididos por módulos, facilitando assim o seu entendimento e leitura.

O objectivo deste subcapítulo é fornecer uma visão geral de todos os casos de uso do projecto e ilustrar o modo como interagem entre si.

4.3.1 Diagrama de casos de uso

Como já foi dito, a secção está dividida pela apresentação dos diferentes módulos do projecto, sendo em cada um deles apresentado o respectivo diagrama de casos de uso.

Módulo de Prescrição e Agendamento

A figura seguinte (Figura 4.2) apresenta o diagrama de casos de uso do módulo de prescrição e agendamento. Os casos de uso presentes na figura obedecem aos requisitos funcionais anteriormente definidos para o módulo de prescrição e agendamento.

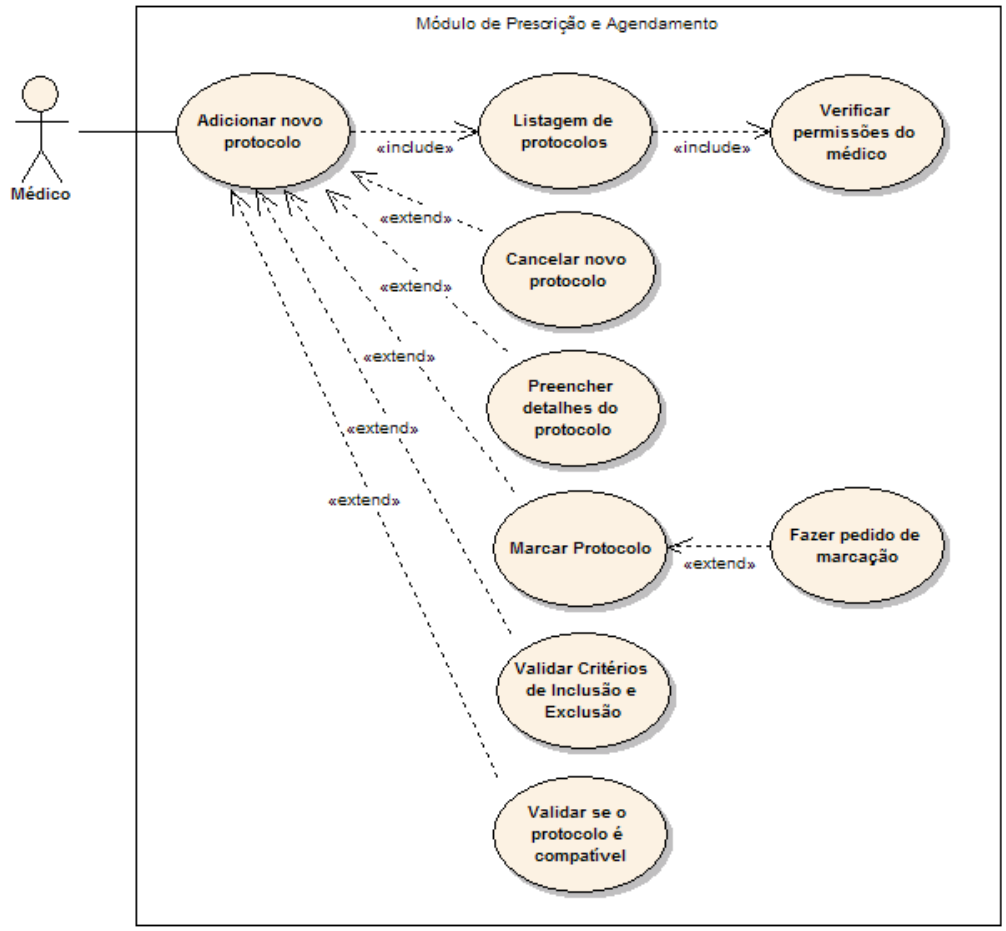

Figura 4.2 - Diagrama de casos de uso do módulo de prescrição e agendamento

Módulo de Marcação

A Figura 4.3 ilustra o diagrama de casos de uso do módulo de marcação. Como se pode constatar pela análise da figura, estes casos de uso obedecem aos requisitos funcionais anteriormente definidos para o módulo de marcação.

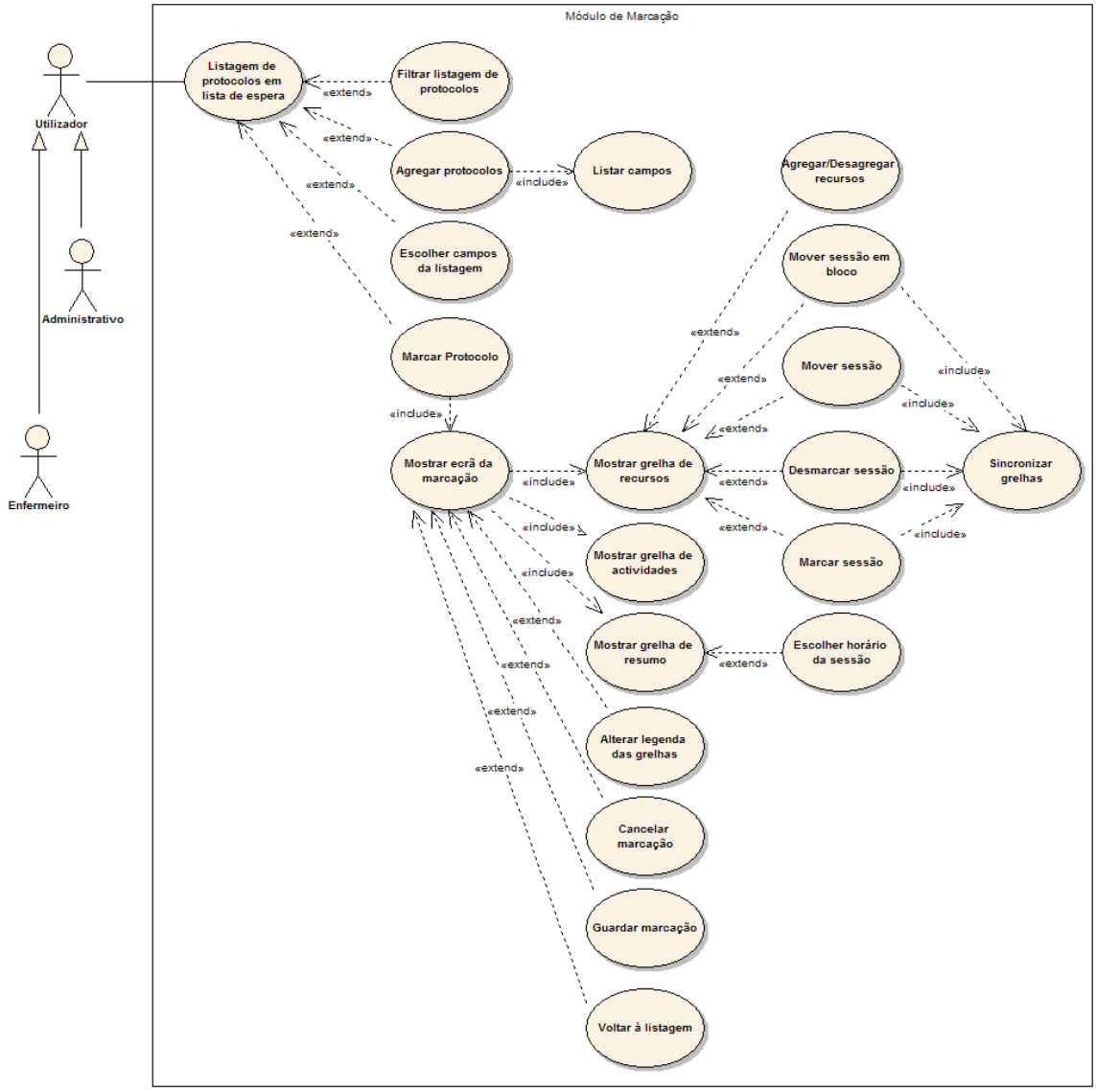

Figura 4.3 – Diagrama de casos de uso do módulo de marcação

4.3.2 Descrição dos actores

Nesta secção são descritos os actores que interagem com o sistema, sendo esta descrição dividida pelos dois módulos do projecto: módulo de prescrição e agendamento e módulo de marcação.

Módulo de Prescrição e Agendamento

O médico é o único interveniente neste módulo. O médico é caracterizado por ser uma entidade com elevado grau literário, embora na área das tecnologias de um modo geral se

constate possuir algumas reticências na sua utilização. Normalmente adopta um comportamento hesitante em relação a adopção de novas ferramentas tecnológicas e tenta delegar as suas funções para outras entidades. A cativação do médico é muito importante neste projecto, porque o sucesso do produto depende essencialmente dele.

O médico não necessita possuir grandes conhecimentos na área das tecnologias para saber utilizar este módulo. A única exigência é que esteja minimamente familiarizado com os conceitos básicos de um computador do ponto de vista do utilizador.

Módulo de Marcação

No módulo de marcação já existe uma maior diversidade de actores, nomeadamente três, embora os que realmente importam sejam só dois. O actor Utilizador apenas serve de elemento agregador, uma vez que tanto o Enfermeiro como o Administrativo têm permissões semelhantes sobre o sistema.

Os enfermeiros são conhecidos por aceitar bem a introdução de novos mecanismos que facilitem a sua rotina diária. Assim sendo, a tarefa de persuasão sobre esta classe não é necessária, porque logo à partida estes estão receptivos para a sua utilização. Os enfermeiros normalmente não têm grande dificuldade na utilização de sistemas informáticos, pois nos hospitais portugueses já há algum tempo que é prática comum a sua utilização.

A classe dos administrativos também não apresenta grandes dificuldades em utilizar sistemas informáticos, uma vez que a utilização de tecnologias já faz parte do seu quotidiano. É necessário no entanto ter alguma cautela na interacção com os administrativos, porque normalmente são um pouco reticentes em relação à introdução de novos sistemas. Essas reticências explicam-se pelo facto de terem medo de ser substituídos pelos novos sistemas, pois estes simplificam algumas das suas tarefas diárias.

Ao contrário do módulo anterior (módulo de prescrição e agendamento), a utilização do módulo de marcação já implica um maior à vontade com as tecnologias, já que os conceitos introduzidos na interface são um pouco mais complexos. Mesmo assim, considera-se que a adaptação a este módulo é fácil e estima-se que no final de uma semana os novos utilizadores consigam utilizá-lo de forma intuitiva.

4.3.3 Descrição detalhada de um caso de uso

O objectivo desta secção é apresentar a descrição detalhada de um caso de uso da aplicação, demonstrando assim qual a metodologia seguida pelo autor. O caso de uso escolhido como exemplo é o *Marcar Sessão* do módulo de marcação.

Descrição do caso de uso

A tabela seguinte (Tabela 4.3) contém uma descrição detalhada do caso de uso *Marcar Sessão*.

Tabela 4.3 – Descrição do caso de uso *Marcar Sessão*

Nome:	Marcar Sessão
Descrição:	Este caso de uso é utilizado para marcar uma nova sessão num protocolo. As sessões são marcadas através de uma grelha, como se

	pode verificar no protótipo fornecido (Figura 4.5). Todo o processo de marcação é descrito detalhadamente no diagrama de actividades (Figura 4.4).
Criado por:	José Carvalho
Data da Criação:	30 – 04 - 09
Actores:	Utilizador
Pré-Condições:	1 - Ainda existem sessões por marcar 2 - A célula ainda não está seleccionada 3 - Ainda não existe nenhuma Sessão marcada nesse dia
Pós-Condições:	1 - Caso as restrições sejam válidas, a célula deve estar marcada e as restantes grelhas também devem estar actualizadas. 2 - Caso as restrições não se verifiquem, então a aplicação não deve fazer nada e deve surgir uma mensagem de alerta
Fluxo Normal:	1 - Aceder ao módulo de marcação 2 - Seleccionar um dos protocolos pendentes (Marcar Protocolo) 3 - Marcar Sessão
Fluxos Alternativos:	1 - Aceder ao módulo de marcação 2 - Seleccionar um dos protocolos pendentes (Marcar Protocolo) 3 - Agregar/Desagregar Recursos 4 - Marcar Sessão
Excepções:	1 - Se o utilizador tentar marcar uma sessão e já existir uma nesse dia, o sistema mostra uma mensagem a informar. 2 - Se o utilizador tentar marcar uma sessão e já não houver vagas nesse recurso, o sistema deve mostrar uma mensagem informativa.
Inclui:	Sincronizar Grelhas
Prioridade:	Elevada
Frequência de Utilização:	Elevada
Regras de Negócio Cobertas:	Marcar uma sessão
Pressupostos:	O utilizador já deve estar com o login feito na aplicação do processo clínico.
Notas e Problemáticas:	Ao marcar uma sessão, a grelha de recursos fica assinalada com um visto

Diagrama de actividades:

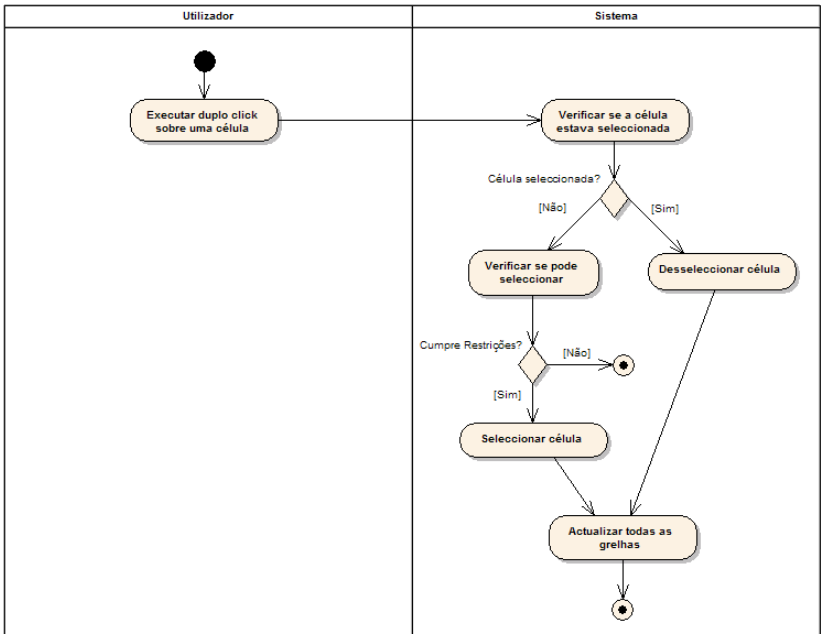

Figura 4.4 – Diagrama de actividades do caso de utilização

Protótipo da interface:

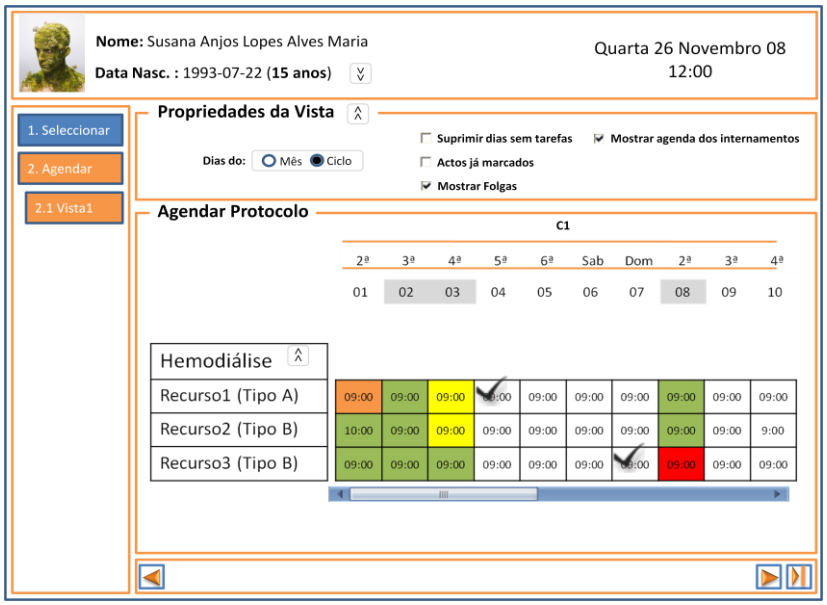

Figura 4.5 – Protótipo da interface de agendamento

4.4 Arquitectura do projecto

Com esta secção pretende-se evidenciar a arquitectura adoptada para o desenvolvimento da aplicação. Ao escolher a arquitectura da aplicação foram tidos em conta alguns factores fundamentais: expansibilidade; modularidade; capacidade de teste; desempenho e segurança. A arquitectura da aplicação é descrita a dois níveis: lógico e físico.

4.4.1 Arquitectura lógica

A arquitectura lógica refere-se à decomposição hierárquica do sistema em módulos lógicos e à especificação das interfaces e dependências entre módulos. A figura seguinte (Figura 4.6) ilustra a arquitectura adoptada.

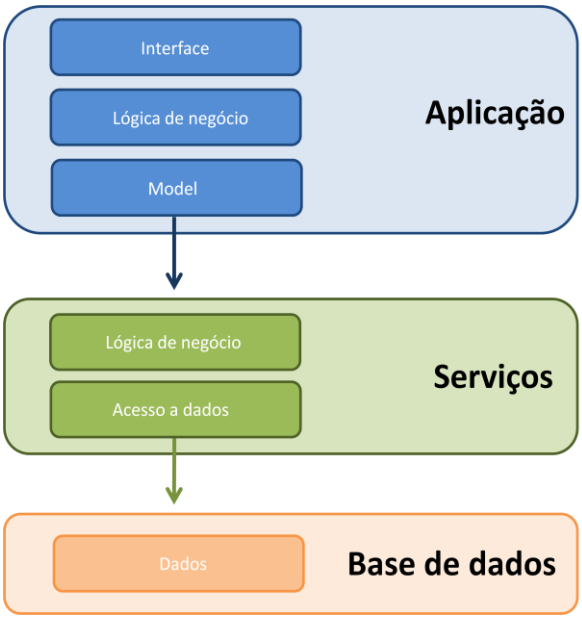

Figura 4.6 – Diagrama da arquitectura lógica da aplicação

A arquitectura adoptada consiste em três camadas: camada de Aplicação - modela as interfaces da aplicação, sendo que estas seguem o padrão MVP; camada de Serviços - inclui um conjunto de serviços que permitem estabelecer a ponte entre a interface e a base de dados, promovendo a modularidade e a extensibilidade; camada de Base de Dados (BD) - é responsável por armazenar todos os dados da aplicação.

Os serviços, para além de estabelecerem a ponte entre a interface e a BD, também possibilitam incluir alguma da lógica de negócio da aplicação. Ao incluir alguma da lógica de negócio na camada de serviços, promove-se a reutilização e retira-se responsabilidade à interface (o que é útil quer por questões de segurança quer de desempenho).

Para facilitar a compreensão de como é efectuado o acesso aos dados e identificar as interfaces utilizadas na comunicação entre as diversas camadas, é apresentado o diagrama seguinte (Figura 4.7).

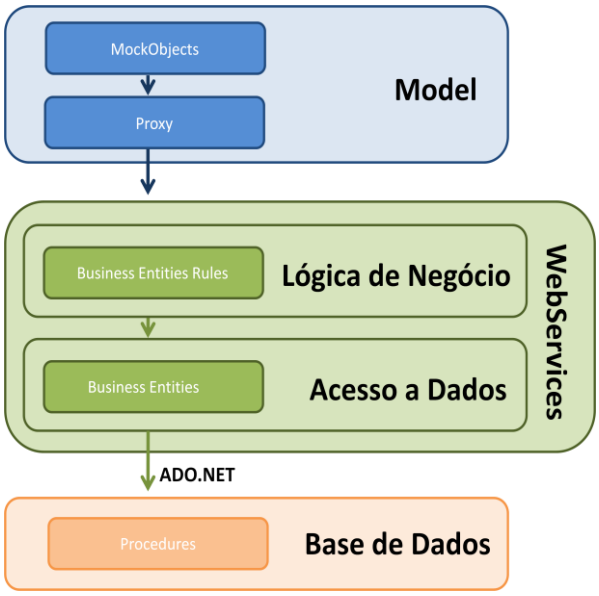

Figura 4.7 – Diagrama do acesso a dados

Na figura anterior (Figura 4.7) vê-se em detalhe o modo como foi implementada a lógica dos dados e como é efectuado o acesso a estes. Ao nível da interface (*Model*) foram criados *MockObjects*[3], que representam a informação e permitem abstrair toda a interface da sua verdadeira implementação. Para além dos *MockObjects*, a interface também contém uma entidade *Proxy*[4], que permite estabelecer a ponte com os serviços. No fundo o *Proxy* é a entidade que representa a interface dos serviços e é responsável por encontrá-los.

A camada de *WebServices*[5] (Serviços) contém uma representação de alto nível do modelo de dados presente na BD e permite efectuar todos os acessos/operações sobre a BD. As *Business Entities Rules* representam as funcionalidades implementadas ao nível dos serviços e utilizam as *Business Entities* para trabalhar sobre os dados. As *Business Entities* mapeiam algumas das classes existentes na BD e permitem efectuar todo acesso à base de dados, incluindo as operações básicas de CRUD (*Create, Read, Update, Delete*). As *Entities Rules* acedem à BD através da interface de ADO.NET[6].

Por último surge a camada de Base de Dados. Todas as operações efectuadas sobre a BD subentendem a criação de um *Procedure*[7], que implementa a acção necessária.

4.4.2 Arquitectura física

A arquitectura física refere-se à decomposição do sistema em módulos físicos e à especificação das interfaces e dependências entre os módulos. Na figura seguinte (Figura 4.8) pode-se constatar a arquitectura adoptada.

[3] Objectos que simulam o comportamento de objectos reais.
[4] Servidor que faz o mapeamento entre pedidos de clientes e os recursos de servidores.
[5] Sistema desenhado para suportar a interacção entre computadores através da rede.
[6] Conjunto de classes definidas pela *NET Framewok* que permite o acesso a dados numa BD.
[7] É uma rotina definida ao nível da BD.

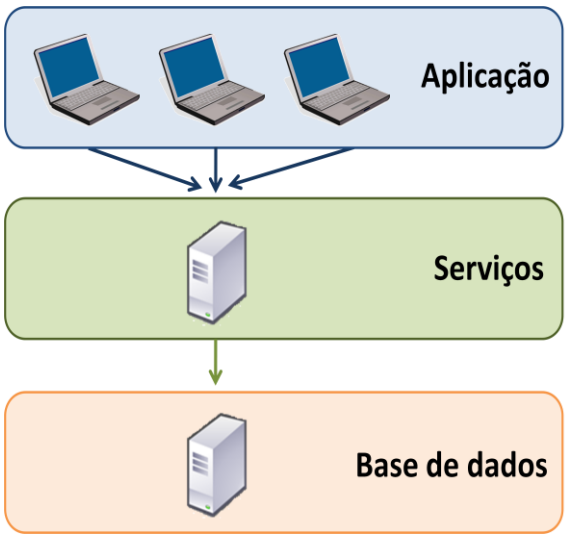

Figura 4.8 – Diagrama da arquitectura física da aplicação

A arquitectura física da aplicação foi projectada de forma a permitir que as três camadas identificadas na arquitectura lógica possam estar alojadas em sítios distintos. A vantagem de uma abordagem deste género é possibilitar o aumento de desempenho, segurança e disponibilidade, já que é possível duplicar as componentes em cada uma das camadas.

A interacção entre a aplicação e os serviços é feita através de uma rede local, da internet ou de GPRS (*General Packet Radio Service*). A comunicação entre os serviços e a BD pode ser estabelecida pelos mesmos meios identificados para a comunicação entre a aplicação e os serviços.

4.5 Integração com a solução da empresa

Neste subcapítulo pretende-se dar uma perspectiva de alto nível de como os módulos produzidos são integrados no contexto das soluções da empresa. Como já foi referido na introdução do relatório (1.1), os módulos produzidos para o agendamento foram integrados no módulo já existente denominado processo clínico. A figura seguinte (Figura 4.9) ilustra como os módulos produzidos no âmbito deste projecto (representados na figura por Agendamento) se integram na solução existente.

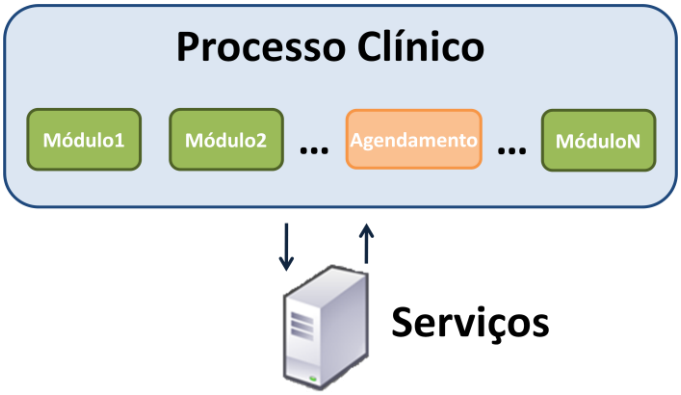

Figura 4.9 – Módulo de agendamento no contexto do processo clínico

Tendo em conta a arquitectura utilizada para toda a solução (a Infra-estrutura compilada para CAB), os módulos agora produzidos são facilmente integrados nesta. Para integrar o módulo na solução existente basta apenas carrega-lo na Shell do processo clínico e depois injectar as suas interfaces na janela principal. Os novos módulos utilizam os mesmos recursos dos já existentes, nomeadamente, o servidor de serviços.

É através do servidor de serviços que os módulos que constituem o processo clínico conseguem aceder à informação presente na base de dados, tal como foi descrito no subcapítulo da arquitectura (4.4). No diagrama anterior (Figura 4.9) o servidor de serviços está representado como apenas um elemento, mas na realidade as suas funções podem ser desempenhadas por várias unidades.

Capítulo 5

Implementação

Neste é capítulo é descrito o estado actual da aplicação. Não será feita uma análise muito detalhada em relação ao produto, na medida em que apenas se pretende dar uma perspectiva geral das funcionalidades que foram concluídas com sucesso. Para uma descrição mais detalhada do produto, o leitor pode consultar os anexos A e B que apresentam uma descrição exaustiva das suas funcionalidades e capacidades.

5.1 Estado actual da aplicação

Esta secção encontra-se dividida em duas partes, correspondentes aos dois módulos que compõem a aplicação: módulo de prescrição e agendamento e o módulo de marcação. Na descrição actual de cada módulo são abordados três temas: a interface, a ligação com a base de dados e a lógica de negócio.

O projecto até à fase actual incidiu principalmente na definição das interfaces dos dois módulos. No subcapítulo 4.2 já foi apresentada uma das principais razões para o desenvolvimento se ter centrado nas interfaces; dificuldade na definição de um conjunto de requisitos estáveis que descrevessem a aplicação pretendida. Foi necessário gastar algum tempo na análise e consolidação dos requisitos, para que na fase de implementação os imprevistos fossem minimizados. Uma alteração na fase de análise de requisitos é muito menos custosa e problemática do que na fase de implementação.

Após a fase de definição de requisitos, em que foram acordadas as funcionalidades da aplicação, seguiu-se a fase de definição das interfaces. Nesta fase foi depositado bastante esforço na construção de uma interface que apresentasse um grau de usabilidade elevado. Mais uma vez a proximidade com o cliente foi preponderante, facilitando o processo de ajuste e validação das ideias ponderadas. O canal de comunicação utilizado para chegar ao cliente foi a prototipagem, através dela o cliente consegue perceber com maior clareza o que é proposto e até no período final experimentar (protótipos evolutivos).

Uma vez definida a disposição dos componentes e a forma como o utilizador deve interagir com eles, passou-se à fase de definição dos estilos da interface. Esse processo resultou de uma parceria com a *designer* da empresa, que ajudou a definir o *layout* final da aplicação.

5.1.1 Módulo de Prescrição e Agendamento

O módulo de prescrição e agendamento recebeu uma menor ênfase no decorrer do projecto, face ao módulo de marcação. Todo o trabalho elaborado neste módulo foi no âmbito da construção das suas interfaces. Ao contrário do módulo seguinte (5.1.2), neste caso ainda não foram definidos os *MockObjects* que emulam os dados ao nível local (das interfaces).

As interfaces estão quase completamente concluídas, faltando apenas corrigir alguns detalhes, na tentativa de facilitar a interacção com o utilizador.

Ao nível da lógica de negócio não foram feitos grandes avanços, falta implementar toda a lógica que permite verificar se a marcação de um protocolo é valida. Um protocolo pode ser prescrito a um paciente caso não exista nenhum conflito com os seus protocolos activos, ou não exista nenhuma contra indicação (estado médico, alergias, etc).

A figura seguinte (Figura 5.1) mostra a interface inicial do módulo de prescrição e agendamento.

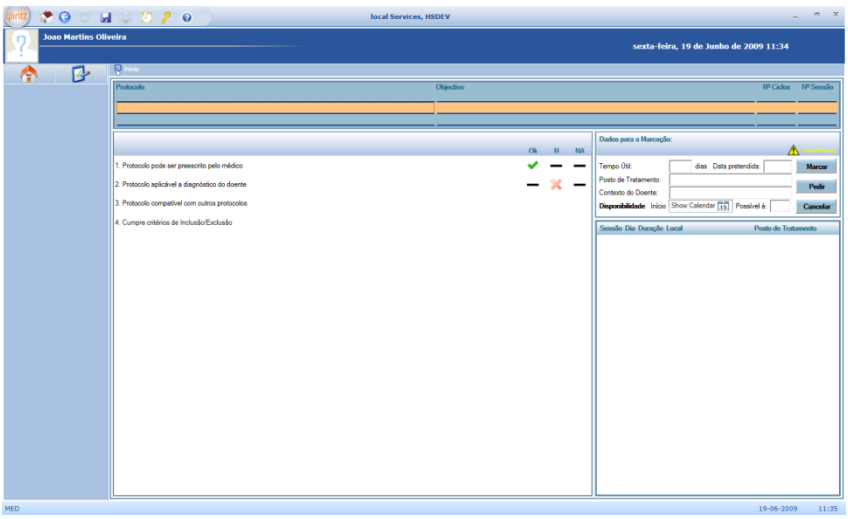

Figura 5.1 – Ecrã inicial do módulo de prescrição e agendamento

O ecrã inicial deste módulo é a base de toda a operação de prescrição de protocolos. Neste ecrã subentende-se que já se está na área de um paciente e pretende-se adicionar um novo protocolo. Ao adicionar um novo protocolo, ele é acrescentado à lista de protocolos do paciente, que aparece na parte superior da aplicação. Antes de poder efectuar a marcação do novo protocolo é necessário validar um conjunto de restrições. Do lado esquerdo do ecrã existe uma lista de restrições, que necessitam de ser validadas antes de ser possível marcar o protocolo. Uma vez validado o protocolo, é necessário preencher um conjunto de características que indicam o contexto da prescrição do protocolo e identificam a data em que o médico pretende marcar o protocolo.

Na parte inferior direita da interface é apresentada uma listagem com todas as actividades que compõem o protocolo.

Para além do ecrã inicial também foi elaborado o ecrã de confirmação dos critérios de inclusão e exclusão do protocolo. Na figura seguinte (Figura 5.2) é possível visualizar esta interface.

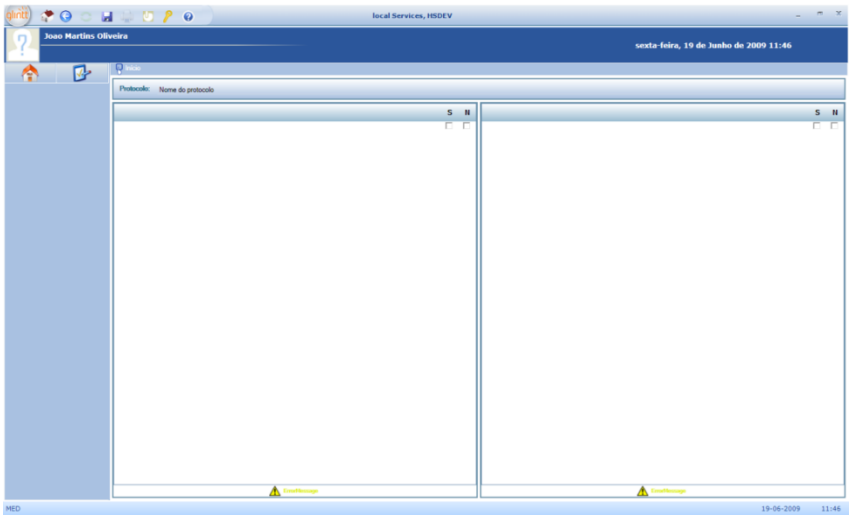

Figura 5.2 – Ecrã de confirmação dos critérios de inclusão e exclusão do protocolo

No ecrã anterior (Figura 5.2) o médico deve confirmar se o doente está realmente capaz de cumprir o protocolo proposto. A validação do protocolo é feita através da verificação dos critérios de inclusão e exclusão do mesmo no contexto do doente. Na prática isso é feito na interface através da selecção da *checkbox* correspondente ao Sim (S) em todos os critérios de inclusão e ao inverso nos critérios de exclusão.

5.1.2 Módulo de Marcação

O módulo de marcação é o estandarte do projecto, foi nele que incidiu grande parte do esforço. As interfaces do módulo estão completamente prontas, faltando apenas a ligação com a BD e a validação de algumas restrições na interacção com a grelha de marcação.

Embora a ligação com a BD ainda não esteja pronta, foram criados objectos locais (*MockObjects*) que simulam o seu funcionamento, podendo assim basear todo o acesso e manipulação de dados através deles. O que falta neste momento é criar um conjunto de *WebServices* que acedam à BD e a liguem aos MockObjects, permitindo assim efectuar as operações necessárias (CRUD). Os *Procedures* na BD não são da responsabilidade do autor, a empresa delegou essa tarefa a outros elementos dos seus quadros. A delegação dessa tarefa surge porque se pretende importar alguma da lógica de negócio utilizada em outras soluções da empresa, como por exemplo, a parte de agendamento deve usar os calendários utilizados na área de medicação.

Uma vez apresentado o estado actual do módulo de marcação, falta apresentar as interfaces finais do produto. De seguida são apresentados os dois ecrãs disponibilizados pelo módulo. A figura seguinte (Figura 5.3) apresenta o ecrã inicial deste módulo.

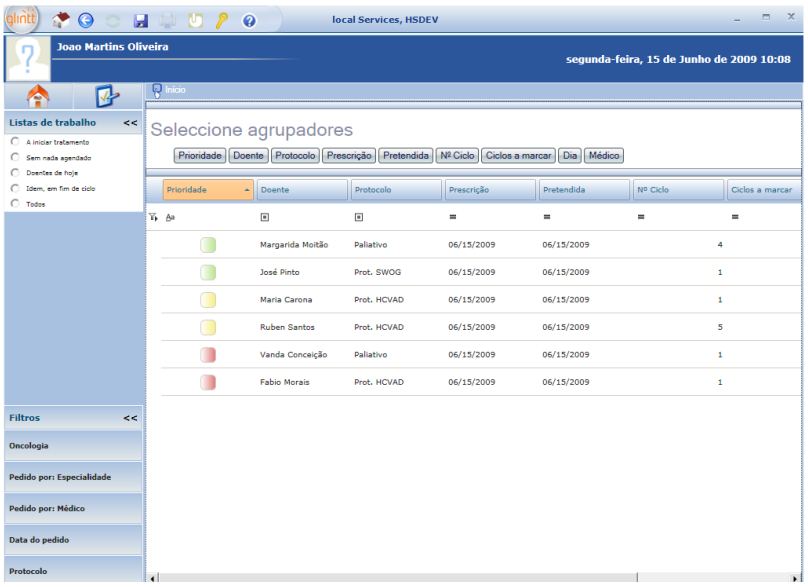

Figura 5.3 – Ecrã inicial do módulo de marcação

O ecrã inicial do módulo de marcação serve apenas para seleccionar os protocolos que se pretende agendar. Uma vez escolhido o protocolo que se pretende agendar, o utilizador é encaminhado para o ecrã de marcação, que é ilustrado na figura seguinte (Figura 5.4).

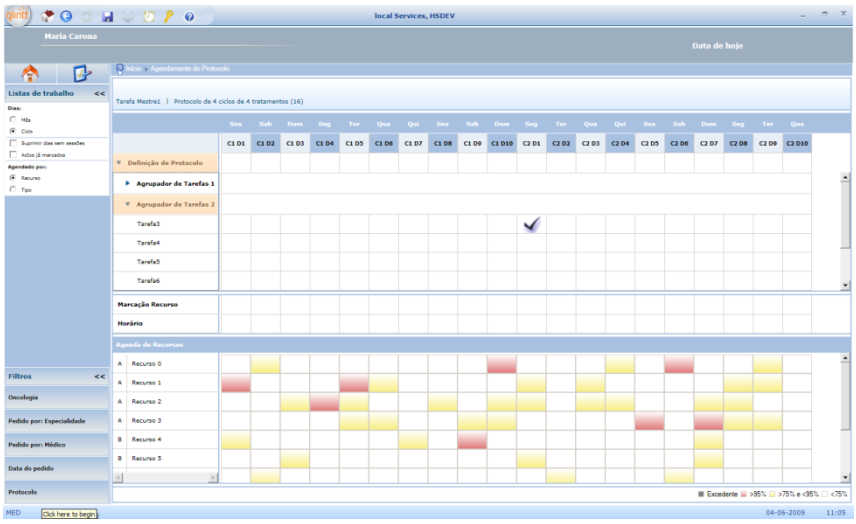

Figura 5.4 – Ecrã de marcação do protocolo

Como já foi realçado diversas vezes ao longo do documento, o ecrã de marcação do protocolo é a pérola da aplicação, permitindo agendar de uma forma fácil e intuitiva as sessões de um protocolo. Toda a informação que é ilustrada na interface foi pensada ao pormenor, tentando reproduzir um ambiente de trabalho agradável, intuitivo, claro e seguro para os utilizadores.

Capítulo 6

Conclusões e Trabalho Futuro

Por último, mas não menos importante, chega a hora da reflexão em que é necessário meditar sobre todo o trabalho elaborado no contexto do projecto. Pretende-se neste capítulo verificar a satisfação dos objectivos propostos inicialmente e ponderar em relação ao trabalho futuro.

6.1 Satisfação dos Objectivos

Para começar é importante relembrar quais foram os objectivos delineados no início do projecto. Um primeiro objectivo era encontrar uma solução para o problema de agendamento de protocolos nos hospitais, definindo um mecanismo que permitisse agilizar e apoiar a sua concretização. Um segundo objectivo foi o respeito por duas características do software consideradas fundamentais para o sucesso da aplicação: flexibilidade (fácil manutenção e expansibilidade) e usabilidade.

No que a toca à definição do mecanismo de agendamento, esta tarefa fica marcada pela colocação em segundo plano do algoritmo de agendamento automático. As investigações desse algoritmo ficaram-se pela análise inicial apresentada na revisão bibliográfica (2.1.2). Este facto deveu-se à alteração das prioridades dos requisitos do projecto e consequente a alteração dos seus objectivos, tal como é explicado na secção 4.2.1. Assim sendo, a culpa de ele não se ter realizado não pertence ao autor do documento. O restante mecanismo de agendamento foi concretizada com sucesso, as negociações com o cliente (interno à empresa) chegaram a bom porto e foi possível validar o conjunto de requisitos que o regulam. Nem sempre as negociações foram fáceis, mas foi possível testar e colocar em prática algumas das competências adquiridas durante o período académico. Algumas vezes as competências adquiridas mostraram-se insuficientes para lidar com as situações, tendo sido necessário recorrer a elementos da empresa mais experientes que ajudaram a encontrar o caminho certo. É de salientar, que todo este processo foi enriquecedor e contribuiu para o amadurecimento e aperfeiçoamento das competências do autor.

No âmbito das características do software, também se considera que os objectivos foram totalmente atingidos. A questão da flexibilidade do software foi convenientemente tratada, tendo resultado na construção da Infra-estrutura de migração de CAB para PRISM. A Infra-estrutura foi fundamental no contexto do projecto, na medida em que permitiu a construção de uma aplicação que pode ser integrada no imediato com as soluções da empresa (em CAB) e ao mesmo tempo já está preparada, e tira o máximo partido, da nova *Framework* que a empresa pretende adoptar (PRISM). Tanto a *Framework* CAB como PRISM assentam em ideias que privilegiam a modularidade e a flexibilidade, por isso, a única questão que foi necessária endereçar foi a organização das interfaces. A organização das interfaces foi solucionada através da adopção do padrão MVP.

A questão da usabilidade também foi cuidadosamente tratada, estando à vista os seus resultados nas interfaces dos módulos produzidos. Uma análise cuidada dessas interfaces mostra que foram cuidadosamente pensadas, no intuito de fornecer ao utilizador uma experiência rica e intuitiva.

O único ponto parcialmente cumprido foi a implementação completa do projecto, mas tendo em conta o tempo disponível e as contrariedades que foram surgindo, considera-se que os resultados não ficaram aquém do esperado.

6.2 Trabalho Futuro

A conclusão da implementação do projecto é o ponto principal a apontar como trabalho futuro.

Um outro passo importante é pensar na questão do algoritmo de agendamento, que foi colocado em segundo plano neste trabalho e não chegou a ser desenvolvido. Existem duas abordagens possíveis em relação a esta questão. Por um lado pode-se tentar melhorar a solução existente, trabalhando a questão do desempenho e incluindo a parte de optimização das soluções (aumento da satisfação dos clientes, ou optimização na gestão dos recursos). A outra abordagem possível é construir um algoritmo de raiz, considerando o autor que esta abordagem é mais aliciante e motivadora, contudo, é uma solução mais demorada e o resultado final não está assegurado.

O trabalho futuro, também pode passar pelo acrescentar de novas funcionalidades à solução existente, ou alterar alguns dos pressupostos utilizados. Uma abordagem interessante seria o alargamento do agendamento das tarefas para todo o conteúdo do protocolo e não só para a tarefa mestre. Este caminho aumenta a autonomia dos utilizadores da aplicação, permitindo-lhes, caso pretendam, ter o controlo da marcação de todas as tarefas de uma sessão e não só da tarefa mestre.

Uma outra funcionalidade interessante seria a inclusão, no módulo de marcação, de algoritmos que ajudassem os utilizadores na tomada de decisões, como por exemplo, um mecanismo de geração de hipóteses de agendamento. Esse mecanismo poderia sugerir possíveis soluções de agendamento, que se preocupam com a perspectiva do paciente e tentassem melhorar a sua qualidade de vida.

Um caminho completamente distinto, mas igualmente interessante, seria construir uma funcionalidade que permitisse a marcação de vários protocolos em simultâneo, tentando optimizar a solução geral. Esta vertente poderia alcançar soluções que satisfizessem melhor a globalidade das necessidades dos pacientes, conseguindo assim aumentar o seu índice de satisfação.

Bibliografia

[AD06] A. Aguiar and G. David, "Patterns for documenting frameworks: customization," in *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*. Portland, Oregon: ACM, 2006, pp. 1-10.

[Adl06] M. Adler. (2006, Nov.) MAGMASYSTEMS BLOG. [Online]. http://magmasystems.blogspot.com/2006_11_01_archive.html

[AL03] e. H. L. Aarts and J. K. Lenstra, *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[Ale79] C. Alexender, *Timeless Way of Building*. The Oxford University Press, 1979.

[ALMV07] C. Andrade, S. Livermore, M. Meyers, and S. V. Vliet, *Professional WPF Programming: .NET Development with the Windows® Presentation Foundation*. Wiley Publishing, 2007.

[Atk97] S. Atkinson, "Cognitive Deficiencies in Software Library Design," 1997.

[Ber07] J. Berchte, "Evaluation of Plug-In Architectures for a Test," Setembro 2007.

[Blu02] C. Blum, "Metaheuristics for Group Shop Scheduling," in *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, 2002, pp. 631-640.

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal, *Pattern-Oriented Software Arquitecture*. Willey, 1996.

[Boo06] J.-P. Boodhoo. (2006) Design Patterns: Model View Presenter. [Online]. http://msdn.microsoft.com/en-us/magazine/cc188690.aspx

[Bru05] S. Brunning. (2005,) Library vs Framework. [Online]. http://mail.python.org/pipermail/python-list/2005-August/336257.html

[CF08] T. C. Chiang and L. C. Fu, "A rule-centric memetic algorithm to minimize yhe number of tardy jobs in the job shop," *International Journal of Production research*, 2008.

[Cop92] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[CZ01] E. K. P. Chong and S. H. Zak, *An Introduction to Optimization*. Wiley, 2001.

[Daw90] R. Dawkins, *The Selfish Gene*. USA: Oxford University Press, 1990.

[Dei96] J. Deighton, *The Future of Interactive Marketing*. Harvard Business Review, 1996.

[DHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *ECOOP (Object-Oriented Programming, Proceedings of 7th European Conference)*, 1993.

[DS04A] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.

[Emb06] T. Embassy. (2006,) Library vs. Framework. [Online]. http://techembassy.blogspot.com/2006/05/library-vs-framework.html

[FB91] E. Falkenauer and S. Bouffouix, "A Genetic Algorithm for Job Shop," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, Sacramento, California, 1991, pp. 824-829.

[FHLS97] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson, *Hooking into Object-Oriented Application Frameworks*. Boston, Massachusetts, United States: ACM, 1997.

[Fow96] M. Fowler, *Analysis Patterns - Reusable Object Models*. Addison-Wesley Professional, 1996.

[GHJV97] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, 1997.

[Hel06] A. Hellesøy. (2006, Jan.) Inversion of Control. [Online]. http://docs.codehaus.org/display/PICO/Inversion+of+Control

[HHKPVV01] M. Hakala, et al., "Annotating Reusable Software Architectures with Specialization Patterns," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, Amsterdam, Netherlands, 2001, pp. 171-180.

[Inf09] Infragistics. Wpf Controls. [Online]. http://www.infragistics.com/dotnet/netadvantage/wpf.aspx#Overview

[ISC97] S. Ishikawa, M. Silverstein, and A. Christopher, *A Pattern Language*. Oxford University Press, 1997.

[ISO98] ISO, "Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11 : Guidance on usability," Patent 9242-11, Mar. 15, 1998.

[JF88] R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-*

Oriented Programming, pp. 22-35, Julho 1988.

[Joh92] R. E. Johnson, "Documenting frameworks using patterns," in *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 1992, pp. 63-76.

[KG02] N. Krasnogor and S. Gustafson, "Toward Truly "memetic" Memetic Algorithms: discussion and proofs of concept," School of Chemistry and School of Computer Science & IT, 2002.

[KGR95] K. Knshna, K. Ganeshan, and D. J. Ram, "Distributed Simulated Annealing Algorithms for Job Shop Scheduling," *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, vol. 25, pp. 1102-1109, Jul. 1995.

[KP88] G. E. Krasner and S. T. Pope, "A Description of the Model-View-Controller User Interface Paradigm is the Smalltalk-80 System," 1988.

[LTMS07] S. Lopes, A. Tavares, J. Monteiro, and C. Silva, "Design and Description of a Classification System Framework for Easier Reuse," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, University of Minho, Guimarães, Portugal, 2007, pp. 71-82.

[Mac08] M. MacDonald, *Pro WPF in C# 2008, Windows Presentation Foundation with .NET 3.5*. Apress, 2008.

[Mal98] N. Malovic. (1998) VusCode - Coding dreams since 1998. [Online]. http://blog.vuscode.com/malovicn/archive/2007/10/25/model-view-presenter-mvp-design-pattern-close-look-part-1-passive-view.aspx

[Mcr08] Mcrosoft. (2008,) Model-View-Presenter Pattern. [Online]. http://msdn.microsoft.com/en-us/library/cc304760.aspx

[MF04] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Springer, 2004.

[Mic05] Microsoft. (2005,) Smart Client - Composite UI Application Block. [Online]. http://msdn.microsoft.com/en-us/library/aa480450.aspx

[Mic09a] Microsoft. (2009,) Inversion of Control. [Online]. http://msdn.microsoft.com/en-us/library/dd458907.aspx

[Mic09b] Microsoft. (2009,) Composite Application Guidance for WPF and Silverlight. [Online]. http://msdn.microsoft.com/en-us/library/dd458861.aspx

[Mic09c] Microsoft. About GDI+. [Online]. http://msdn.microsoft.com/en-us/library/ms533798(VS.85).aspx

[Mic09d] Microsoft. Windows Forms. [Online]. http://msdn.microsoft.com/en-us/library/dd30h2yb(VS.80).aspx

[Mil05] J. D. Miller. (2005, Jun.) The Dependency Injection Pattern – What is it and why do I care?. [Online]. http://codebetter.com/blogs/jeremy.miller/archive/2005/10/06/132825.aspx

[Mos89] P. Moscato, "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms," California Institute of Technology, 1989.

[MZ99] C. Marques and S. v. d. Zwaan, "Ant Colony Optimisation for Job Shop Scheduling," Instituto de Sistemas e Robótica, Instituto Superior Técnico, 1999.

[Nat06] A. Nathan, *Windows Presentation Foundation Unleashed*. Sams, 2006.

[New09] R. Newman. Introduction To CAB/SCSF. [Online]. http://richnewman.wordpress.com/intro-to-cab-toc/

[OL96] I. H. Osman and G. Laporte, "Metaheuristics: A bibliography," *Annals of Operations Research*, vol. 63, pp. 513-623, 1996.

[Pin08] M. L. Pinedo, *Scheduling - Theory, Algorithms, and Systems*, Third Edition ed. USA: Springer, 2008.

[Pla07] D. S. Platt, *Programming Microsoft® Composite UI Application Block and Smart Client Software Factory*. Microsoft Press, Julho 25, 2007.

[Pot96] M. Potel, "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java," 1996.

[PXH04] Z. Pin, L. Xiao-ping, and Z. Hong-fang, "An Ant Colony Algorithm for Job Shop Scheduling Problem," in *Proceedings of the 5'"World Congress on Intelligent Control and Automation*, China, 2004.

[Raz07] S. Razzaq, *A Framework for Building Smart Clients*. Carlsbad, California, USA, 2007.

[Raz07a] S. Razzaq, *A Framework For Building Smart Clientes - Microsoft Composite Application Block*. California, USA, Março 2007.

[Sav99] S. Savitha, "Design Patterns in Object-Oriented Frameworks," *Computer*, pp. 24-32, 1999.

[SCKR99] H.-J. Shin, I.-W. Choi, S.-D. Kim, and S.-Y. Rhew, "A Design of Object-Oriented Framework Repository," in *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on, vol3*, San Diego, CA, USASeoul, Korea, 1998, pp. 2686-2691vol3.

[Scu08] E. Sculli's. (2008,) Introduction to Applications supported by Composite UI Application Block (CAB). [Online]. http://blogs.southworks.net/esculli/2008/08/26/introduction-to-applications-supported-by-composite-ui-application-block-cab/

[Som04] I. Sommerville, *Software Engineering*, 7ªçãth ed. Addison-Wesley, 2004.

[Whi05] E. White, *Pro .NET 2.0 Graphics Programming, Building Custom Controls using GDI+*. Apress, 2005.

[Xha08] F. Xhafa, *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, A. Abraham and F. Xhafa, Eds. Springer, 2008.

[YZFGW08] T. Yu, J. Zhou, J. Fang, Y. Gong, and W. Wang, "Dynamic Scheduling of Flexible Job Shop Based on Genetic Algorithm," in *International Conference on Automation and Logistics*, Qingdao, China, 2008.

[ZSG08] G. Zhang, Y. Shi, and L. Gao, "A Genetic Algorithm and Tabu Search for Solving Flexible Job Shop Schedules," in *International Symposium on Computational Intelligence and Design*, China, 2008.

Anexo A

Descrição do Módulo de Marcação

A.1 Resumo

Neste anexo é apresentado em primeiro lugar o contexto onde está inserida a funcionalidade de marcação. O passo seguinte é documentar a funcionalidade de marcação, apresentando cada uma das suas interfaces e o seu ciclo de funcionamento.

A.2 Apresentação do Contexto

A funcionalidade de agendamento de protocolos foi integrada no panorama do processo clínico, tendo sido necessário adapta-la ao seu conceito. Neste capítulo é apresentando o ambiente envolvente da interface de agendamento, expondo às áreas funcionais que são herdadas do processo clínico.

A figura seguinte (Figura A.2.1) é uma ilustração do ecrã inicial do agendamento, é possível através dela apresentar as diversas áreas funcionais que são herdadas do processo clínico.

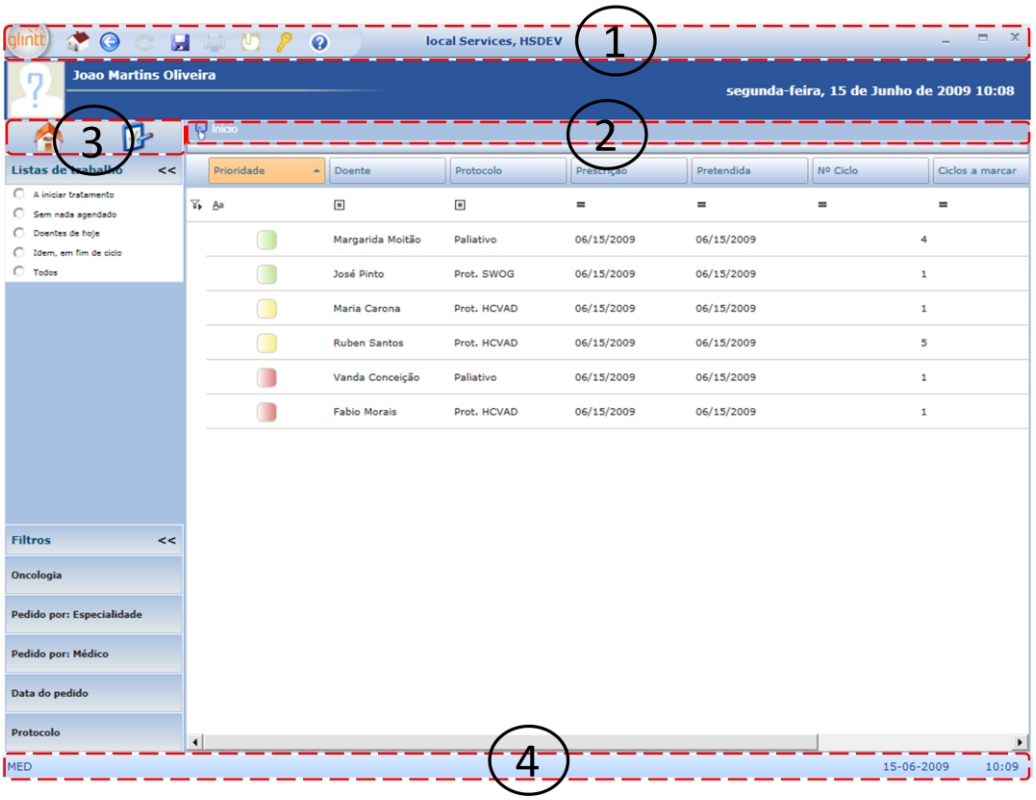

Figura A.2.1 - Zonas envolventes do agendamento

Na figura anterior (Figura A.2.1) estão identificadas as zonas funcionais herdadas do processo clínico. Cada uma das zonas está envolvida por uma linha a tracejado e está numerada.

- A zona um é uma barra de tarefas que fornece algumas funcionalidades disponibilizadas pela aplicação principal, tais como, opções de navegação, guardar informação, ajuda e voltar ao ecrã principal. Para além disso também facilita o acesso a um menu geral onde estão listadas as principais funcionalidades do processo clínico.
- A zona dois é um *BreadCromp*, através dele é possível verificar o caminho percorrido até à opção actual e caso seja necessário pode-se navegar para as opções anteriores.
- A zona três tem dois botões, um para fechar o processo clínico e o outro para voltar ao início do processo clínico.
- A zona quatro é uma barra de estado, em que aparece o nome do utilizador, a data actual e a hora.

A.3 Apresentação dos Ecrãs do Agendamento

Agora que as áreas funcionalidades disponibilizados pelo processo clínico estão devidamente comentados, é altura de começar a explicar as restantes áreas funcionais dos ecrãs que compõem o processo de agendamento.

O processo de agendamento é composto por dois ecrãs. Um ecrã inicial, onde é possível aceder a todos os protocolos em lista de espera para marcação, e um ecrã auxiliar onde é

elaborada a marcação de um protocolo. De seguida serão apresentados os dois ecrãs e explicados os componentes que os compõem.

A.3.1 Ecrã inicial do Agendamento

Na figura seguinte (Figura A.3.1) pode-se ver as zonas que compõem o ecrã inicial do agendamento.

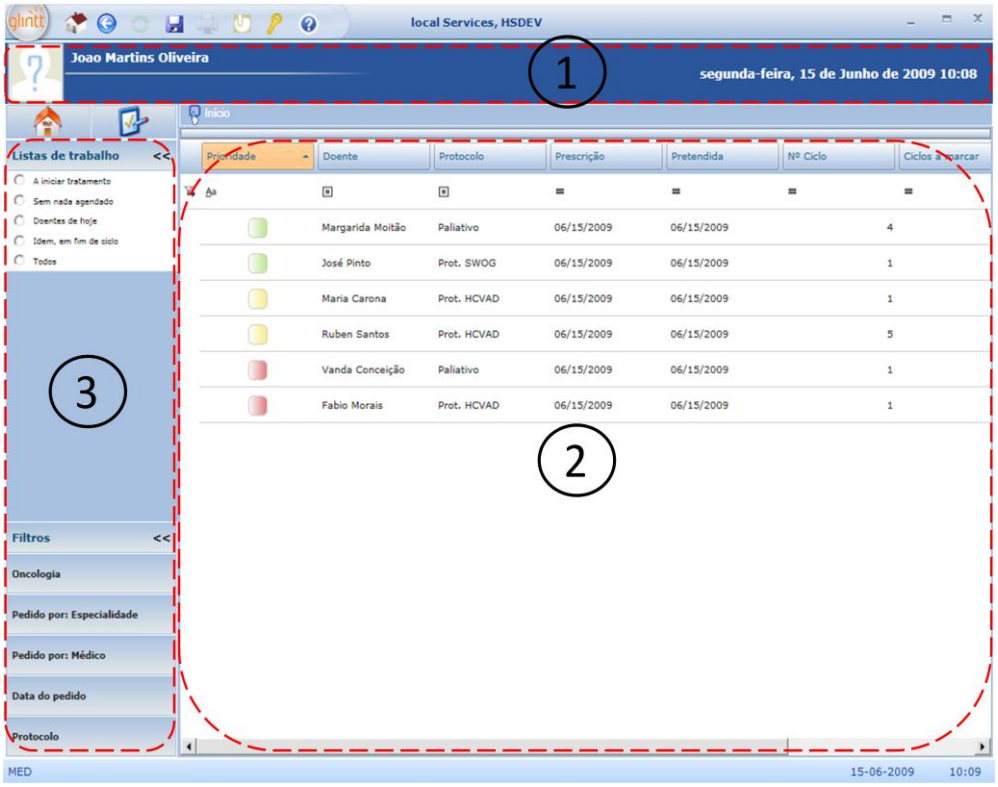

Figura A.3.1 - Zonas do ecrã inicial

De seguida será feita uma breve descrição das zonas numeradas no ecrã inicial (Figura A.3.1).

- A zona um é um cabeçalho da aplicação e identifica o utilizador que está a interagir com o sistema. Para além da informação do utilizador também aparece uma referência para a data e dia actual.
- A zona dois é onde aparece a listagem dos protocolos em lista de espera para marcação.
- A zona três tem duas funcionalidades distintas: por um lado contém uma lista de filtros que podem ser utilizados para condicionar a informação que é listada na zona dois; por outro lado serve de menu e disponibiliza alguns atalhos para aceder a outras funcionalidades da aplicação.

A.3.1.1 Zona de Listagem de Protocolos

Na construção da zona de listagem de protocolos foi utilizado um controlo da *Infragistics*[8] que permitiu facilitar a tarefa de implementação. O controlo da *Infragistics* utilizado é denominado por *DataGrid* e permite mostrar um conjunto de informação em forma de tabela. Para além das funcionalidades usuais deste tipo de controlos, a *DataGrid* da *Infragistics* contém um conjunto de opções que permitem melhorar a experiência do utilizador. As opções mais relevantes para o caso em questão são: agrupar a informação da tabela por campos; filtros para condicionar a informação visível e um conjunto de detalhes visuais que enriquecem a interface.

Na figura seguinte (Figura A.3.2) é possível ver em detalhe a aparência final da zona de listagem de protocolos.

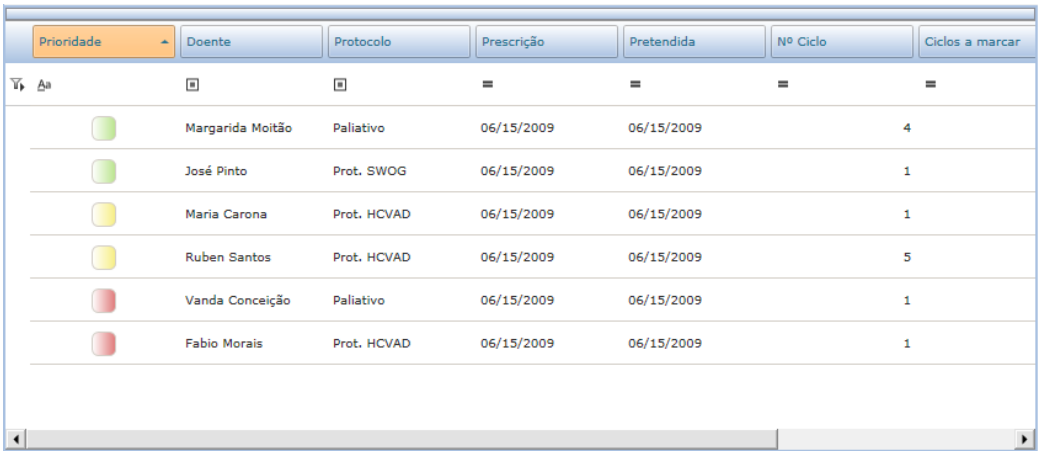

Figura A.3.2 - Componente que permite a listagem dos protocolos

A zona de listagem de protocolos é composta simplesmente pelo componente da *Infragistics* descrito anteriormente. Esse componente está dividido em diversas áreas funcionais. Na figura seguinte (Figura A.3.3) pode-se ver claramente as diversas áreas.

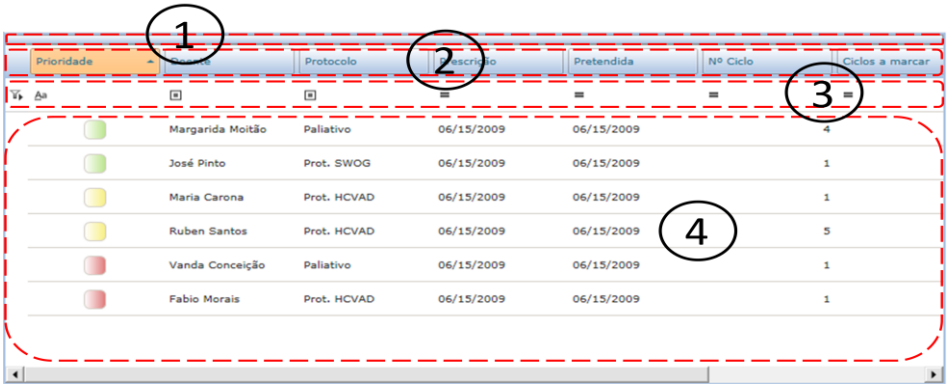

Figura A.3.3 - Áreas funcionais de uma DataGrid

[8] A Infragistics é uma empresa que desenvolve componentes para interfaces

A área um é a zona de agrupamento. Na figura anterior (Figura A.3.3) essa zona está escondida, mas caso se pressione o botão que se encontra no seu local, então torna-se possível aceder às suas funcionalidades.

A área dois é simplesmente o cabeçalho da tabela. O cabeçalho é composto pela identificação de cada uma das colunas da tabela e é possível através dela escolher o campo que ordena o conteúdo da tabela.

A área três é a zona de filtros. Os filtros estão divididos pelas diversas colunas da tabela, sendo possível filtrar pelos campos que se pretender.

A área quatro é simplesmente o local onde são mostrados os conteúdos da tabela. Através de um duplo *click* numa das linhas da tabela é possível a aceder a zona de marcação desse protocolo.

Detalhes da zona um

Para facilitar a compreensão do conceito de agrupamento e permitir verificar o comportamento do componente perante essa acção, serão apresentados algumas imagens ilustrativas.

Para começar pode ver-se na figura seguinte (Figura A.3.4) a área de agrupamento que é fornecida.

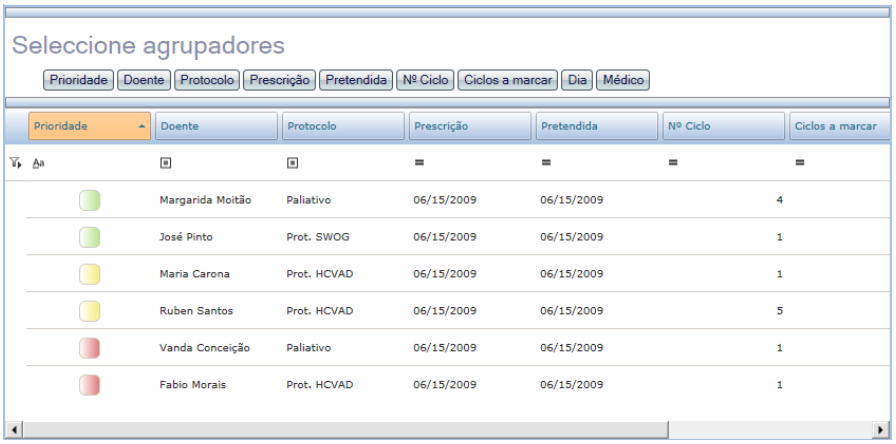

Figura A.3.4 - Zona de agrupamento

Como se pode verificar, na zona superior do componente apareceu uma nova zona, essa zona é responsável pelo agrupamento da informação. Na zona de agrupamento existe um conjunto de botões que ilustram cada um dos campos da tabela. Para agrupar a tabela segundo um determinado campo basta apenas arrastar o seu botão para a parte superior. A figura seguinte (Figura A.3.5) é uma boa ilustração do estado final do componente depois de agrupar a tabela segundo o campo “Prioridade”.

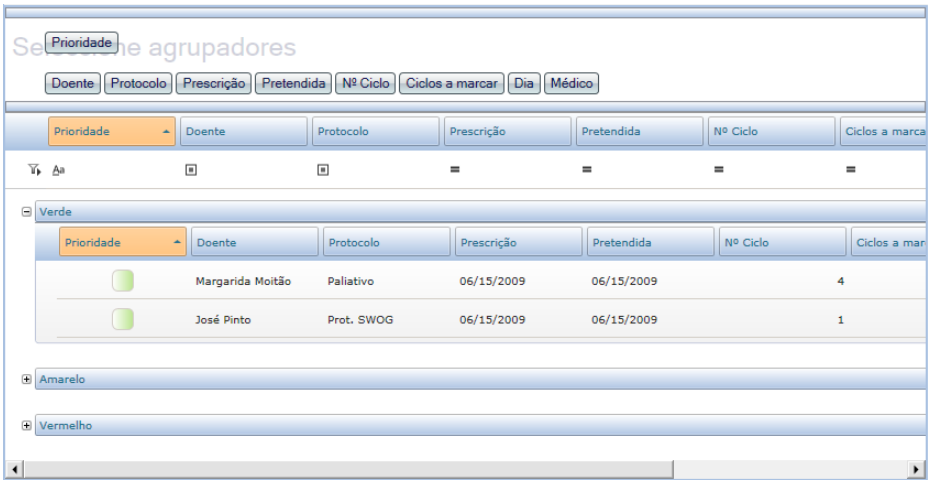

Figura A.3.5 - Exemplo de uma acção de agrupamento

Na figura anterior (Figura A.3.5) pode-se constatar as consequências de agrupar a tabela pelo campo "Prioridade". A tabela passa a mostrar um conjunto de separadores que ilustram os diversos valores possíveis para o campo escolhido. Dentro de cada um dos separadores pode-se encontrar as linhas que respeitam essa restrição.

A.3.2 Ecrã Auxiliar de Marcação do Protocolo

Neste capítulo serão apresentados os detalhes do ecrã de agendamento, dando ênfase às áreas funcionais que o constituem. Na figura seguinte (Figura A.3.6) estão destacados as principais áreas funcionais da interface.

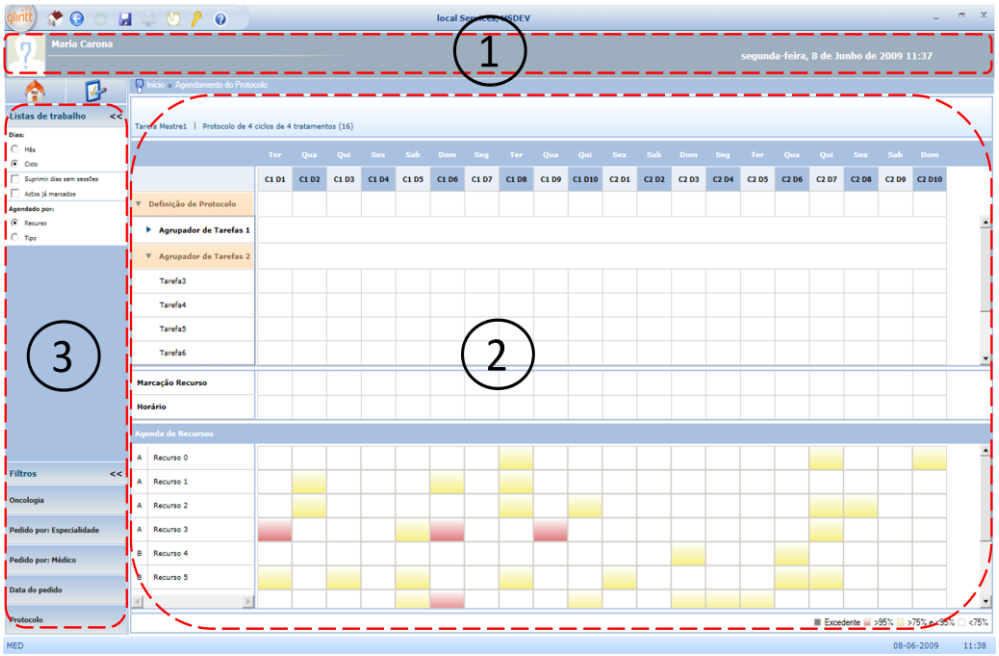

Figura A.3.6 - Zonas da interface de agendamento

A zona um é o cabeçalho da aplicação e identifica o utilizador ao qual esta associado o protocolo que está a ser agendado. O utilizador é identificado através do seu nome e imagem. O cabeçalho também fornece a data e hora actual. A zona dois é a área de trabalho, é através dela que é possível efectuar a concretização da marcação. A zona três tem duas funcionalidades distintas: por um lado permite alterar algumas propriedades da grelha de marcação, como por exemplo, agrupar os recursos por tipo ou identificar os dias do calendário (dias de ciclo, ou dias do mês); por outro lado serve de menu e disponibiliza botões para aceder a outras funcionalidades da aplicação.

Depois de fornecida uma perspectiva geral de todas as áreas disponíveis na aplicação pode-se passar para uma descrição mais detalhada da zona de marcação do protocolo (zona dois presente na Figura A.3.6). Para começar é importante apresentar os diversos componentes presentes nessa área, assim como, uma breve explicação das suas competências. A figura seguinte (Figura A.3.7) delimita cada um dos componentes, facilitando a sua identificação.

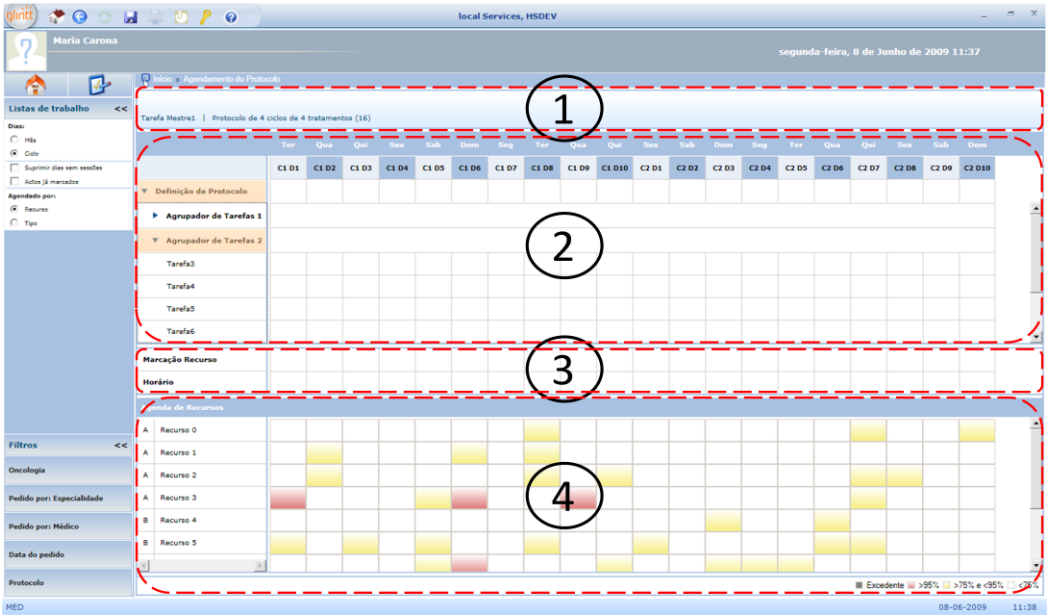

Figura A.3.7 - Principais zonas do mecanismo de marcação

- A zona um é um cabeçalho da área de marcação e fornece alguns detalhes do protocolo, como por exemplo, o nome do protocolo, o seu número de ciclos e sessões.
- A zona dois fornece a legenda das células, identificando o dia a que correspondem. Na primeira linha da grelha (definição do protocolo) é possível constatar o número de sessões dos ciclos e a sua disposição inicial ao longo do tempo. As restantes linhas da grelha identificam as diversas tarefas de um protocolo e mostram os dias em que foram marcadas.
- A zona três indica os dias em que foram agendadas as tarefas mestres. Para cada marcação é identificado o tipo de recurso alocado e o respectivo horário.
- A zona quatro é a zona interactiva e é onde os utilizadores podem realmente efectuar a marcação. As linhas representam os recursos disponíveis para a elaboração da tarefa mestre e através da selecção de uma célula é possível alocar esse recurso.

A.3.2.1 Detalhes dos Componentes da Área de Marcação

Neste capítulo pretende-se explicar os detalhes de funcionamento dos componentes da área de marcação.

Área de detalhes do protocolo

Esta componente (Figura A.3.8) pretende evidenciar todas as tarefas que compõem um protocolo, assim como, uma perspectiva geral da marcação do protocolo. A grelha mostra o agendamento de cada uma das tarefas em relação as tarefas mestres marcadas. Esta componente também serve de cabeçalho para todas as grelhas, fazendo uma correspondência entre as células e os dias. Na primeira linha da grelha é evidenciado o número de sessões por ciclo e a relação entre elas, por exemplo, a sessão dois acontece um dias depois da sessão um.

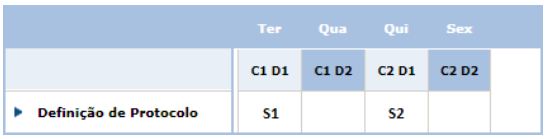

Figura A.3.8 - Figura do componente com o cabeçalho em dias do ciclo

Este componente possibilita a comutação da legenda das células, variando entre dias do ciclo ou dias do mês. A Figura A.3.8 e a Figura A.3.9 mostram as duas variantes do cabeçalho.

Figura A.3.9 - Figura do componente com o cabeçalho em dias do mês

Os detalhes do protocolo, ou seja, as tarefas que o constituem, encontram-se por defeito escondidos. Para conseguir aceder a esses detalhes é preciso pressionar a *label* "Definição de Protocolo", que controla o acesso a essa informação. Depois de abrir os detalhes do protocolo é possível conferir que as tarefas estão agrupadas por separadores (Figura A.3.10), facilitando a sua pesquisa. O controlo utilizado para efectuar esta listagem permite mostrar estruturas de dados hierarquizadas.

Figura A.3.10 - Agregadores de tarefas do protocolo

Ao abrir um separador de tarefas são apresentadas todas as tarefas que o constituem (Figura A.3.11), assim como, a correspondente grelha de visualização do agendamento.

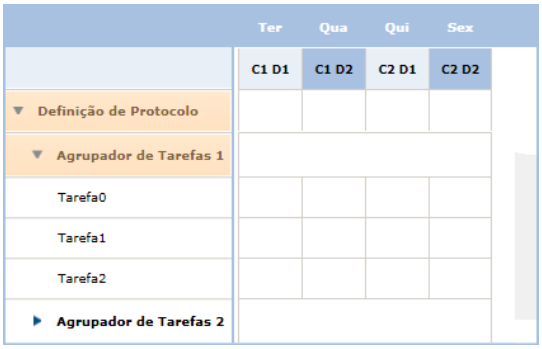

Figura A.3.11 - Figura ilustrativa da organização dos agregadores e da informação que contêm

Área de resumo das células marcadas

Esta componente pretende ser uma área de resumo das tarefas mestres marcadas na grelha de marcação. Nesta componente é possível verificar os dias em que a tarefa mestre foi marcada, o recurso que foi alocado e o respectivo horário (Figura A.3.12). O horário atribuído por defeito é a primeira hora disponível para aquele recurso.

Figura A.3.12 - Resumo da marcação das tarefas mestres

Através desta componente também é possível alterar o horário seleccionado para um recurso. Para efectuar essa operação basta apenas clicar com o botão direito do rato sobre a célula com o horário marcado. Após efectuar essa operação surge uma janela adicional com todos os horários disponíveis do recurso seleccionado (Figura A.3.13). Para alterar o horário seleccionado basta apenas seleccionar o slot pretendido e depois clicar no botão “OK”.

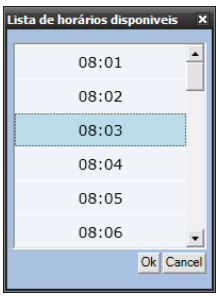

Figura A.3.13 - Janela de selecção de horários

Área de marcação da tarefa mestre

Por fim surge o componente mais importante da interface de agendamento, é a partir deste componente (Figura A.3.14) que é feita toda a marcação das tarefas mestres do protocolo. Foi dispendido bastante esforço no seu desenvolvimento, tentando aumentar o seu grau de usabilidade e simplificando a sua interface.

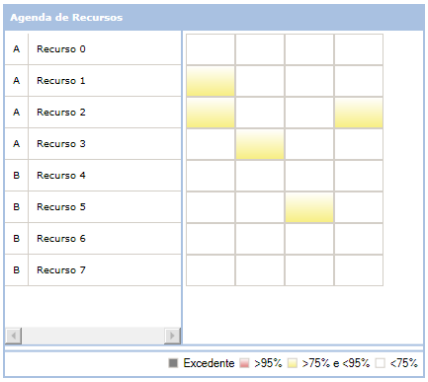

Figura A.3.14 - Grelha listada por recursos

O componente contém do lado esquerdo uma listagem dos recursos existentes que satisfazem as necessidades da tarefa mestre do protocolo seleccionado. Os recursos podem aparecer agrupados por tipo ou então expandidos (Figura A.3.14 versus Figura A.3.15 - Grelha agregada por tipo dos recursos). Ao agrupar os recursos por tipo, cada uma das linhas representara o conjunto de recursos daquele tipo. Cada uma das células da linha corresponderá ao recurso daquele tipo que tem a maior disponibilidade, facilitando assim a tarefa de escolha dos utilizadores. Caso um recurso esteja seleccionado para um determinado dia, antes de efectuar a operação de agrupamento por tipos, então naquele dia a célula que aparecerá na vista agrupada é a célula seleccionada.

Figura A.3.15 - Grelha agregada por tipo dos recursos

Do lado direito o componente contém uma grelha com a disponibilidade dos recursos por dia. Para facilitar a compreensão dos utilizadores a disponibilidade dos recursos pode ser constatada através da cor das células. Sendo o branco a cor escolhida para um grau de ocupação baixo e o vermelho para um grau de ocupação elevado dos recursos. Na zona inferior do componente pode ver-se a legenda das cores das células.

Interacção com o componente

A grelha permite a marcação da tarefa mestre através de duplo *click* sobre a célula que se pretende marcar (Figura A.3.16). A marcação de uma tarefa mestre personifica a marcação de uma sessão do protocolo, na medida em que, as outras tarefas daquela sessão são marcadas por arrasto. Para desmarcar o raciocínio é o mesmo, basta apenas fazer duplo *click* sobre a célula que se pretende desmarcar. Ao desmarcar uma sessão, todas as sessões seguintes são desmarcadas automaticamente, sendo necessário remarcar tudo outra vez.

Antes de marcar uma tarefa é necessário verificar se a célula onde se pretende marcar respeita um conjunto de restrições subjacentes ao protocolo. As principais restrições são:

- Tempo estipulado no protocolo entre sessões.
- Dia do ciclo estabelecido no protocolo para aquela sessão (existindo uma benesse, derivado ao conceito de folga).

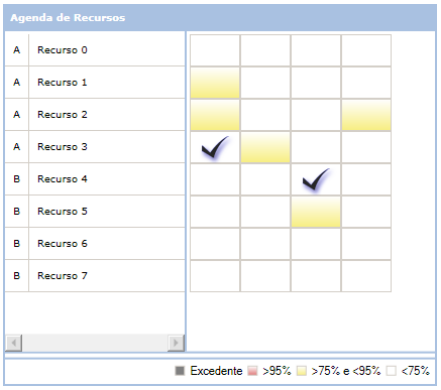

Figura A.3.16 - Exemplo de marcação de duas sessões

Tendo em conta o conceito de folga que as sessões têm, a interface permite o arrastamento de tarefas na grelha (Figura A.3.17), conduzindo a um de dois comportamentos. Ou se arrasta apenas a célula seleccionada ou então arrasta-se a célula seleccionada e todas as sessões consecutivas (arrastamento em bloco). Antes de efectuar esta operação é preciso validar se a folga definida no protocolo é congruente com o deslocamento pretendido. A folga definida no protocolo é geral a todas as sessões, sendo partilhada por pelo conjunto.

A selecção do comportamento a adoptar depois do deslocamento da célula deve ser validado pelo utilizador, surgindo uma janela onde ele é questionado sobre a abordagem a seguir.

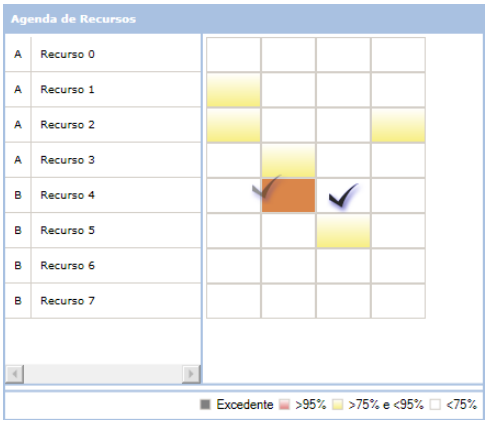

Figura A.3.17 - Exemplificação da funcionalidade de arrastamento de uma sessão

O facto de a interface proporcionar a possibilidade de agrupar as linhas por tipo de recurso, introduziu uma complexidade adicional no componente. Essa complexidade surge porque é preciso sincronizar a vista por recursos com a vista por tipo. Quando é efectuada uma alteração na vista por recursos é preciso reflecti-la na vista por tipos, assegurando a consistência da

interface. Para facilitar a compreensão da dimensão do problema será apresentado um exemplo prático onde é exemplificada a sincronização entre as duas vistas.

Em primeiro lugar optou-se por selecciona uma das células da grelha, que por acaso corresponde ao recurso dois e é do tipo A. Como se pode verificar na figura seguinte (Figura A.3.18), a cor da célula correspondente ao recurso seleccionado é amarela. Ao agrupar a grelha por recursos, a célula que ficou visível para aquele recurso no dia escolhido foi a mesma, mantendo-se assim a coerência da interface.

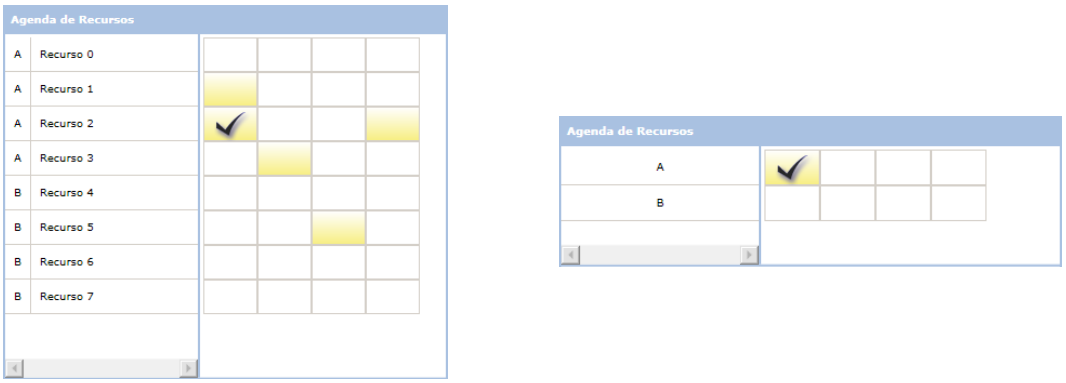

Figura A.3.18 - Ilustração inicial das duas vistas antes de efectuar a movimentação

Depois de analisada a fase inicial do exemplo, chegou a hora de alterar o recurso e o dia da marcação da sessão anterior (Figura A.3.19). Para esse efeito basta apenas clicar sobre a célula que se pretende alterar e arrastar o visto para a nova célula.

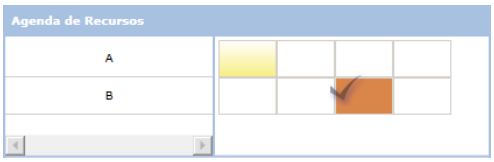

Figura A.3.19 - Deslocamento de uma marcação

Depois de efectuar a acção assinalada na figura anterior chegou a hora de analisar as suas repercussões (Figura A.3.20). Em primeiro lugar pode-se verificar que o visto referente ao recurso/dia alocado para a sessão moveu-se, deslocando-se para a célula pretendida. Em segundo plano, mas não menos importante, pode-se constatar que na vista agregada a cor correspondente a disponibilidade da célula de origem do movimento foi actualizada. Esse comportamento deve-se à necessidade de manter actualiza a célula com o recurso de maior disponibilidade.

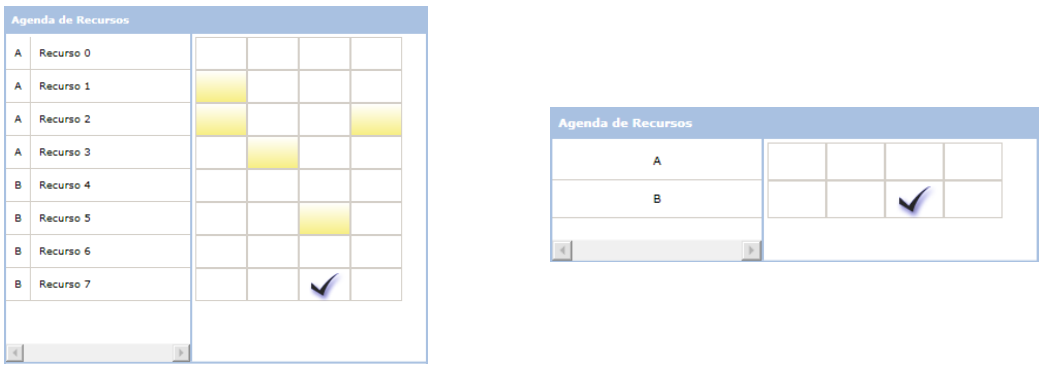

Figura A.3.20 - Ilustração final das duas vistas depois de efectuar o deslocamento

Restrições adicionais da interface

Existem algumas acções que são filtradas pelo componente, tentando assim limitar a possibilidade de ocorrerem situações erróneas ou a possibilidade de se efectuarem acções inconsistentes com a lógica de negócio da aplicação. As restrições adicionais são:

- Caso um elemento da grelha seja deslocado para fora dos limites da grelha, o seu movimento é cancelado e o elemento volta para o local inicial.
- Não é possível arrastar elementos para cima de outros. Esse comportamento será tido como anormal e o movimento será mais uma vez cancelado.
- Tentativas de deslocar componentes para trás não serão permitidas, optando-se mais uma vez por cancelar o movimento.
- Antes de efectuar um movimento em bloco é necessário verificar se essa acção era valida. Para validar a operação é necessário aferir se existe espaço disponível nos ciclos consecutivos para suportar esta operação.
- Não é possível ter mais do que um elemento marcado por coluna, já que cada elemento marcado na grelha representa uma sessão. Não é possível ter mais do que uma tarefa mestre marcada por sessão.

A.3.3 Explicação da interacção entre as componentes da área de marcação

O componente principal da área de marcação é a grelha de marcação (zona três da Figura A.3.7), as restantes componentes da área de marcação reflectem as alterações efectuadas nesse componente, tal como a figura seguinte ilustra (Figura A.3.21).

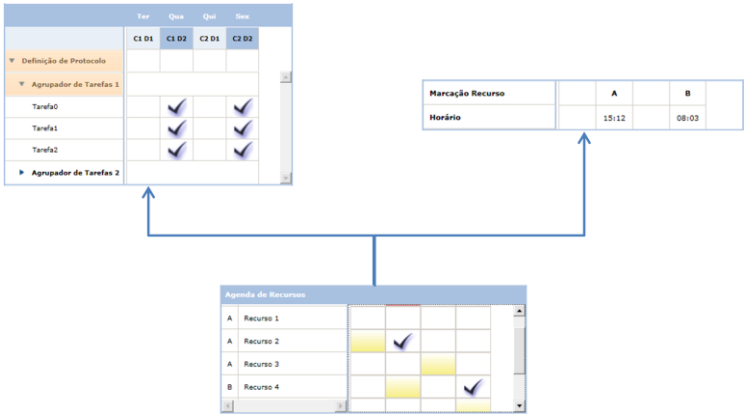

Figura A.3.21 - Hierarquia dos componentes

A figura anterior (Figura A.3.20) mostra a dependência entre os componentes, todas as operações efectuadas na grelha inferior implicam a actualização das grelhas superiores. Nesta figura pode-se verificar que na grelha inferior foram seleccionadas duas células. A selecção dessas células implicou a actualização dos outros componentes, eliminando possíveis incongruências. Na grelha superior do lado direito (zona três da Figura A.3.7), pode-se verificar que foram sinalizados os dias marcados. A grelha superior do lado esquerdo (zona dois da Figura A.3.7), por sua vez também foi actualizada, marcando as restantes tarefas correspondentes à sessão em causa.

Quando são efectuados deslocamentos na zona de marcação, as alterações também têm de ser reflectidas nas restantes componentes. Caso ocorra uma deslocação em bloco, então nas restantes zonas os elementos também têm de se mover em bloco.

Anexo B

Descrição do Módulo de Prescrição e Agendamento

B.1 Resumo

Neste anexo é apresentado em primeiro lugar o contexto onde está inserida a funcionalidade de prescrição e agendamento. O passo seguinte é documentar essa funcionalidade, apresentando cada uma das suas interfaces e o seu ciclo de funcionamento.

B.2 Apresentação do Contexto

A funcionalidade de prescrição e agendamento de protocolos foi integrada no panorama do processo clínico, tendo sido necessário adapta-la ao seu conceito. Neste capítulo é apresentando o ambiente envolvente da interface de agendamento, expondo às áreas funcionais que são herdadas do processo clínico.

A figura seguinte (Figura B.2.1) é uma ilustração do ecrã inicial do agendamento, é possível através dela apresentar as diversas áreas funcionais que são herdadas do processo clínico.

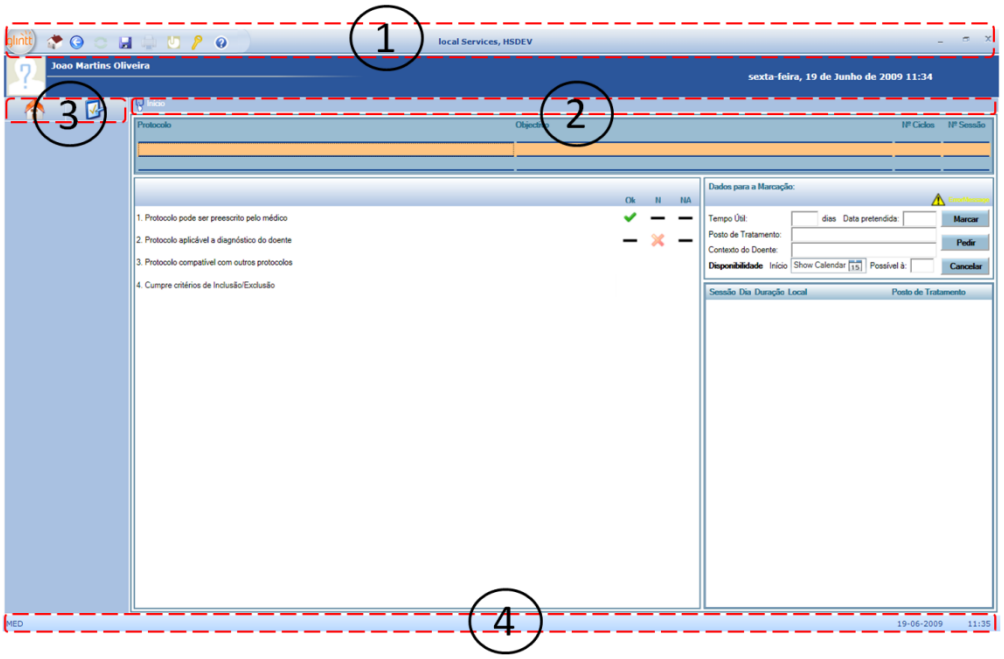

Figura B.2.1 - Zonas envolventes do agendamento

Na figura anterior (Figura B.2.1) estão identificadas as zonas funcionais herdadas do processo clínico. Cada uma das zonas está envolvida por uma linha a tracejado e está numerada.

- A zona um é uma barra de tarefas que fornece algumas funcionalidades disponibilizadas pela aplicação principal, tais como, opções de navegação, guardar informação, ajuda e voltar ao ecrã principal. Para além disso também facilita o acesso a um menu geral onde estão listadas as principais funcionalidades do processo clínico.
- A zona dois é um *BreadCromp*, através dele é possível verificar o caminho percorrido até à opção actual e caso seja necessário pode-se navegar para as opções anteriores.
- A zona três tem dois botões, um para fechar o processo clínico e o outro para voltar ao início do processo clínico.
- A zona quatro é uma barra de estado, em que aparece o nome do utilizador, a data actual e a hora.

B.3 Apresentação dos ecrãs do módulo de prescrição e agendamento

O módulo de prescrição e agendamento é composto por dois ecrãs, um ecrã de prescrição do protocolo ao paciente e outro ecrã de confirmação dos critérios de inclusão e exclusão dos protocolos. O médico começa pelo ecrã de prescrição e escolhe o novo protocolo do paciente. Uma vez seleccionado o novo protocolo, o médico deve preencher todos os campos requisitados. Antes de o médico poder efectuar a marcação do protocolo tem de passar pelo ecrã de confirmação dos critérios de inclusão e exclusão do protocolo, só depois é possível efectuar a marcação com sucesso.

De seguida são apresentados os dois ecrãs em detalhe, assim como o seu modo de funcionamento.

Ecrã de inicial da prescrição

O ecrã inicial de prescrição é composto por três zonas: o cabeçalho da aplicação, onde aparece a informação do utilizador e a data actual; uma zona de listagem dos protocolos do paciente e outra zona de caracterização do novo protocolo que se pretende adicionar ao paciente. A figura seguinte (Figura B.3.1) ilustra convenientemente o que foi descrito e divide o ecrã por zonas para ser mais fácil.

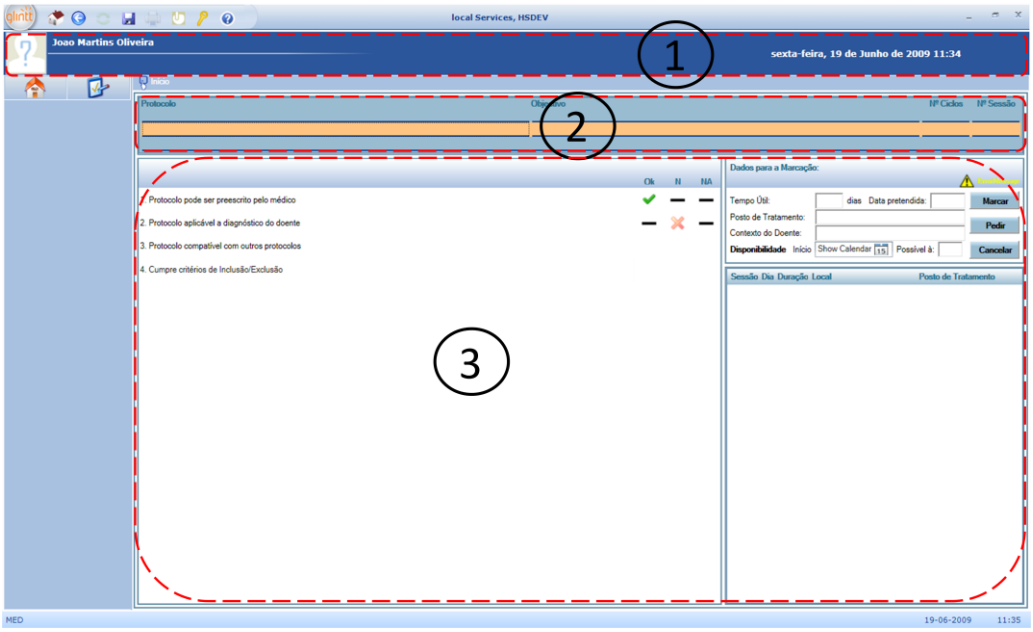

Figura B.3.1 - Ecrã inicial da prescrição e agendamento

É a partir da zona de listagem dos protocolos do paciente (Figura B.3.2) que todo o processo de prescrição inicia. O médico deve adicionar um novo protocolo a lista e só depois pode preencher os seus detalhes na zona de trabalho (Figura B.3.3).

A forma como é adicionado o novo protocolo à lista ainda não está acordada, é necessário definir a forma como se acede a uma listagem de todos os padrões pré-definidos existentes. Existem duas possibilidades viáveis de aceder à listagem de protocolos padrão: ou se adiciona um botão ao menu do lado esquerdo; ou então tem de se clicar com o botão direito do rato sobre a listagem de protocolos do paciente.

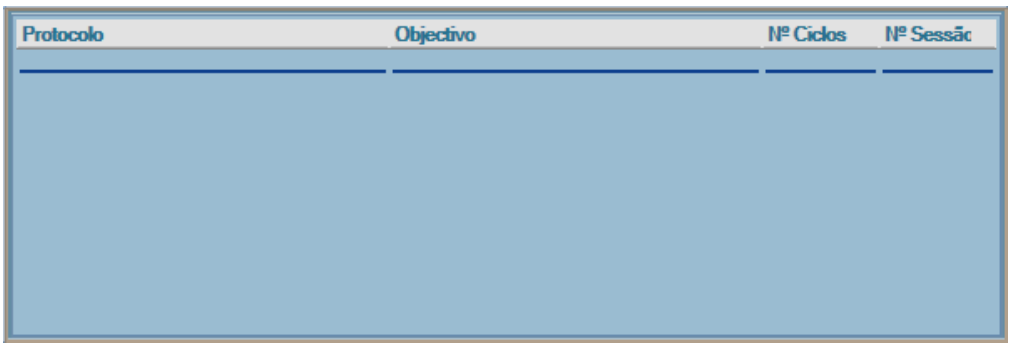

Figura B.3.2 - Zona de listagem dos protocolos do paciente

Uma vez adicionado o novo protocolo passa-se para a zona de trabalho (Figura B.3.3) onde são preenchidos os campos necessários para efectuar a marcação do protocolo.

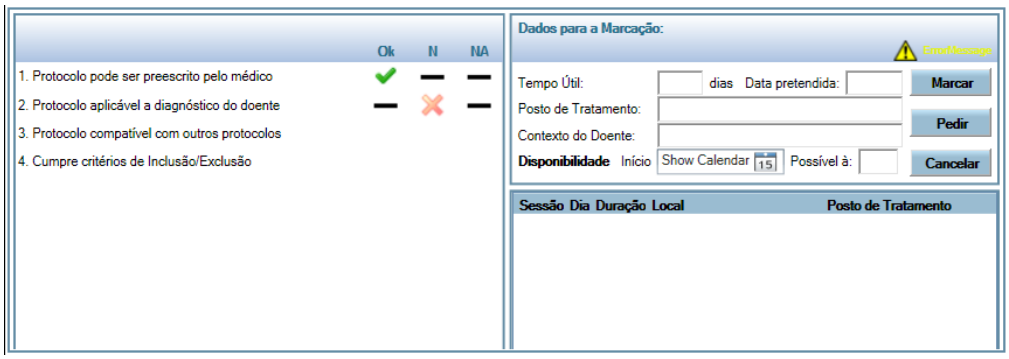

Figura B.3.3 - Área de trabalho do ecrã de prescrição e agendamento

A zona de trabalho é composta por três partes: zona de validação; zona de preenchimento dos detalhes e zona de apresentação das actividades do protocolo (apenas de consulta). Na figura seguinte (Figura B.3.4) é possível verificar as zonas, elas estão claramente identificadas.

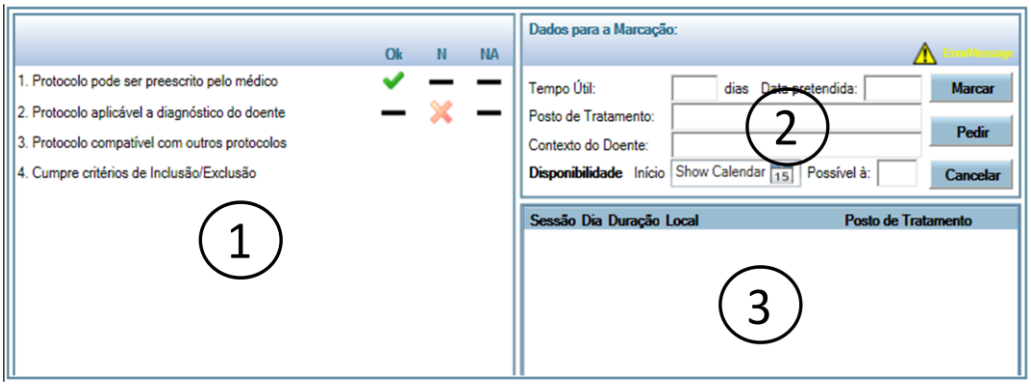

Figura B.3.4 - Divisão da área de trabalho em zonas

Tal como já foi dito, antes de poder efectuar a marcação (na zona 2, botão marcar) é necessário validar se é possível, para isso é necessário consultar a zona 1. Na zona 1 a aplicação verifica se um conjunto de critérios são satisfeitos, critérios esses que são obrigatórios para poder ser efectuada a marcação. De entre todos os critérios existe apenas um que é validado

explicitamente pelo médico, que são os critérios de inclusão e exclusão de um protocolo. Para confirmar os critérios é necessário aceder ao ecrã de confirmação, que está acessível através de um clique com o botão esquerdo do rato sobre a opção 3 da zona 1.O ecrã de confirmação de critérios é apresentado em detalhe no ponto seguinte.

Antes de passar para o ecrã de confirmação é importante referir que depois de tudo validado e dos campos do protocolo estarem preenchidos é possível passar à sua marcação (zona 2 botão "Marcar"). Caso a operação não ocorra com sucesso (não existam vagas que respeitem as restrições), então o médico pode efectuar adicionar um pedido de marcação ao sistema (zona 2 botão "Pedir") ou então cancelar a prescrição do protocolo (zona 2 botão "Cancelar").

Ecrã de confirmação dos critérios de inclusão e exclusão do protocolo

O ecrã de confirmação dos critérios de inclusão e exclusão pode ser visualizado na figura seguinte (Figura B.3.5).

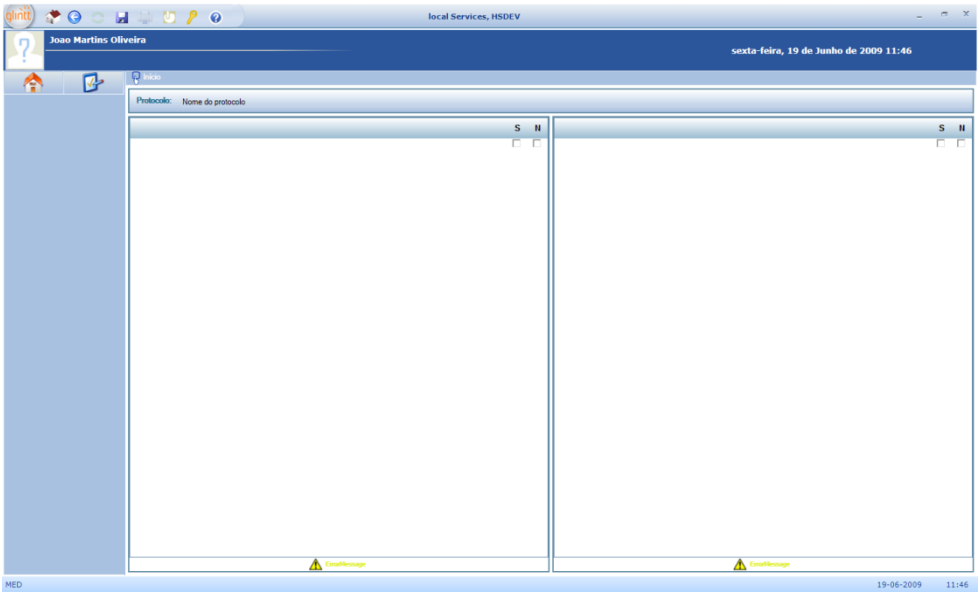

Figura B.3.5 - Ecrã de confirmação dos critérios de inclusão e exclusão de um protocolo

O ecrã de confirmação contém apenas um componente (Figura B.3.6) onde é possível confirmar os critérios. O componente tem uma zona para os critérios de inclusão (esquerda) e para os de exclusão (direita). Cada um dos critérios é representado por uma linha na zona correspondente e é lhe associado duas *checkboxes*, uma para o sim (S) e outra para o não (N). Por defeito os critérios de inclusão aparecem com o não seleccionado e os de exclusão com o sim. A tarefa do utilizador é trocar o valor das checkboxes, reconhecendo assim que tomou conhecimento de todas as restrições do protocolo.

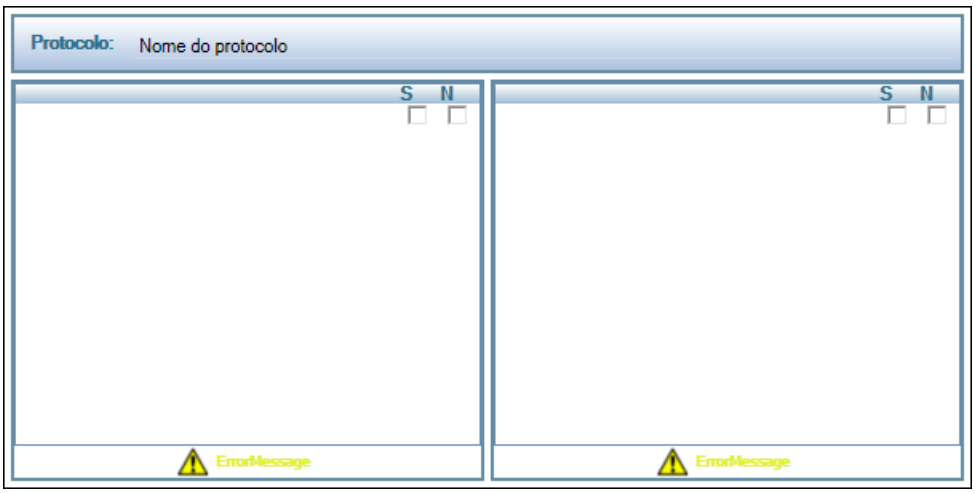

Figura B.3.6 - Zona de confirmação dos critérios de inclusão e exclusão do protocolo

Anexo C

Protótipos do Módulo de Marcação

C.1 Introdução

Neste anexo são apresentados e descritos os primeiros protótipos que foram gerados no âmbito da definição do módulo de gestão de conflitos, que com o desenrolar do projecto passou a chamar-se módulo de marcação.

C.2 Protótipos e Descrição

Em primeiro lugar é apresentado o ecrã de listagem dos Protocolos (Figura C.2.1) em lista de espera para marcação (devido à impossibilidade de marcar para a data pretendida). Neste ecrã pretende-se apenas escolher um protocolo para marcar.

Ecrã inicial

Figura C.2.1 - Ecrã inicial do módulo de gestão de conflitos

Uma vez apresentado o ecrã inicial e as suas funções, é necessário efectuar uma descrição mais detalhada das suas características. Para isso é feita uma divisão em zonas da figura anterior (Figura C.2.1) e depois é feita uma descrição breve de cada uma. Na figura seguinte (Figura C.2.2) é possível ver a divisão.

Figura C.2.2 - das principais zonas do ecrã inicial do módulo de gestão de conflitos

A figura anterior (Figura C.2.2) é composta por cinco zonas, sendo que cada uma delas tem a seguinte função:

1. Título da aplicação e informação do dia
2. Agregadores da tabela dos protocolos pendentes de marcação
3. *BreadCrum*, indica a funcionalidade actual e qual foi o percurso efectuado
4. Tabela dos protocolos pendentes de marcação

5. Barra de navegação

Uma vez apresentadas as zonas, de seguida são apresentadas algumas das funcionalidades do ecrã. Na figura seguinte (Figura C.2.3) é possível verificar qual é o impacto dos agregadores na listagem de protocolos pendentes.

Figura C.2.3 - Demonstração dos agregadores no ecrã inicial

Na imagem anterior (Figura C.2.3) é possível verificar o estado final da listagem depois de seleccionado o agregador "Protocolo". A listagem passa a estar agrupada pelos protocolos similares.

Para além dos agregadores existe outra funcionalidade que pretende facilitar a busca de protocolos com certas características, os filtros. A grelha fornece uma zona com um conjunto de filtros, tal como se pode ver na figura seguinte (Figura C.2.4).

Figura C.2.4 - Filtros da listagem de protocolos

A zona de filtros possibilita filtrar cada um dos campos individualmente, oferecendo ao utilizador uma funcionalidade mais completa e que se adapta melhor às suas necessidades. Para facilitar a compreensão dos filtros, na figura seguinte (Figura C.2.5) é demonstrado um caso prático.

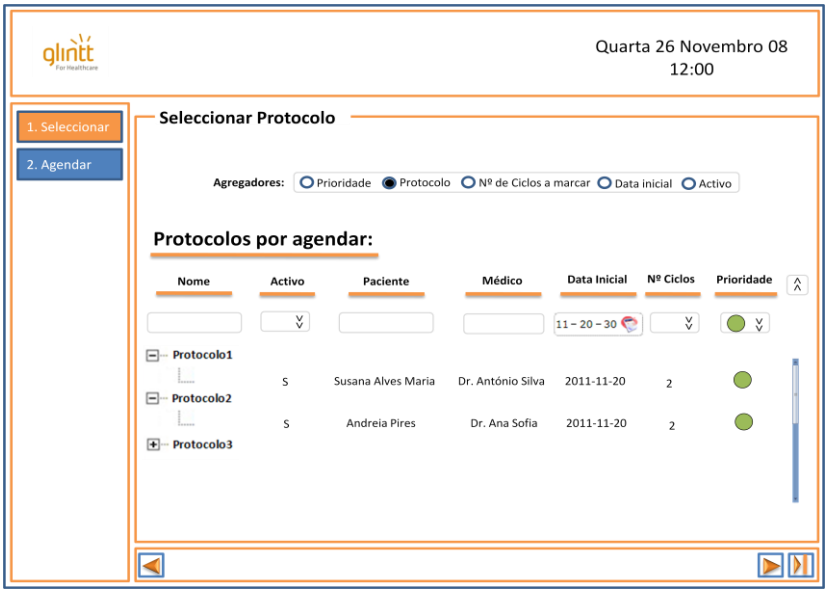

Figura C.2.5 - Demonstração da utilização dos filtros

Como se pode verificar na figura anterior (Figura C.2.5), a listagem foi filtrada pelo campo prioridade, sendo apenas visíveis os protocolos em lista de espera com prioridade alta (“Verde”).

Ecrã de marcação (por dias)

Depois de apresentado o ecrã inicial passa-se agora para a exposição do ecrã de marcação dos protocolos. O ecrã de marcação de protocolos surge depois de ser seleccionado um protocolo no ecrã inicial e utilizada a opção de avançar na barra de navegação. A aparência do ecrã de marcação pode ser vista na figura seguinte (Figura C.2.6).

Figura C.2.6 - Ecrã de marcação de um protocolo

Utilizando a mesma abordagem do ecrã inicial, é feita uma divisão em zonas do ecrã de marcação, tal como se pode verificar na imagem seguinte (Figura C.2.7).

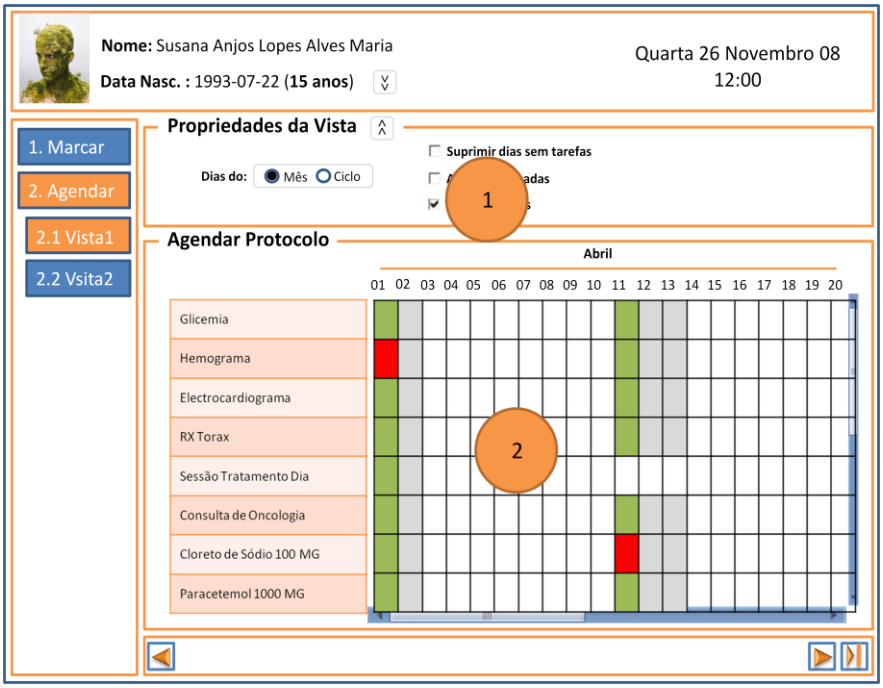

Figura C.2.7 - Divisão do ecrã de marcação de protocolos

Na figura anterior (Figura C.2.7) apenas são identificadas as zonas distintas do ecrã inicial, partindo-se do pressuposto que as demais desempenham o mesmo papel.

1. Propriedades da vista, permite definir as propriedades de visualização da grelha de marcação:
   a. Dias do – escolhe a legenda da grelha
      i. Mês – mostra os dias do mês
      ii. Ciclo – mostra os ciclos e os seus dias
   b. Suprimir dias sem tarefa – caso esta opção esteja activa os dias em que não existem tarefas não aparecem na grelha
   c. Actos já marcados – caso esta opção esteja activa aparecem na vista as tarefas que o paciente já tem marcadas e que são externas ao protocolo em questão.
   d. Mostrar folgas – caso esta opção esteja activa a grelha mostra as possíveis folgas de uma tarefa.
2. Grelha onde é possível marcar todas as tarefas (na perspectiva dos dias).

Uma vez apresentadas as zonas do ecrã de marcação é importante agora mostrar como cada uma das suas opções funciona e demonstrar como se pode efectuar uma marcação. De seguida são apresentadas figuras ilustrativas de todos os processos.

Propriedade "Dias do"

Na figura seguinte (Figura C.2.8) é possível verificar o estado final da legenda da grelha depois de alterado o modo de "Mês" para "Ciclo".

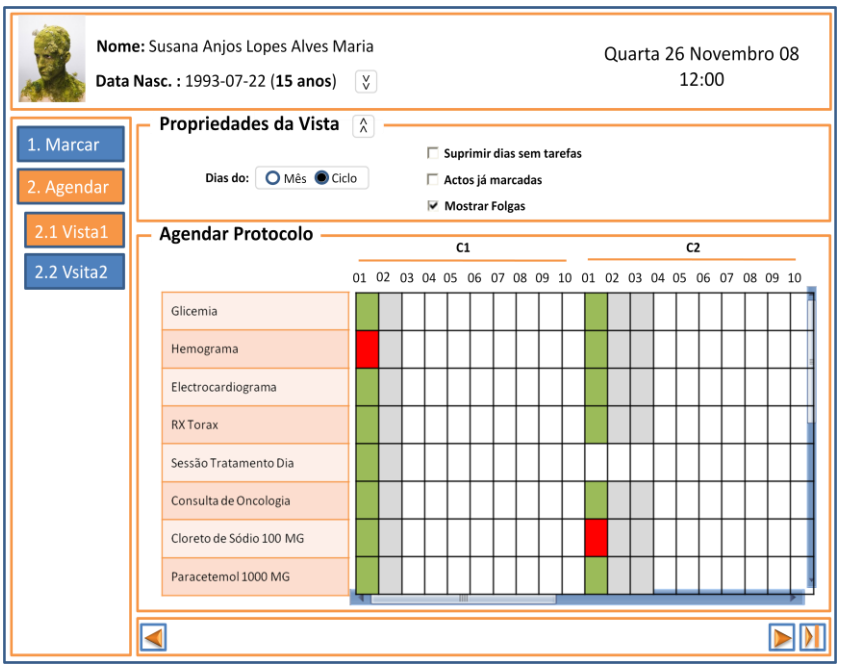

Figura C.2.8 - Ilustração da propriedade “Dias do”

Como se pode verificar, a legenda das colunas da grelha de marcação passou a ilustrar os dias em relação ao ciclo e não ao mês.

Propriedade “Mostrar folgas”

A grelha tem uma opção que permite mostrar as folgas das actividades do protocolo. Quando a opção está activa as folgas são ilustradas na grelha através da cor cinzento, como se pode ver na imagem anterior (Figura C.2.8). Ao desactivar essa opção, a informação das folgas desaparece e a grelha passa ao estado da figura seguinte (Figura C.2.9).

Figura C.2.9 - Ilustração da propriedade "Mostrar Folgas"

Propriedade "Actos já marcados"

A grelha oferece uma funcionalidade que permite mostrar as actividades que o paciente já tem marcadas e que são externas ao protocolo em causa. Esta funcionalidade permite ao utilizador ter uma percepção do panorama geral do paciente e facilita a gestão de possíveis conflitos. Na figura seguinte (Figura C.2.10) é possível verificar o que acontece á grelha depois de activar a propriedade "Actos já marcados".

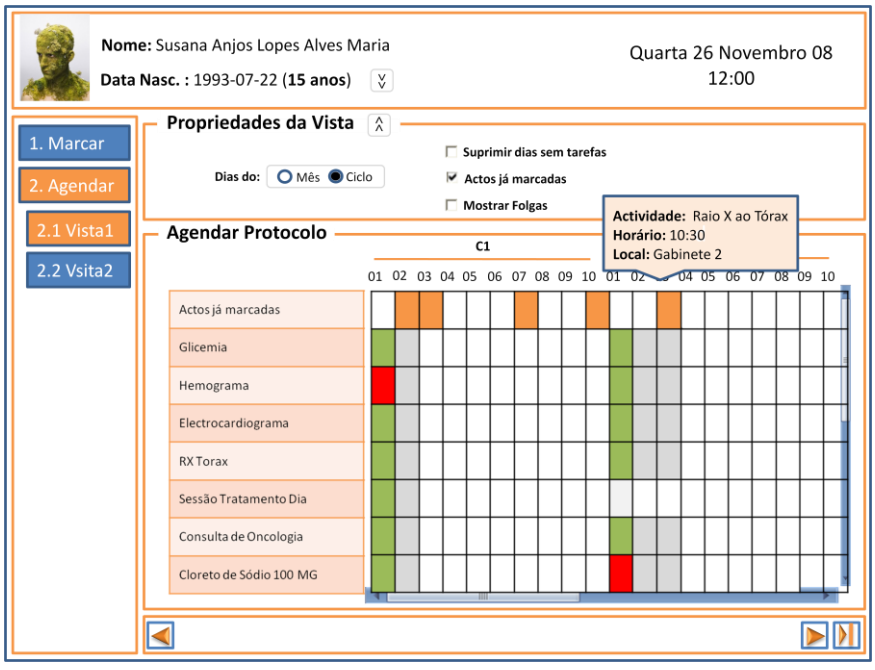

Figura C.2.10 - Ilustração da propriedade "Actos já marcados"

Pode-se verificar na figura anterior (Figura C.2.10) que as actividades já marcadas aparecem na primeira linha da grelha e são identificadas através de uma célula laranja. Ao passar com o rato sobre uma célula laranja surge um *pop-up*[9] que fornece informação sobre as actividades que estão marcadas para aquele dia.

Propriedade "Suprimir dias sem tarefas"

Esta propriedade permite apresentar na grelha apenas os dias que têm actividades marcadas e as suas folgas. Na figura seguinte (Figura C.2.11) é possível verificar o estado final da grelha depois de activar a propriedade (face á Figura C.2.10).

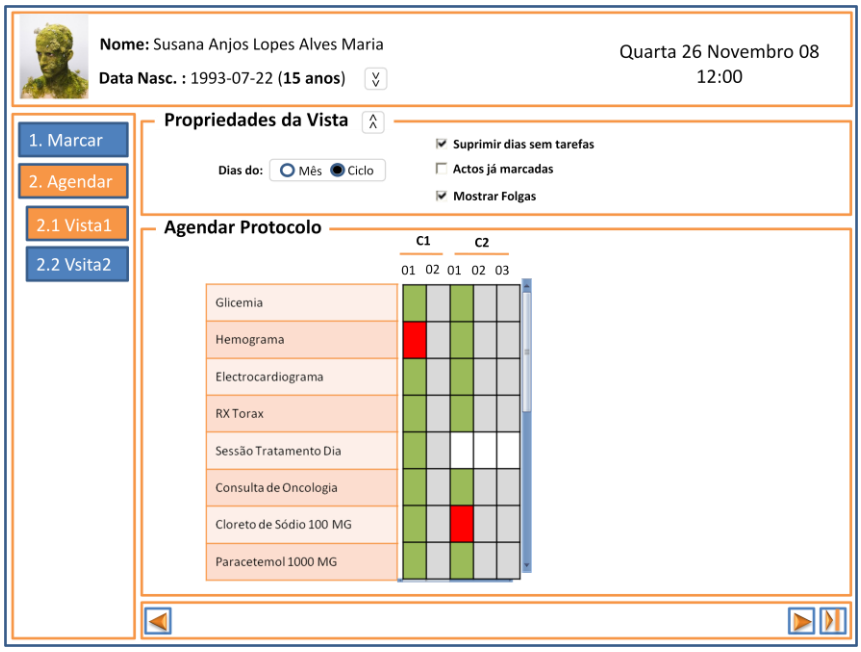

Figura C.2.11 - Ilustração da propriedade "Suprimir dias sem tarefas"

Alteração do horário de uma marcação

Após efectuar a marcação de uma actividade, a grelha permite que seja alterado o horário dessa marcação. Para isso basta apenas clicar com o botão direito do rato na célula pretendida e surge uma *pop-up* com os horários disponíveis para esse dia. Uma vez apresentada a lista o utilizador apenas necessita de escolher o novo horário pretendido A figura seguinte (Figura C.2.12) mostra uma ilustração deste processo.

[9] Pop-up – é uma janela extra que surge no ecrã principal e é utilizada para mostrar informação adicional

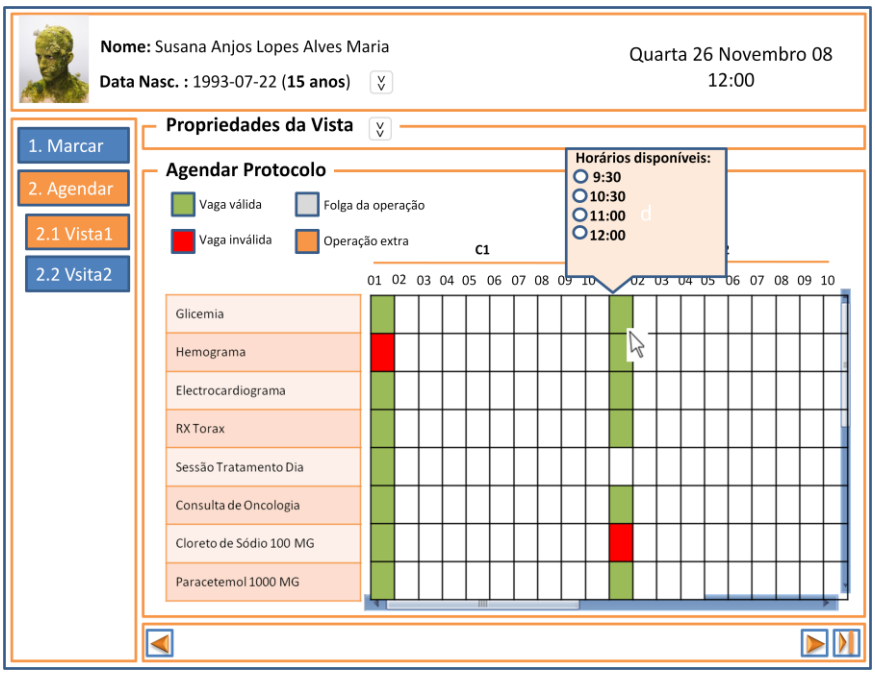

Figura C.2.12 - Ilustração da alteração do horário de uma marcação

Mover em bloco as marcações de um dia

Para facilitar a alteração em bloco da marcação das tarefas de um dia, a grelha permite que sejam arrastadas todas as actividades de um determinado dia, assim como todas as actividades que lhes sucedem. Para efectuar essa operação basta apenas arrastar a legenda do dia que se pretende mover e depois largar no dia pretendido. A figura seguinte (Figura C.2.13) é uma boa ilustração desse processo.

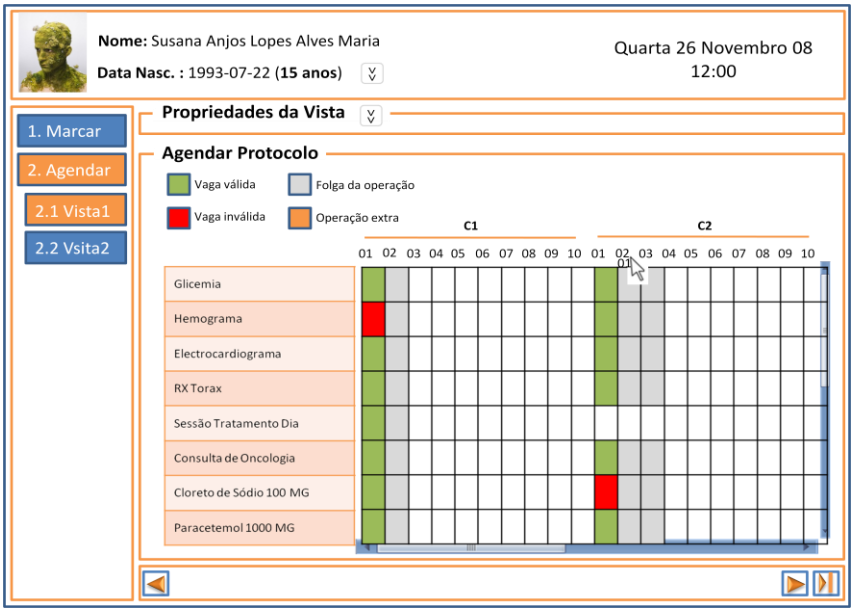

Figura C.2.13 - Ilustração do arrastamento em bloco de um conjunto de actividades

Ecrã de marcação (num dia)

Durante a marcação de um protocolo as vezes é necessário ter uma perspectiva mais detalhada de um dia da marcação. Esta perspectiva é útil para optimizar a agenda de um paciente, minimizando o seu desconforto e aproximando os horários das diversas actividades que necessita efectuar. Na figura seguinte (Figura C.2.14) é possível verificar o modo como a grelha é apresentada nesta nova vista.

Figura C.2.14 - Ecrã de marcação na perspectiva de um dia apenas

Ao observar a figura anterior (Figura C.2.14) é possível verificar que a legenda das colunas se alterou, agora em vez de mostrar os dias mostra as horas. Através desta vista é possível ter uma maior sensibilidade em relação à agenda de um paciente para um dia, sendo possível efectuar pequenas alterações nos horários. A dimensão temporal das células pode ser alterada, para isso basta apenas utilizar a barra que se encontra no canto superior direito da grelha. A possibilidade de alterar a dimensão temporal das células, permite ao utilizador adaptar melhor a vista à tarefa que esta a marcar, passando as células a ter a dimensão da duração da tarefa. Na figura seguinte (Figura C.2.15) é possível verificar como fica a grelha depois de alterar a dimensão de uma célula (de 30 para 10 minutos).

Figura C.2.15 - Alteração do tempo que cada uma das células representa

C.3 Opinião do cliente em relação aos protótipos sugeridos

O cliente (interno à empresa) gostou do conceito geral da aplicação, embora tivesse considerado que o ecrã de marcação tivesse um bocado incompleto e fosse necessário explorar alguns conceitos.

Na opinião do cliente o ecrã inicial de listagem de protocolos está quase completo, falta apenas adicionar uma lista com alguns filtros padrão, agilizando algumas buscas mais frequentes.

Quanto ao ecrã de marcação a história é diferente o cliente enumerou uma lista de recomendações que são necessárias seguir:

- Gostou da ideia dos "Actos já marcados" e considera que pode trazer algum valor acrescentado ao produto, embora pense que a curto prazo não seja fundamental (baixa prioridade).
- A forma como a grelha indica as folgas das actividades não é satisfatória para ele, é necessário ponderar outra solução para o problema. Talvez através da colocação de um pequeno símbolo nas células, em vez da célula toda marcada.
- Disponibilidade dos recursos, o cliente pretende que em cada célula apareça a sua disponibilidade de marcação. A disponibilidade deve aparecer por cores, mostrando a taxa de ocupação dos recursos.
- As células marcadas devem ser ilustradas através de um símbolo, por exemplo um visto. A situação actual, em que as células marcadas apenas indicam se a sua marcação é valida ou não, não chega, é preciso aperfeiçoar o mecanismo.

- O cliente também gostou da Vista2 da grelha, em que é mostrada a perspectiva de apenas um dia. Tal como no caso dos "Actos já marcados", também esta vista deve ficar em segundo plano, sendo concretizada numa fase mais avançada do produto.
- Por último o cliente sugeriu que a forma como é feito o deslocamento da marcação de uma actividade fosse diferente. O cliente pretende que em vez de arrastar a coluna se possa arrastar a célula da actividade e que no final, ao largar no sítio pretendido, se possa escolher entre deslocar em bloco ou unitário.

Anexo D

Guidelines para migrar um módulo de CAB para PRISM

D.1 Estrutura base em CAB

Ao longo deste documento assume-se que o módulo CAB que se deseja migrar para PRISM segue uma estrutura semelhante à utilizada pela *Smart Client Software Factory* (Figura D.1.1).

Segundo esses padrões, a inicialização de um módulo será feita por uma classe que derive de *ModuleInit*. Nessa classe será injectada uma referência para o "root work item". No método Load() é adicionada uma nova instância de um *WorkItem* que será responsável por adicionar serviços, instalar handlers de comandos e eventos e criar as *Views* necessárias.

Cada *View* obtém, por *Dependency Injection*, uma nova instância do respectivo *Presenter*. De acordo com o padrão *Model-View-Presenter*, a *View* contém apenas o código necessário para gerir os seus controlos, reencaminhando os eventos gerados pela interacção com o utilizador para o *Presenter*. Este, por sua vez, é responsável por tratar os eventos gerados pela *View* e actualizar o estado dos controlos de acordo com a lógica de negócio. Para isso, a *View* deve expor uma interface que permita ao *Presenter* actualizar o estado da *View*. O *Presenter* apenas mantém uma referência para a interface.

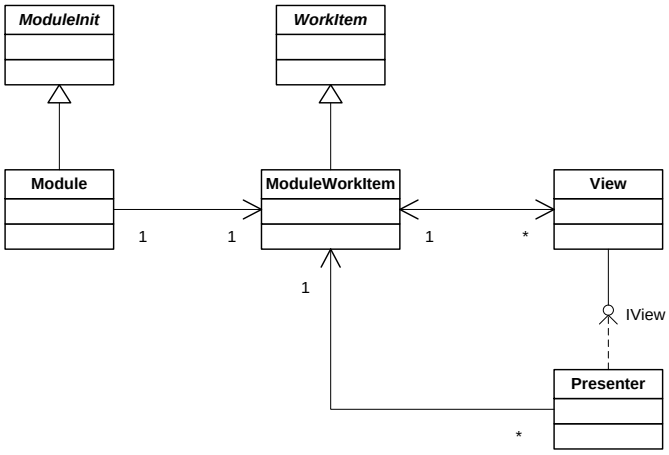

Figura D.1.1 - Modelo de referência usado em CAB

Sempre que for necessário actualizar uma determinada *View* em resposta a um evento que acontece noutra *View*, a comunicação deve ser realizada utilizando eventos entre os respectivos *Presenters (*Figura D.1.2*)*.

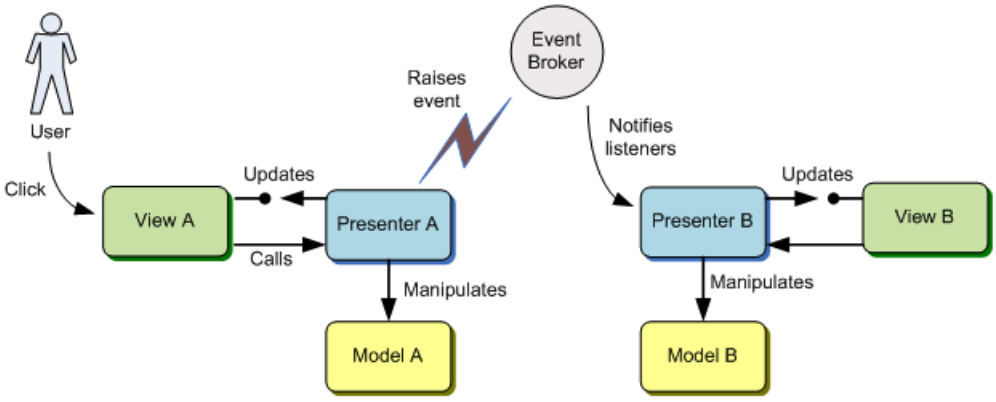

Figura D.1.2 - Ilustração da comunicação entre *views*

D.2 Alterações no projecto de Infra-estrutura

As *guidelines* apresentadas neste documento para migração de módulos CAB para um projecto PRISM dependem de alguns elementos adicionais no projecto de Infra-estrutura.

O projecto de Infra-estrutura deve conter os seguintes elementos (incluídos juntamente com este documento):

- Controller.cs
- GlobalCommands.cs
- IExtensibleShell.cs
- Presenter.cs
- PrismState.cs
- Hosts:
    - HostWinForm

- HostWpfControl

O projecto de Infra-estrutura deve conter as seguintes referências:

- Assemblies necessárias para CAB (CompositeUI, CompositeUI.WinForms, CompositeUI.WPF, ObjectBuilder);
- Assemblies necessárias para Prism (Composite, Presentation, UnityExtensions, ObjectBuilder2, ServiceLocation, Unity);
- PresentationCore;
- PresentationFramework;
- System.Drawing;
- WindowsBase;
- WindowsFormsIntegration.

D.3 Migração de um módulo

O primeiro passo para migrar um módulo de CAB para PRISM será adicionar as referências necessárias para PRISM.

D.3.1 Correspondência de conceitos

Na tabela seguinte (Tabela D.3.1) pode-se ver uma correspondência entre os principais conceitos de CAB para PRISM. A grande diferença reside no conceito de *WorkItem*, que em PRISM não existe.

Tabela D.3.1 Correspondência de conceitos entre CAB e PRISM

CAB	PRISM
IWorkspace	IRegion (O acesso é feito através do RegionManager)
WorkItem	Controller (classe abstracta em anexo)
WorkItem como container	UnityContainer incluído no Controller
ModuleInit	IModule

D.3.1.1 Hierarquia de WorkItems

Em CAB, todos os *WorkItems* fazem parte de uma estrutura em árvore, cuja raiz é o *RootWorkItem*. Em PRISM, essa noção corresponde a uma hierarquia de *UnityContainers*, cada um deles contendo um *WorkItem*. A correspondência de um *UnityContainer* a cada *Workitem* permite simular a hierarquia existente em CAB (Figura D.3.1).

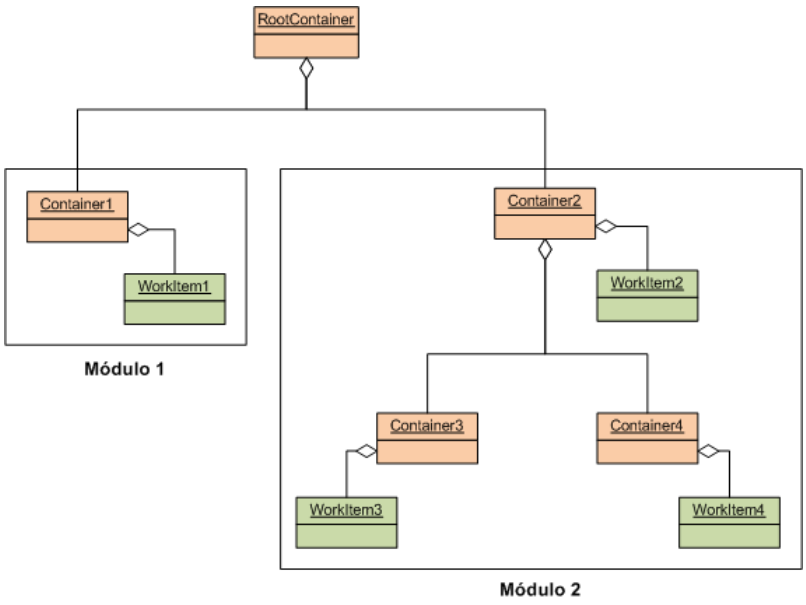

Figura D.3.1 - Simulação da hierarquia de WorkItems utilizando containers

D.3.2 Classe ModuleInit

A classe de inicialização do módulo, que em CAB descende de *ModuleInit*, deverá passar a descender de Microsoft.Practices.Composite.Modularity.IModule. As propriedades que eram utilizadas para injectar o "parent WorkItem" devem ser removidas, uma vez que já não são necessárias. Deve ser criado um novo construtor para a classe que permita a injecção do *RegionManager* e do *UnityContainer*:

Tabela D.3.2 - Diferenças na inicialização do módulo (recepção de propriedades)

CAB	`private WorkItem parentWorkItem;`
	
	`[ServiceDependency]`
	`public WorkItem ParentWorkItem`
	`{`
	`  set`
	`  {`
	`  parentWorkItem = value;`
	`  }`
	`}`

PRISM	`private IRegionManager regionManager;`
	`private IUnityContainer unityContainer;`
	
	`public CABSampleModuleInit(IRegionManager regionManager, IUnityContainer unityContainer)`
	`{`
	`  this.regionManager = regionManager;`
	`  this.unityContainer = unityContainer;`
	`}`

A interface *IModule* obriga a que seja implementado um método "Initialize()", que será responsável pela inicialização do módulo. Esse método poderá apenas chamar o método "Load()" já definido (que era usado para o mesmo propósito que em CAB). Uma vez que a classe de inicialização já não é subclasse de *ModuleInit*, o método "Load()" já não pode ser *override*.

Caso existam *handlers* de comandos instalados na classe de inicialização, estes devem ser convertidos segundo as indicações em D.3.4.2 .

D.3.2.1 Como correr um WorkItem (Controller)?

Tabela D.3.3 - Como migrar a criação e lançamento de um *WorkItem*

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

PRISM	`IUnityContainer child = unityContainer.CreateChildContainer();`
	`child.RegisterInstance<IUnityContainer>(child);`
	`MyWorkItem workItem = new MyWorkItem(regionManager, child);`
	`child.RegisterInstance<MyWorkItem>(workItem);`
	`unityContainer.RegisterInstance<MyWorkItem>(workItemName, workItem);`
	`workItem.Parent = this;  //ou null caso o work item seja lançado do ModuleInit`
	`workItem.regionName = regionName;`
	`workItem.Run();`
	`workItem.Activate();`

Em PRISM não existe uma hierarquia de *WorkItems*, mas pode ser simulada utilizando uma hierarquia de *Containers*. Ao lançar um novo *WorkItem*, é criado um novo *Container* onde ele é registado. O novo *WorkItem* também é registado no *Container*-pai, associado a um nome. Isto permite que existam vários *WorkItems*-filho do mesmo tipo.

Caso sejam necessários parâmetros para a configuração inicial do *WorkItem*, eles devem ser adicionados como propriedades do *WorkItem*, que são preenchidas antes de se chamar o método "Run()". Na tabela anterior (Tabela D.3.3) é possível verificar essa situação, a propriedade *regionName* é um bom exemplo disso.

A propriedade *Parent* apenas deve ser colocada a *null* sempre que se criar um *WorkItem* na raiz da hierarquia de um módulo.

D.3.3 WorkItems

Segundo a correspondência de conceitos acima, todas as classes que derivem de *WorkItem* devem passar a derivar de uma nova classe *Controller*, que permitirá emular algumas funcionalidades do *WorkItem*. Os passos necessários para a migração de um *WorkItem* são os seguintes:

1. Fazer *override* aos métodos "Run()" e "Activate()" definidos na classe *Controller*. O método "Run()" será responsável por todas as inicializações necessárias, enquanto o método "Activate()" será responsável pela activação propriamente dita das *Views*.
2. Para que seja possível mostrar as *Views*, é necessário definir um construtor que permita injectar o *RegionManager* e o *UnityContainer*:

Tabela D.3.4 - Particularidades do construtor do *Controller* em PRISM

PRISM	`private IRegionManager regionManager;`
	`private IUnityContainer unityContainer;`
	
	`public SampleWorkItem(IRegionManager regionManager, IUnityContainer unityContainer)`
	`{`
	`this.regionManager = regionManager;`
	`this.unityContainer = unityContainer;`
	`}`

D.3.3.1 Como carregar uma View a partir de um WorkItem (Controller)?

O método "AddView", definido na classe Controller, permite adicionar uma *View* em *WPF* ou *Windows Forms* a uma região. A região é identificada por um nome, e pode corresponder a uma região "normal" em PRISM, ou a um *Workspace* definido dentro de uma *View* em *Windows Forms*. O tratamento dos quatro casos possíveis é feito de forma transparente.

Tabela D.3.5 - Como migrar a criação e activação de uma *View*

CAB	`IMyView view = this.Items.AddNew<MyView>();` `workspace.Show(view);`
PRISM	`MyView view = this.AddView<MyView>(regionName, regionManager, container, true);` `this.ActivateView<MyView>(regionName, regionManager, container, view);`

D.3.3.2 Como injectar estado?

A classe *Controller* inclui um dicionário que simula o funcionamento da colecção *State* dos *WorkItems*.

Para que seja possível simular a injecção de estado com o atributo *State*, foi criado um atributo *PrismState*. O *PrimState* recebe uma *String* e permite que a propriedade a que o atributo

foi associado seja preenchida com o estado obtido através do *WorkItem* acima na hierarquia, tal como em CAB. Assim, os atributos *State* devem ser substituídos por *PrismState* (a classe deverá ser incluída no projecto de Infra-estrutura) (Tabela D.3.6):

Tabela D.3.6 - Como injectar estado num *Controller* em PRISM

PRISM

```
//[State("mensagem")]
[PrismState("mensagem")]
public string StateMsg
{
    get { return stateMsg; }
    set
    {
        stateMsg = value;
    }
}

public override void Run()
          {

    //...
    InjectState();
}
```

Para que o atributo tenha efeito**,** é necessário chamar a função "InjectState()" no método "Run()" do *Controller*, já que é ela que irá preencher as propriedades marcadas com o atributo *PrismState*. Pressupõe-se que a propriedade *Parent* do *WorkItem* "filho" foi preenchida antes de ser chamado o método "Run()".

Caso seja necessário injectar estado numa *View*, pode ser utilizado o mesmo atributo. No entanto, em vez da função "InjectState()", será necessário chamar a função "InjectMyState()", no construtor da *View* (Tabela D.3.7). Para isso, é necessário obter uma referência para o Controller cujo estado se pretende injectar na *View* (através de *constructor injection*, por exemplo).

Tabela D.3.7 - Como injectar estado numa *View* em PRISM

PRISM

```
public MyView(MyWorkItem workItem)
          {
                    InitializeComponent();
    workItem.InjectMyState(this);
          }
```

Como subscrever o evento *StateChanged*

Se o conceito de *State* não existia em PRISM então é natural que a funcionalidade *StateChanged* também não estivesse disponível. Tal como o próprio nome sugere, o *StateChanged* serve para indicar ao objecto que herda as propriedades, que elas sofreram uma alteração. Para emular essa funcionalidade adicionou-se ao *Controller* de PRISM um *EventHandler*, que é disparado sempre que ocorrem alterações nos estados de um *Controller*. Os

interessados em receber esses avisos têm de subscrever esse evento. Na tabela seguinte (Tabela D.3.8) é possível verificar o procedimento necessário para completar essa acção.

Tabela D.3.8 - Como migrar a subscrição do evento *StateChanged*

CAB	`// Subscribing`
	`[StateChanged("mensagem")]`
	`public void OnMessageChanged(object sender, StateChangedEventArgs e)`
	`{`
	`//...`
	`}`

PRISM	`// Subscribing`
	`WorkItem.StateChanged += new EventHandler<StateChangedEventArgs>(OnMessageChanged);`
	`public void OnMessageChanged(object sender, StateChangedEventArgs e)`
	`{`
	`//...`
	`}`

D.3.4 Commands

Em CAB o conceito de comando está bem delineado e agregado ao *WorkItem*. O *WorkItem* é que disponibiliza todos os mecanismos necessários para funcionar com essa funcionalidade, permitindo adicionar novas instâncias e aceder às já existentes.

Em PRISM esse conceito não existe de raiz, por isso foi necessário arranjar uma forma de emular o conceito. Com esse intuito foi criada uma nova classe, *GlobalCommands*, que irá servir de repositório para os mesmos. Essa classe contém um dicionário que guarda para cada comando a acção correspondente e fornece as funções necessárias para manipular os comandos.

De seguida será apresentado o paralelismo entre as acções efectuadas em CAB e em PRISM.

D.3.4.1 Como registar um handler para um Command?

Tabela D.3.9 - Como registar 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));`

D.3.4.2 Como associar um botão a um Command?

Tabela D.3.10 - Como migrar a associação de um comando a um botão

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

D.3.4.3 Como executar um Command?

Tabela D.3.11 - Como migrar a execução de um comando

CAB	`workItem.Commands["button1Click"].execute();`
PRISM	`GlobalCommands.Commands["button1Click"].Execute(null);`

D.3.5 Views

As *Views* não sofrem grandes alterações com a migração para PRISM, os detalhes que são necessários alterar são os seguintes:

1. Remover o atributo [SmartPart] da classe:

Tabela D.3.12 - Exemplificação da eliminação da propriedade SmartPart da View

PRISM	`//[SmartPart]`
	`public partial class SampleView : UserControl, ISampleView`
	`{ ...`

2. Na propriedade onde é injectado o *Presenter*, o atributo *CreateNew* deve ser alterado para *Dependency*:

Tabela D.3.13 - Exemplificação das alterações necessárias na criação do *Presenter* da *View*

PRISM

```
//[CreateNew]
[Dependency]
public SamplePresenter Presenter
{
    set
    {
        _presenter = value;
        _presenter.View = this;
     }
}
```

No caso de a *View* incluir *Workspaces* (sendo uma *View* em *Windows Forms*) estes terão que ser manualmente registados, para que sejam reconhecidos pela função "AddView" do *Controller*. O registo deve ser feito após a *View* ser carregada, utilizando as funções definidas na classe *Presenter*. Só depois do registo dos *Workspaces* é que é possível adicionar-lhes *Views*. As principais funções para esse efeito são:

- *UpdateListWorkSpaces* – Recebe todos os controlos de uma dada *View* e devolve através do parâmetro "workspaces" a lista de todas as *WorkSpaces* presentes nesses controlos.
- *RegisterWorkSpaces* – está função recebe a lista dos *WorkSpaces* presentes numa *View* e regista-os no *Container*.

Tabela D.3.14 - Como registar as *WorkSpaces* de uma *View* (Forms) em PRISM

PRISM

```
protected override void OnLoad(EventArgs e)
{
    List<IWorkspace> workspaces = new List<IWorkspace>();
    _presenter.UpdateListWorkSpaces(this.Controls, workspaces);
    _presenter.RegisterWorkSpacesOnContainer(workspaces);
    _presenter.OnViewReady();
    base.OnLoad(e);
}
```

No caso de ser uma *view* em WPF, contendo regiões, estas não deverão ser registadas no ficheiro XAML. O registo das regiões deverá passar a ser feito em código:

Tabela D.3.15 - Como registar as *Regions* de uma *View* (WPF)

PRISM

```
IRegionManager regionManager =
ServiceLocator.Current.GetInstance<IRegionManager>();
RegionManager.SetRegionManager(someControl, regionManager);
RegionManager.SetRegionName(someControl, regionName);
```

A obtenção de uma referência para o *RegionManager* também pode ser obtida por *Dependency Injection*.

D.3.6 Presenters

Os *Presenters* devem passar a descender da classe *Presenter*, incluída no projecto de Infra-estrutura. Esta classe inclui um *UnityContainer*, injectado automaticamente, que é exactamente o mesmo que é injectado no *Controller*. Caso seja necessário aceder ao *Controller* para algo mais do que obter o seu *Container*, pode ser injectada uma referência para o *Controller*. A injecção é efectuada de um modo semelhante ao que era feito em CAB, alterando apenas o atributo *ServiceDependency* para *Dependency*:

Tabela D.3.16 - Como injectar o *WorkItem* na *View*

PRISM

```
private SampleWorkItem workItem;

//[ServiceDependency]
[Dependency]
public SampleWorkItem WorkItem {
    get
    {
        return workItem;
    }
    set
    {
        workItem = value;
    }
}
```

D.3.7 Eventos

No caso dos eventos a conversão entre as duas *Frameworks* não é muito linear, existe alguns conceitos em PRISM que não estão presentes em CAB e vice-versa.

- No caso de PRISM é preciso definir uma classe para cada evento que se pretenda utilizar e essa classe tem de derivar do *CompositePresentationEvent*. Em CAB é apenas necessário reservar um nome para o evento, que passa a ser o seu identificador.
- Os eventos são guardados em entidades diferentes nas duas *Frameworks*. No caso de CAB encontram-se no *WorkItem* dentro do *EventTopics*, já em PRISM estão no *eventAggregator*.
- A assinatura dos handlers é diferente nos dois casos. Em CAB o *handler* tem de receber a origem do evento, *sender*, e uma variável do tipo *EventArgs*, por onde pode ser passado algum parâmetro se necessário. No caso de PRISM, a única restrição é no tipo do atributo declarado no *handler* do evento, tem de ser igual ao do tipo definido na classe do evento.

D.3.7.1 Como publicar um evento?

Tabela D.3.17 - Como migrar a publicação de um evento

CAB	`workItem.EventTopics["MyEvent"].Fire(sender, eventArgs, null, PublicationScope.Global);`

PRISM	`// Defining the event`
	`public class MyEvent : CompositePresentationEvent<int>`
	`{`
	`  //...`
	`}`
	
	`// Publishing`
	`int arg = 1;`
	`eventAggregator.GetEvent<MyEvent>().Publish(arg);`

A tabela anterior (Tabela D.3.17) é uma boa ilustração dos dois primeiros pontos referidos no texto de introdução aos eventos.

D.3.7.2 Como subscrever um evento

Tabela D.3.18 – Como migrar a subscrição de um evento

CAB	`// Subscribing`
	`[EventSubscription("MyEvent", Thread = ThreadOption.UserInterface)]`
	`public void OnMyEvent(object sender, DataEventArgs<int> e)`
	`{`
	`  //...`
	`}`

PRISM	`// Subscribing`
	`eventAggregator.GetEvent<MyEvent>().Subscribe(MyHandler, ThreadOption.UIThread);`
	
	`// Defining the handler`
	`public void MyHandler(int arg)`
	`{`
	`  //...`
	`}`

Em CAB para subscrever um evento é apenas necessário colocar o atributo *EventSubscription* antes da função que se pretende instalar. Em PRISM é ligeiramente diferente, em vez de colocar o atributo antes da função é necessário aceder ao *eventAggregator* e associar ao evento a função desejada.

D.3.8 Serviços

Os serviços em CAB e em PRISM seguem a mesma estrutura básica. A forma de os registar é ligeiramente diferente (utilizando um atributo em CAB, e registando-o directamente

num *Container* em PRISM). Além disso, de acordo com a correspondência de conceitos acima, as referências para os serviços deixarão de ser obtidas através do *WorkItem*, passando a ser obtidas através de um *Container*.

D.3.8.1 Como definir um serviço

Tabela D.3.19 - Como migrar a subscrição de um serviço

CAB

```
    public interface IDistanceCalculatorService
    {
        int ComputeDistance(int latitude, int longitude);
    }

    // This is a service that won't be created until needed.
    [Service(typeof(IDistanceCalculatorService), AddOnDemand = true)]
    public class DistanceCalculatorService :IDistanceCalculatorService
    {
        public DistanceCalculatorService()
        {
        }

        public int ComputeDistance(int latitude, int longitude)
        {
            //...
        }
}
```

PRISM

Exactamente igual ao extrato de código anterior, a única diferença é que preciso comentar a seguinte linha:

```
    // [Service(typeof(IDistanceCalculatorService), AddOnDemand = true)]
```

Em substituição do atributo Service, o serviço deve ser registado num container, uma vez que não é associado automaticamente:

```
    IDistanceCalculatorService myService = new DistanceCalculatorService ();
    this.Container.RegisterInstance<IDistanceCalculatorService>(myService);
```

D.3.8.2 Chamar um serviço

Tabela D.3.20 - Como migrar a execução de um serviço

CAB	`private IDistanceCalculatorService myService ;`
	
	`[ServiceDependency]`
	`public IDistanceCalculatorService Service`
	`{`
	`set { myService = value; }`
	`}`
	
	`//OU`
	
	`IDistanceCalculatorService myService =`
	`WorkItem.Services.Get<IDistanceCalculatorService>();`
	
	`myService.ComputeDistance(5, 4);`

PRISM	`IDistanceCalculatorService myService =`
	`Container.Resolve<IDistanceCalculatorService>();`
	`myService.ComputeDistance(5, 4);`

O procedimento de chamar um serviço é bastante similar nas duas *Frameworks*, em ambos os casos é preciso em primeiro lugar obter uma referência para o repositório onde está definido o serviço e só depois é possível invoca-lo. A única diferença entre elas reside no local onde é obtida a referência para o repositório, sendo que em CAB procura-se no *WorkItem* e em PRISM no *Container*.

D.3.9 UIExtensionSites

Para simular a funcionalidade oferecida pelos *UIExtensionSites*, em CAB, foram criadas várias classes e interfaces no projecto de Infra-estrutura, com vista a tornar esse aspecto o mais transparente possível. Seguindo o mesmo paradigma utilizado em CAB, a Shell é responsável por "publicar" quais os elementos que os módulos poderão modificar. Estes, por sua vez, têm acesso a esses elementos para adicionar ou modificar controlos de acordo com as suas necessidades.

D.3.9.1 Como registar UIExtensionSites?

Tabela D.3.21 - Como migrar o registo da Shell nos *UIExtensionSites*

CAB	`RootWorkItem.UIExtensionSites.RegisterSite("SiteName", Shell.MainMenuStrip);`

PRISM

```
Shell.UIExtensionSites.Add("SiteName", Shell.MainMenuStrip);
UIExtensionService.Shell = Shell;


public partial class Shell : Window, IExtensibleShell
{
    //...

    private UIExtensionSitesCollection _extensionSites;
    public UIExtensionSitesCollection UIExtensionSites
    {
        get
        {
            if (_extensionSites == null)
            {
                _extensionSites = new UIExtensionSitesCollection();
            }
            return _extensionSites;
        }
    }

    //...
}
```

Tal como se pode ver no exemplo acima, a Shell deve implementar a interface IExtensibleShell, que obriga a que exista uma propriedade *UIExtensionSites*, onde devem ser registados todos os controlos que a Shell deseja disponibilizar.

Além disso, para que os módulos tenham acesso à Shell e aos controlos que ela publicou, é necessário registar a Shell na classe estática *UIExtensionService*.

D.3.9.2 Como aceder aos UIExtensionSites?

Tabela D.3.22 - Como aceder ao *UIExtensionSites* da *Shell* e adicionar um novo elemento

CAB

```
WorkItem.UIExtensionSites["SiteName"].Add(newButton);
```

PRISM

```
MenuItem mainMenuStrip = UIExtensionService.Shell.UIExtensionSites["SiteName"] as
     MenuItem;
mainMenuStrip.Items.Add(newButton)
```

Na tabela anterior (Tabela D.3.22) é possível verificar como aceder ao *UIExtensionSites* registado pela Shell (com o nome "SiteName"). Uma vez obtido o *UIExtensionSites* é possível adicionar-lhe um novo elemento, neste caso adicionou-se um novo botão ao menu.