Classificação e Comparação de Métodos Ágeis de Desenvolvimento de Software

Mauro Jorge Pereira de Almeida
Mestrado em Informática
Sob orientação do Prof. Doutor João Miguel Fernandes

Universidade do Minho
Departamento de Informática
Braga, 26 de Novembro de 2008

Resumo

A eterna procura pela melhoria da qualidade na concepção de software levou, nos últimos anos, ao aparecimento de inúmeros paradigmas e métodos de desenvolvimento de software. Mais recentemente, com o objectivo de responder positivamente às exigências dos mercados e à crescente complexidade dos projectos de software, surgiram os métodos de desenvolvimento ágeis (MDAs). O elevado número de métodos propostos dificulta a escolha do método a aplicar no contexto de um projecto de software. Por forma a facilitar essa mesma escolha, torna-se necessário realizar um estudo que classifique e compare os diversos métodos de desenvolvimento de software existentes, ágeis ou tradicionais. No entanto, a realização de um estudo comparativo de métodos de desenvolvimento de software revela-se uma tarefa difícil de concretizar, nomeadamente pela dificuldade em encontrar contextos similares de aplicação dos métodos analisados e em proceder a uma validação científica das ideias propostas nesse mesmo estudo.

Este trabalho descreve o estudo comparativo realizado a dois MDAs, o eXtreme Programming (XP) e o Scrum. Na realização deste trabalho e para reduzir as dificuldades inerentes a este tipo de estudo, optou-se por reunir um conjunto de atributos relevantes e utilizá-los como base de comparação para os métodos de desenvolvimento analisados. Os atributos seleccionados consistem num subconjunto de quatro das onze Áreas de Conhecimento (ACs) definidas no SWEBOK e na relação existente entre os princípios apresentados no manifesto ágil e as práticas introduzidas por cada um dos métodos analisados. Através deste conjunto de atributos pretende-se (1) avaliar o grau de abrangência de cada método a um conjunto de áreas de conhecimento que deveriam ser, em teoria, consideradas por todos os métodos de desenvolvimento e (2) avaliar o grau de agilidade de cada método analisado.

O trabalho termina com a apresentação dos resultados obtidos, onde são apontadas algumas diferenças e semelhanças entre os métodos analisados. Através destes resultados é possível concluir que cada método analisado define explicitamente um conjunto de práticas com domínios de aplicação distintos. Por exemplo, as práticas definidas pelo XP são predominantemente orientadas para as fases de implementação e de testes de um projecto de software, enquanto que as práticas definidas pelo Scrum são orientadas para a gestão de projectos. Adicionalmente, é apresentada uma grelha que permite a criação de um relatório com base no estudo efectuado e num conjunto de dados introduzido pelo utilizador.

Abstract

In the last few years, the eternal demand for improving the quality of software development has lead to the appearance of several paradigms and software development methods. Recently, in turn to respond positively to the markets' demands and the growing complexity in software projects, the agile development methods (ADMs) have emerged. The rising number of proposed methods increases the challenge of choosing the method to apply in the context of a software project. In order to ease that challenge, rises the need to conduct a study which classifies and compares the existing development methods, agile or traditional. However, performing such a study reveals difficult to achieve due to the challenge in finding similar application contexts for the analyzed methods and in validating the ideas proposed in that same study.

This work describes a comparative analysis performed on two ADMs, eXtreme Programming (XP) and Scrum. In this work, and with the purpose of reducing the challenges within such study, a set of relevant features have been gathered and each of the methods was compared against it. This set is composed by four of the eleven Knowledge Areas (KAs) defined in SWEBOK and by the existing relation between the principles defined in the agile manifesto and the practices proposed by each of the analyzed methods. Through this selected set of attributes we will (1) asses the coverage degree of each method to a group of knowledge areas which should, in theory, be transversal to all development methods and (2) asses the agility degree for each of the analyzed methods.

The work ends with a presentation of the obtained results, where similarities and differences between both methods are pointed out. Through this results we can conclude that each method explicitly defines a set of practices with distinct application domains. For example, XP is predominantly focused on the implementation and test phases of a software project, while Scrum is focused on projects' management. In addition, a workbook that allows the generation of a report based on the conducted study and on the users' input is presented.

Agradecimentos

Esta dissertação representa o culminar de mais uma fase na minha vida académica e profissional, que nunca teria sido possível sem o apoio e ajuda de algumas pessoas a quem gostaria de agradecer.

Em primeiro lugar gostaria de agradecer ao Professor Doutor João Miguel Fernandes pela confiança depositada em mim, por toda a ajuda proporcionada e pelo rigor e empenho com que supervisionou a realização desta dissertação de mestrado. Espero que a nossa colaboração não termine com a finalização deste trabalho. Agradeço à minha família que sempre me acompanhou e que continuará certamente sempre ao meu lado, apoiando as minhas decisões e ajudando-me a alcançar todos os meus objectivos. Um beijo especial à Joana pela paciência, pelo carinho e pela constante motivação. O último ano foi rico em desafios que conseguimos superar. Sem a tua ajuda não teria conseguido. Obrigado aos meus amigos (vocês sabem quem são) por estarem "simplesmente" ao meu lado. Por último, mas talvez mais importante, ao meu avô Manuel Augusto Pereira que esteve sempre comigo. Obrigado.

Lista de Acrónimos

CMM Capability Maturity Model

CMMI Capability Maturity Model Integration

IEEE Institute of Electrical and Electronics Engineers

ISA International Federation of the National Standardizing Associations

ISO International Organization for Standardization

JTC Joint Technical Committee

KPA Key Process Areas

LASCAD London Ambulance Service Computer Aided Despatch

MDA Método de Desenvolvimento Ágil

NATO North Atlantic Treaty Organization

RAD Rapid Application Development

ROI Return Of Investment

RUP Rational Unified Process

SEI Software Engineering Institute

SWEBOK Software Engineering Body of Knowledge

TI Tecnologias de Informação

TIC Tecnologias de Informação e Comunicação

UML Unified Modeling Language

UNSCC United Nations Standards Coordinating Committee

XP eXtreme Programming

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O desenvolvimento de software remonta à segunda metade do séc. XX, tendo-se tornado nos últimos anos num dos produtos mais importantes da sociedade moderna pela crescente utilização nas mais variadas áreas, desde os simples aparelhos domésticos aos mais complexos sistemas de assistência médica.

Desde o início do desenvolvimento de software, a procura constante da melhoria do nível de serviços prestados e melhoria do produto final tem sido tema de análise por parte da comunidade científica. Assim, diversos métodos, paradigmas, modelos e processos têm sido apresentados com o objectivo de lidar com a enorme complexidade que caracteriza os projectos de software. Alguns desses modelos e métodos, denominados de tradicionais, destacam-se pelo facto de depositarem grande importância na documentação ou pela forma rígida como o processo de desenvolvimento é tratado. Esta abordagem resulta num conjunto de fases que rege, de forma estática, o ciclo de vida do projecto desde a recolha, análise e validação de requisitos à fase de testes. A tendência original nos métodos de desenvolvimento tradicionais (MDTs) era uma abordagem com transições rígidas entre as fases do processo e a impossibilidade no retrocesso a fases anteriores. Mais recentemente têm sido apresentados e adoptados MDTs que incluem desenvolvimento incremental e iterativo. Exemplos destes métodos são o método em cascata, método em espiral e *Rational Unified Process* (RUP)[1].

Em contraste com os MDTs surgem os métodos de desenvolvimento ágeis (MDAs). Os MDAs são formalizados em 2001, com o manifesto ágil (*http://agilemanifesto.org/*), quando dezassete entusiastas desta nova forma de desenvolver software se reúnem para dis-

[1] Apesar de ser geralmente considerado como tradicional, alguns autores consideram o RUP como metamodelo. Outros ainda, incluem o RUP no grupo dos métodos de desenvolvimento ágeis (MDAs) [ASRW02]

cutir esta nova temática. Neste manifesto foram definidos quatro valores que caracterizam a forma como o desenvolvimento de software é abordado na utilização dos MDAs:

Indivíduos e interacções são mais importantes que processos e ferramentas;

Software executável é mais importante que documentação completa e detalhada;

Colaboração do cliente é mais importante do que negociação de contratos;

Respostas rápidas a alterações é mais importante do que seguir o plano inicial.

Adicionalmente, o manifesto ágil propõe um conjunto de doze princípios que suportam os valores apresentados. Estes princípios serão apresentados, em detalhe, ao longo deste documento.

Os MDAs evidenciam-se, relativamente aos MDTs, pela capacidade de se adaptarem às alterações que ocorrem, sucessivamente, no decorrer dos projectos de software. Assim, os MDAS surgem com o objectivo de tornar o desenvolvimento de software mais rápido e eficiente. Entende-se por eficiência no desenvolvimento de software a capacidade de, por um lado, alcançar todos os objectivos de um projecto com um menor orçamento ou com um menor *time to delivery*, ou, por outro lado, de aumentar a qualidade na implementação de um projecto com os mesmos recursos disponíveis.

Ao longo da presente dissertação, os conceitos de "método" e "método de desenvolvimento" serão utilizados frequentemente. Os conceitos de "método", "metodologia" e "modelo" são muitas vezes utilizados erroneamente como sinónimos, na engenharia de software, o que suscita dúvidas e ambiguidade. Com o objectivo de contextualizar o leitor da presente dissertação, define-se *método de desenvolvimento* como sendo o conjunto específico das actividades e acções a realizar no desenvolvimento de um projecto de software. Um método de desenvolvimento define também os intervenientes e a ordem pela qual as actividades e acções especificadas devem ser realizadas.

1.1 Enquadramento

Actualmente as Tecnologias de Informação e Comunicação (TICs) têm vindo a desempenhar um papel fundamental na sociedade e na economia mundial. Representando um negócio na ordem dos 2,15 biliões anuais, cujo maior componente são as Tecnologias de Informação (TI) (41%), seguidas dos equipamentos e hardware (38%) e do software

(21%) [Bei03], a competição, por parte das empresas de TICs, tem aumentado ferozmente.

Contrastando com a importância inerente ao desenvolvimento de software, estudos revelam números preocupantes na qualidade do desenvolvimento de software [Pre01, sta03]. Um dos mais referenciados estudos na área, apresentado pelo Standish Group e denominado de *Chaos Report*, revela que, num só ano, os Estados Unidos gastaram $81 milhões de dólares em projectos cancelados. O estudo revela também que 31% dos projectos analisados foram cancelados antes de serem concluídos e que 51% desses projectos excederam em cerca de 50% o orçamento previsto. No que diz respeito a casos de sucesso, o estudo refere que em grandes empresas somente 9% dos projectos cumpriram o orçamento e requisitos previstos, sendo que para pequenas e médias empresas este número aumenta para 16%.

Embora em grande parte dos casos as falhas e erros no desenvolvimento de software resultem apenas no aumento dos custos e perdas financeiras, existem outros onde erros no software resultam em perdas humanas. Exemplo disso são os casos do Therac-25, uma máquina de terapia radioactiva onde erros de programação deram origem à emissão de doses descontroladas de radiação, ferindo seis pessoas [LT93], e o caso do LASCAD (London Ambulance Service Computer Aided Despatch), que tinha como objectivo a automatização do processo de controlo de ambulâncias em Londres [FD96].

1.2 Motivação

A procura por um método de desenvolvimento capaz de responder às necessidades do mercado de desenvolvimento de software levou a que inúmeros métodos tenham vindo a ser propostos ao longo dos últimos anos. Torna-se assim difícil efectuar uma distinção entre os dois grandes grupos de métodos de desenvolvimento, os tradicionais e os ágeis, e entre cada um dos métodos que integram estes dois grupos.

Apesar desta quantidade mais que suficiente de métodos de desenvolvimento propostos, a procura por um "método ideal" continua sem fim aparente. Isto deve-se ao facto de, como Brooks descreve no seu livro "The mythical man-month" [FPB95], não existir uma *"silver bullet"*. Torna-se então inevitável a necessidade de obtenção do conhecimento, sobre os métodos de desenvolvimento existentes, necessário à escolha do método a aplicar. No entanto essa tarefa não é fácil, devido ao grande número de métodos existentes. Esta dificuldade é acrescida pelo facto de que um dos grandes grupos de métodos de desenvolvimento, os ágeis, é bastante recente. Apesar de serem tema de estudo de vários artigos e livros, o número de estudos objectivos, no que se refere à sua aplicabilidade prática, é

reduzido quando comparado com os estudos referentes aos MDTs. Torna-se assim importante a realização de uma análise comparativa, objectiva e imparcial, entre os MDTs e os MDAs.

1.3 Objectivos

Com esta dissertação pretende-se apresentar um contributo na criação de um guia para a adopção de MDAs em empresas de software, auxiliando na escolha do método ágil que melhor se adequa à empresa ou a determinado tipo de projectos. Desta forma, ao longo da dissertação será apresentado um estudo analítico realizado a um conjunto de MDAs. Este estudo deverá ser objectivo e imparcial, evidenciando algumas das vantagens e desvantagens na aplicação de um ou outro MDA.

Como objectivo desta dissertação pretende-se efectuar uma análise comparativa a um conjunto de MDAs, previamente seleccionados, utilizando como base de comparação um conjunto de atributos. A decisão na escolha dos atributos seleccionados para a comparação prende-se com:

1. A avaliação do grau de abrangência do método a um conjunto de áreas de conhecimento que deveriam ser, em teoria, transversais a todos os métodos de desenvolvimento;
2. A identificação do nível de cobertura dos MDAs analisados aos princípios apresentados no manifesto ágil.

Na análise efectuada foram seleccionados cinco atributos como base de comparação. No sentido de satisfazer o primeiro ponto do objectivo desta comparação, foram seleccionadas quatro das onze Áreas de Conhecimento (ACs) definidas no *Software Engineering Body of Knowledge* (SWEBOK): 1) Requisitos de Software, 2) Construção de Software, 3) Teste do Software e 4) Gestão da Engenharia de Software. Para satisfazer o segundo ponto do objectivo desta comparação, foi seleccionado um quinto atributo com o qual se pretende analisar a relação existente entre os princípios apresentados no manifesto ágil e as práticas introduzidas por cada um dos MDAs analisados.

1.4 Estrutura da dissertação

A presente dissertação é composta por seis capítulos. Inicia-se com este primeiro capítulo onde é efectuada uma pequena introdução e a apresentação dos objectivos e da motivação

que levaram à escrita desta dissertação. Ainda neste capítulo é efectuado um enquadramento e apresentação da estrutura da dissertação.

No capítulo 2, é efectuada uma introdução à Engenharia de Software e são apresentados conceitos intimamente relacionados com esta. Neste capítulo é também introduzido o conceito de métodos de desenvolvimento destacando as duas grandes vertentes dentro destes, os MDTs e os MDAs. Os primeiros, os MDTs, destacam-se pela ênfase e importância dada à documentação e ao plano definido nas fases iniciais do projecto de software. Os segundos, os MDAs, caracterizam-se pela forma inovadora como o desenvolvimento de software é encarado, destacando-se pela capacidade de se adaptarem às alterações existentes nos projectos de software e pelos valores e princípios que defendem. Neste capítulo é também efectuada uma análise aos MDTs, enumerando as suas características, descrevendo as suas limitações e apresentando em detalhe os MDTs mais utilizados nos últimos anos, evidenciando os casos em que será vantajosa a sua aplicação.

O capítulo 3 centra-se nos MDAs, apresentando a sua origem e características. À semelhança dos MDTs, também a aplicabilidade dos MDAs não se estende a todos os tipos de projectos de software. Desta forma, neste capítulo, são apresentadas as limitações inerentes aos MDAs. A adopção de MDAs não é um processo linear e implica grandes alterações na cultura da empresa, sendo que o sucesso desta adopção apenas será possível se existir suporte e aceitação por parte de todas as entidades responsáveis (equipa de desenvolvimento, gestores de projecto e gestores da empresa). Tendo por base este risco na adopção de MDAs em empresas de software, o capítulo 3 enumera também as possíveis dificuldades que podem advir da adopção de MDAs nesse tipo de empresas. No capítulo 3 é também apresentado o manifesto ágil. Este representa um conjunto de valores e princípios, definidos por um grupo de representantes destes novos métodos de desenvolvimento, que servem de pedra basilar e estão presentes, na teoria, em todos os MDAs.

O capítulo 4 serve de introdução aos MDAs que são alvo da análise apresentada nesta dissertação. Para cada um dos MDAs analisados, é apresentada uma pequena explicação da origem do método e serão apresentadas as práticas que cada um deles advoga, o ciclo de vida e as funções e cargos a desempenhar pelos membros da equipa.

Os capítulos 5 e 6 representam a essência da dissertação, apresentando em detalhe a análise comparativa realizada e os resultados obtidos.

Capítulo 2

Engenharia de Software

Nos últimos 20 anos a Engenharia de Software sofreu uma grande evolução, deixando de ser um conceito praticado de forma obscura por um grupo relativamente pequeno de indivíduos, para se tornar um conceito globalmente reconhecido. Actualmente a Engenharia de Software é tema de investigações científicas, debates e estudo. Para o mundo empresarial o simples programador foi substituído pelo engenheiro de software e novos modelos, processos e métodos foram adoptados sucessivamente, resultado da necessidade de melhorar o desenvolvimento de software [Pre01].

2.1 História

A Engenharia de Software é um conceito recente e em constante evolução. A primeira referência ao termo Engenharia de Software surge em 1968 na conferência da North Atlantic Treaty Organization (NATO) em Garmisch, na Alemanha. Esta conferência tinha como objectivo discutir os problemas associados à crescente complexidade dos sistemas computorizados e à consciencialização da incapacidade e limitação humana para lidar com essa mesma complexidade. Mais recentemente, os aspectos humanos da Engenharia de Software e a problemática da comunicação interpessoal em projectos de software têm sido alvo de um crescente interesse por parte da comunidade científica. A consciencialização desta área da Engenharia de Software é visível no livro "The Mythical Man-Month" de Frederick Brooks [FPB95], publicado em 1975 e revisto em 1995.

A definição de Engenharia de Software, apresentada por Fritz Bauer em Garmisch [NR69], explica que

> *"A Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia a fim de obter software, de forma económica, que seja fiável e*

que trabalhe eficientemente em máquinas reais."

Recentemente, Pressman define a Engenharia de Software como [Pre01]:

1. A aplicação de uma abordagem sistemática, disciplinada e quantificável à concepção, operação e manutenção de software;

2. O estudo das abordagens descritas no ponto 1.

Estas múltiplas definições reflectem a dificuldade em especificar univocamente o que é a Engenharia de Software e a sua natureza multi-facetada. Apesar das múltiplas definições para Engenharia de Software apresentadas ao longo dos anos, Presseman destaca que todas elas reforçam a ideia da necessidade de existência de disciplina no desenvolvimento de software e identifica três fases que constituem o processo de desenvolvimento de software [Pre01]:

Fase da Definição: Esta é a fase durante a qual os engenheiros de software identificam qual a informação a ser processada, quais as funcionalidades e níveis de desempenho desejados, qual o comportamento do sistema que é expectável, quais as interfaces a serem implementadas, quais as possíveis restrições e qual o critério de validação que define o sucesso do projecto;

Fase de desenvolvimento: Nesta fase os engenheiros de software definem quais as estruturas de dados a utilizar, qual a forma de implementação e como será a concepção do sistema, traduzido-o numa linguagem de programação. Adicionalmente é definida a forma como será testado o desempenho do sistema;

Fase de manutenção: A fase de manutenção incide sobre as possíveis alterações, correcções e adaptações necessárias ao produto desenvolvido. Estas alterações podem surgir devido a problemas de implementação, alterações nos requisitos do cliente ou melhorias a efectuar ao sistema.

2.2 Análise de requisitos

Sommerville e Sawyer, no livro *"Requirements Engineering: A Good Practice Guide"*, apresentam a seguinte definição para requisitos [SS97]:

> *"Os requisitos são (...) a especificação do que deve ser implementado. Representam a descrição do comportamento do sistema ou uma propriedade ou atributo do mesmo."*

A definição descreve os requisitos como sendo *"o que deve ser implementado"*, isto é, quais os objectivos do sistema e indica que estes representam a descrição do sistema em termos do seu comportamento, propriedades e atributos.

A análise de requisitos é provavelmente o maior desafio num projecto de software. Citando Frederick Brooks, *"a parte mais difícil na implementação de um sistema de software é decidir exactamente o que implementar"* [FPB95]. Descrito como sendo *"as fundações de qualquer projecto de software"* [Wie05], a análise de requisitos representa uma das actividades mais importantes no ciclo de vida de um projecto de software. Requisitos mal definidos ou pouco claros são muitas vezes apontados como um problema comum nos projectos de software. A experiência mostra que erros introduzidos na fase de análise de requisitos são tardiamente detectados conduzindo a um aumento nos custos de correcção, quando comparados com os custos de correcção em fases iniciais do projecto.

2.3 A Engenharia de Software em camadas

Segundo Pressman [Pre01], a Engenharia de Software é um conceito suportado por 4 camadas: 1) qualidade, 2) processos, 3) métodos e 4) ferramentas. Em qualquer Engenharia, nomeadamente na Engenharia de Software, deve existir um compromisso pela procura constante da qualidade.

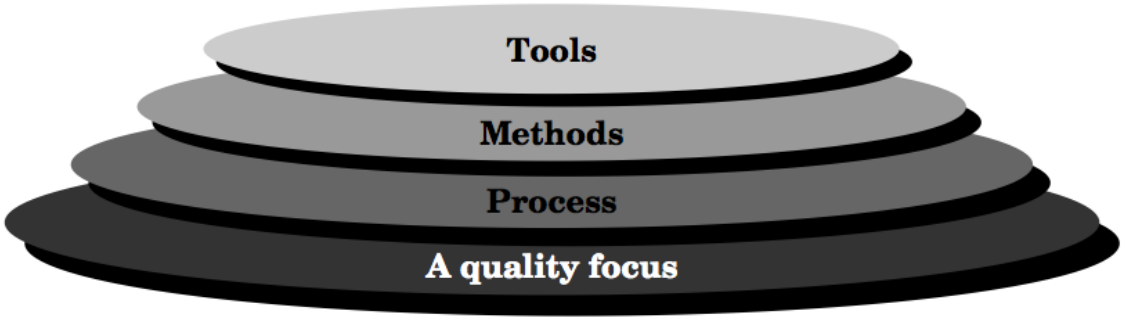

Figura 2.1: Camadas da Engenharia de Software propostas por Pressman [Pre01]

A procura contínua pelo aumento da qualidade e melhoria dos processos utilizados representa a base que suporta a Engenharia de Software. Continuando a sua descrição das camadas que suportam a Engenharia de Software, Pressman define os processos como sendo os alicerces da Engenharia de Software. Um processo define um conjunto de *Key Process areas* (KPAs) que devem ser abordados na aplicação da Engenharia de Software. Estes KPAs representam a base para a gestão de projectos de software e estabelecem o

contexto ao qual os diversos métodos de desenvolvimento são aplicados, a forma como o produto final é produzido e as *milestones*[1] a atingir. Existem vários processos de desenvolvimento que definem abordagens e estratégias distintas, como por exemplo, a norma internacional ISO/IEC 12207, que descreve o método de selecção, implementação e monitorização do ciclo de vida de um projecto de software.

Os métodos representam a camada responsável por introduzir um conjunto de tarefas que dá suporte ao desenvolvimento de software. Essas tarefas incluem análise de requisitos, concepção, desenvolvimento, realização de testes e manutenção. Ao longo dos últimos anos, inúmeros métodos de desenvolvimento têm sido propostos para colmatar o insucesso verificado nos projectos de software, destacando-se duas grandes vertentes - os MDTs e os MDAs. No que diz respeito às ferramentas estas suportam a componente humana existente nos projectos de software, facilitando e por vezes automatizando as tarefas a realizar.

2.4 A qualidade no processo de desenvolvimento

Como foi já referido, a preocupação da Engenharia de Software na melhoria dos processos aplicados tornam a procura pela qualidade na base da Engenharia de Software. Com o objectivo de avaliar quantitativamente e qualitativamente os processos aplicados no desenvolvimento de software, vários modelos e normas têm sido apresentados ao longo dos anos.

2.4.1 CMM e CMMI

Recentemente, a preocupação com a qualidade dos processos aplicados ao desenvolvimento de software tem vindo a aumentar. Apesar dos vários modelos propostos para avaliar a qualidade do processo de desenvolvimento, o CMM (Capability Maturity Model) é o mais conhecido e aplicado nas empresas de software. O facto do CMM resultar de um conjunto de experiências reais, o que faz com que as práticas por ele introduzidas apliquem algo já comprovado, contribui sem dúvida para o seu sucesso e reconhecimento à escala mundial.

Em 1986 Watts Humphrey, do Software Engineering Institute (SEI), e a Mitre Corporation foram encarregues, pelo governo dos EUA, de criar um modelo que permitisse avaliar a capacidade e maturidade dos seus sub-contratados. O resultado foi a criação de uma *framework* de avaliação da maturidade de um processo de software, dando mais

[1] Nome dado a um evento importante no decorrer do projecto. Esse evento é tipicamente a entrega de determinado conjunto de funcionalidades ou o termino de um conjunto de tarefas.

tarde origem ao CMM. Nos anos seguintes este modelo foi continuamente refinado.

Em 1991, o SEI publicou a primeira versão (versão 1.0) do CMM para software (SW-CMM) que descreve os princípios e práticas que definem o grau de maturidade de um processo de software. O SW-CMM está organizado de forma a auxiliar as empresas a elevar o grau de maturidade do seu processo de desenvolvimento, de um nível *ad hoc* e caótico para um nível disciplinado e possível de ser replicado [ACT01]. Ao longo do tempo, novas variantes do CMM têm sido propostos nas mais diversas áreas como gestão de recursos humanos (P-CMM), aquisição de software (SA-CMM) e engenharia de sistemas (SE-CMM). Com a finalidade de integrar os diversos modelos criados e como evolução do CMM, surge em 2002 a primeira versão do Capability Maturity Model Integration (CMMI) que, tendo por base o CMM, define cinco níveis de maturidade para o processo de desenvolvimento numa empresa de software [DEM05]:

Nível 1 (Inicial) - A empresa não possui qualquer tipo de processo formal de desenvolvimento. O processo de desenvolvimento de software é efectuado de forma *ad hoc*;

Nível 2 (Replicável) - O conceito de gestão de projectos, ainda que básico, foi estabelecido. Os custos, âmbito e prazos dos projectos de software são controlados. A principal característica deste nível é a capacidade da empresa em aplicar repetidamente o processo de desenvolvimento, obtendo resultados semelhantes aos de aplicações anteriores;

Nível 3 (Definido) - O processo de desenvolvimento de software encontra-se documentado e integrado com os restantes processos da empresa, tendo sido criados *standards*. Todos os projectos da empresa são desenvolvidos de acordo com o processo da empresa, devidamente actualizado, documentado e formalmente oficializado. Este nível inclui todas as características do nível anterior.

Nível 4 (Gerido) - A empresa recolhe dados qualitativos e quantitativos do processo de desenvolvimento. A recolha destes dados tem como objectivo a monitorização do processo e de possíveis problemas que possam surgir. Estes dados servem também como indicadores da qualidade do processo. Este nível inclui todas as características do nível anterior.

Nível 5 (Optimizado) - A empresa possui um mecanismo que permite a contínua evolução e melhoramento do processo de desenvolvimento. Novas ideias e tecnologias são apli-

cadas ao processo. Os resultados da recolha de dados quantitativos e qualitativos do processo de desenvolvimento são utilizados com o intuito de melhorar o processo.

Associado ao nível de maturidade do processo, o SEI definiu um conjunto de KPAs que descrevem as actividades que devem estar devidamente formalizadas, com o objectivo de se obter um determinado nível de maturidade.

CMM Key Process Areas	
Nível de Maturidade	Key Process Areas
Nível 2	Gestão da configuração do software; Garantia de qualidade; Gestão dos sub-contratados; Acompanhamento e supervisão do projecto de software; Planificação do projecto de software; Gestão de requisitos.
Nível 3	Coordenação inter-grupos; Gestão integrada do software; Programas de treino; Definição dos processos da organização; Focos no processo da organização; Peer-reviewing; Engenharia do produto de software.
Nível 4	Gestão quantitativa dos processos; Gestão da qualidade do software.
Nível 5	Gestão das alterações no processo; Gestão de alterações tecnológicas; Prevenção de defeitos.

Tabela 2.1: KPAs associados a cada nível de maturidade do processo.

Apesar dos sucessos documentados [HCR+94, HG96], o CMM e CMMI não são isentos de duras críticas por parte de alguns membros da comunidade científica [Bac94].

2.4.2 Normas

As normas internacionais de qualidade são criadas com base no trabalho de especialistas de todo o mundo com o objectivo de uniformizar as especificações dos produtos, interfaces, processos e terminologias. As normas são actualmente utilizadas como forma de introduzir um conjunto de "boas práticas" nos processos das empresas, com o objectivo de alcançar assim um maior nível de qualidade. Mais ainda, estas normas são também utilizadas como base de comparação através da qual as empresas, organizações ou produtos de software são certificados [Ema97]:

- Terminologias;

- Procedimentos;
- Modelos;
- *Benchmarks*.

Existem diversos tipos de normas, no entanto as normas internacionais têm vindo a ganhar um enorme reconhecimento a nível mundial, das quais se destacam as normas emitidas pela International Organization for Standardization (ISO).

International Organization for Standardization - ISO e ISO/IEC

Em 1906 é fundada a International Electrotechnical Commission (IEC) e em 1947 a ISO. A ISO é fundada a partir de organizações já existentes: a ISA (International Federation of the National Standardizing Associations) e a UNSCC (United Nations Standards Coordinating Committee). Embora ambas as organizações se destinassem à criação de normas que facilitassem o comércio e trocas internacionais de bens e serviços, a IEC concentrava-se principalmente na criação de normas no campo da Engenharia Electrotécnica.

Em 1987, a ISO e a IEC decidiram estabelecer o Joint Technical Committee (JTC) com o objectivo de criar normas específicas para a área das TICs, que conta actualmente com 19 sub-comités activos. Em Junho de 1989, o JTC iniciou a criação da norma internacional ISO/IEC 12207 [ISO02] com o objectivo de normalizar o ciclo de vida de um processo de software, vindo assim a colmatar um problema há muito existente.

ISO/IEC 12207

Os modelos que definem a maturidade de um processo, como o CMM e CMMI, identificam um conjunto de KPAs que devem estar presentes no sentido de se atingir um determinado nível de maturidade. Definem assim objectivos claros a serem alcançados na execução de um processo. Por sua vez, as normas ISO, em particular a norma ISO/IEC 12207, oferecem uma perspectiva diferente. Em vez de definir um conjunto de objectivos, nível de maturidade organizacional ou de capacidade do processo, a norma ISO/IEC 12207 propõe um conjunto de actividades necessárias para o seu desenvolvimento e manutenção. A norma ISO/IEC 12207 abrange assim todo o ciclo de vida de um processo de software, desde a sua concepção e análise de requisitos, até à manutenção e finalização. Esta norma cobre também a garantia de qualidade, desde produtos adquiridos ou fornecidos, até à qualidade e melhoria dos processos de implementação [EFF+99].

2.5 Gestão de projectos

O conceito de projecto não é algo recente. Ao longo da história da Humanidade é possível observar vários projectos bem sucedidos, resultado do trabalho organizado de um conjunto de indivíduos. As Pirâmides do Egipto ou a Grande Muralha da China são exemplo de importantes projectos históricos [Jur99]. Citando Jason Westland, *"um projecto é a tentativa, única, de produzir um conjunto de resultados respeitando prazos, orçamentos e níveis de qualidade especificados"* [Wes06]. Apesar da heterogeneidade que se verifica nos projectos, no que diz respeito à dimensão, prazos ou orçamento, é possível identificar algumas características comuns a todos os projectos:

- Cada projecto especifica um ou mais objectivos;
- Cada projecto deve ser completado num período de tempo pré-determinado;
- Cada projecto tem um orçamento associado;
- Cada projecto é executado por uma ou mais equipas;
- Cada projecto é único. Não existe por isso uma solução aplicável a todos os tipos de projecto. Citando Brooks, *"Não existe uma silver bullet"* [FPB95].

No seu livro [Wes06], Jason Westland define um ciclo de vida de qualquer projecto que consiste em quatro fases 1) Iniciação, 2) Planificação, 3) Execução e 4) Finalização.

Iniciação

A iniciação do projecto é a primeira fase do seu ciclo de vida. Durante esta fase as oportunidades de negócio e possíveis problemas devem ser identificados. Deve ser também definido um caso de estudo de alto nível, identificando as várias soluções para o desenvolvimento do projecto. Nesta fase deve ser conduzido um estudo no sentido de identificar se cada uma das soluções apresentadas aborda os problemas e as oportunidades de negócio identificadas. Após aprovação de uma das soluções recomendadas são definidos os objectivos e âmbito.

Planificação

Após a definição dos objectivos e âmbito do projecto, este entra numa fase de planificação detalhada. Esta fase inclui:

- Identificação das principais actividades, tarefas, dependências e prazos;

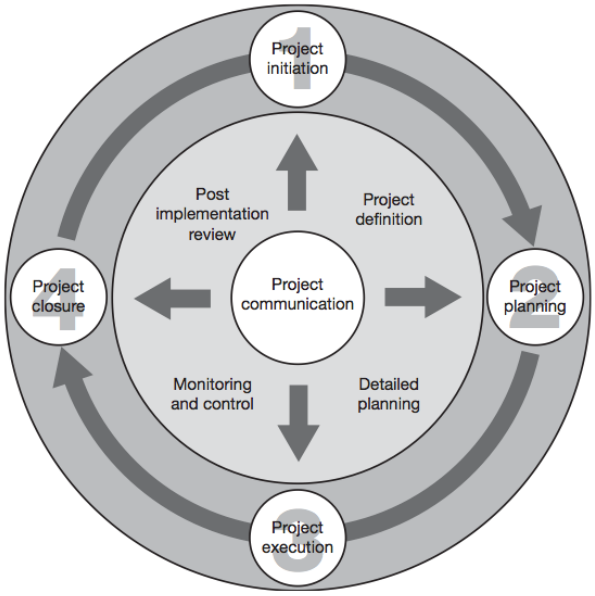

Figura 2.2: Ciclo de vida de um projecto [Wes06]

- Identificação dos recursos necessários (material, equipamento e recursos-humanos);
- Identificação dos custos associados aos recursos necessários;
- Identificação das métricas que asseguram a qualidade do produto final e de que forma estas serão medidas;
- Identificação dos possíveis riscos associados ao projecto e das acções necessárias para colmatar esses riscos;
- Identificação dos pontos de aceitação a atingir de forma a que o produto final seja aceite pelo cliente;
- Identificação dos mecanismos de comunicação e informação necessária a transmitir aos intervenientes;
- Identificação da necessidade de recursos externos ao projecto.

Da concretização desta fase resulta um plano detalhado e o projecto encontra-se pronto para execução.

Execução

A fase de execução diz respeito à implementação dos planos criados na fase anterior. Cada plano é executado e é efectuada a gestão e controlo de cada tarefa ou actividade executada. A gestão de cada actividade é da responsabilidade do gestor do projecto, que deve acompanhar as possíveis alterações ao plano, riscos, níveis de qualidade e critérios de aceitação. Após a fase de execução o produto final é entregue ao cliente. Caso os critérios de aceitação tenham sido cumpridos, o projecto está pronto para a sua finalização.

Finalização

A finalização de um projecto inclui a entrega do produto final ao cliente, juntamente com a documentação acordada, o término dos contratos, a libertação dos recursos envolvidos no projecto e a comunicação do fim do projecto a todos os intervenientes. Uma prática comum é a realização de uma avaliação para quantificar o nível de sucesso do projecto e da gestão do mesmo, identificando os problemas que ocorreram durante a implementação e de que forma estes foram ultrapassados, no sentido de melhorar projectos futuros.

Recentemente a gestão de projectos e do ciclo de vida associado emerge, como actividade fundamental em qualquer projecto, da necessidade de lidar com uma maior complexidade e maior número de entidades envolvidas. A gestão é uma das actividades mais importantes de um projecto e deve ser transversal a todas as fases deste. Sem uma gestão correcta do projecto as probabilidades deste falhar aumentam consideravelmente. Algumas das tarefas incluídas na gestão de projectos são apresentadas de seguida:

- Gestão de requisitos;
- Monitorização e planificação de processos;
- Definição de tarefas e estimativa de esforço associado;
- Gestão da qualidade;
- Avaliação e controlo de riscos;
- Gestão da equipa do projecto;
- Comunicação com o cliente;
- Gestão de configurações;
- Revisões;
- Análise e definição de *milestones*;

- Prevenção de problemas.

Actualmente os projectos de software representam uma fracção considerável da economia mundial. Cerca de 1 milhão de projectos são implementados anualmente [Jal02]. No entanto grande parte destes projectos de software falham devido ao incumprimento dos requisitos, do orçamento ou dos prazos de entrega. Análises recentes indicam que um terço dos projectos de software apresentam custos 125% superiores ao previsto [Jal02]. Tendo em consideração os valores anteriores é possível afirmar que os projectos de software têm muito a beneficiar com uma gestão eficiente.

Os projectos de software são, cada vez mais, desenvolvidos em ambientes instáveis e sujeitos a constantes alterações. A incerteza perante as necessidades "reais" dos utilizadores finais dos produtos de software leva à alteração dos requisitos durante o ciclo de vida de um projecto. As dificuldades encontradas na gestão de projectos de software, quando comparados com outros tipos de projectos, pode ser explicada pelas características do produto resultante. O software, como produto resultante dos projectos de software, destaca-se pela sua:

Intangibilidade. Ao contrário do *hardware* e de outro tipo de produtos, o software é intangível. Como resultado, torna-se difícil medir o progresso e qualidade do software desenvolvido e acompanhar o seu progresso no decorrer do projecto;

Complexidade. A crescente complexidade nos produtos de software desenvolvidos aumenta a dificuldade em compreender o produto não só do ponto de vista técnico, mas também do ponto de vista de gestão do projecto;

Volatilidade dos requisitos. A pressão por parte do mercado de software e a redução no *time-to-market* conduzem a constantes alterações e elevado grau de incerteza na definição de requisitos.

De seguida são apresentadas algumas das principais razões que levam ao insucesso verificado nos projectos de software:

1. Requisitos mal definidos ou pouco claros;
2. Prazos e orçamentos irrealistas;
3. Falta de métodos ou processos de gestão;
4. Utilização de novas tecnologias;

5. Problemas de comunicação inter/intra equipa(s) do projecto.

Dos motivos apresentados, os três primeiros (1 a 3) dizem respeito à gestão de projectos. No que se refere à utilização de novas tecnologias e problemas de comunicação (4 e 5), embora não estejam directamente relacionados com a gestão de projectos, dizem respeito à gestão de riscos de um projecto, sendo esta da responsabilidade da gestão de projectos. A forma como os principais motivos pelos quais os projectos de software falham caiem, directa ou indirectamente, sobre o domínio da gestão de projectos vem realçar a importância desta para o sucesso de um projecto. Assim, é possível afirmar que grande parte dos projectos de software falha devido à inexistência de uma gestão de projectos eficiente.

No entanto a gestão de um projecto não depende exclusivamente da capacidade da pessoa responsável por o gerir. É também necessária a existência de um método de desenvolvimento adequado para dar suporte à gestão do projecto. Sem um método ou processo de desenvolvimento a equipa do projecto não possui nenhum tipo de estrutura capaz de orientar as actividades do projecto, aumentando assim a dificuldade de gerir o tempo, o custo, os riscos e as alterações de requisitos num projecto de software.

2.6 Métodos de desenvolvimento

Estudos recentes [sta03] mostram taxas alarmantes de insucesso nos projectos de software. Cerca de dois terços dos projectos de software falham devido ao incumprimento total ou parcial dos requisitos pré-definidos ou, por outro lado, devido ao seu incumprimento relativamente ao prazo previsto. Os mesmos estudos apontam que 23% dos projectos de software são cancelados ou não implementados. Para estas taxas de insucesso contribuem muitas vezes a adopção de abordagens pouco sistemáticas e disciplinadas. Torna-se assim evidente a necessidade da adopção de um método de desenvolvimento de software adequado.

Uma abordagem sem um método de desenvolvimento definido acarreta pelo menos os três seguintes problemas. Primeiro, após um certo número de iterações entre codificação e resolução de problemas, o código daí resultante possui pouca ou nenhuma estrutura, tornando as alterações ao mesmo bastante difíceis. O segundo problema, consequência do primeiro, é o elevado custo das alterações e testes efectuados ao código implementado. O terceiro problema prende-se com o incumprimento das necessidades do cliente e requisitos estipulados.

Ao longo dos últimos anos, inúmeros métodos ou processos de desenvolvimento têm sido propostos para colmatar o insucesso verificado nos projectos de software. Os chamados MDTs têm sido largamente utilizados pelas equipas de desenvolvimento, no entanto a adopção deste tipo de desenvolvimento é acompanhada por um conjunto de problemas igualmente difíceis de resolver. Os MDTs caracterizam-se por possuírem uma abordagem bastante rígida e disciplinada através de um conjunto de fases. Na prática, esta abordagem inflexível assume que todos os requisitos são conhecidos no início de um projecto e inalteráveis durante o decorrer do mesmo - a experiência mostra que tal não é verdade.

Com o objectivo de tornar o desenvolvimento de software mais rápido e permitir uma implementação capaz de responder às constantes alterações no decorrer de um projecto surgiram os MDAs, que resultam da percepção de que a inflexibilidade e a incapacidade de adaptação a alterações nos requisitos dos projectos, característica dos MDTs, dificultam a sua implementação com êxito. No entanto, os MDAs não demonstram ser a tão procurada *"silver bullet"* e, embora sejam descritos inúmeros casos de sucesso [SA05, Gre01, Bar06], a sua adopção acarreta também dificuldades e problemas. Outro ponto a ter em consideração, aquando da adopção dos MDAs, está relacionado com a sua aplicabilidade e adaptabilidade apenas a um conjunto tipificado de projectos. Embora seja motivo de estudo a sua aplicação a projectos de grande dimensão, os MDAs são direccionados para projectos geralmente de pequena dimensão. Esta observação é também constatada por *Cockburn* ao salientar que *"nenhum método é passível de ser aplicado a todos os projectos"* [Coc02].

O aparecimento dos MDAs levou à divisão dos programadores em duas classes distintas:

Os Tradicionalistas defendem que os MDTs proporcionam mais vantagens que os ágeis, apontando estes últimos como sendo uma técnica de *code-and-fix*;

Os Agilistas defendem que os MDAs são capazes de melhorar o desenvolvimento de software, respondendo melhor e mais facilmente a alterações de requisitos e funcionalidades, reduzindo o número de erros existentes nos produtos desenvolvidos.

No meio existem ainda aqueles que afirmam que cada um dos tipos de métodos possuem as suas vantagens e desvantagens e que cada um deles é apropriado para tipos específicos de projectos [Boe81, Boe02].

2.7 Métodos de desenvolvimento tradicionais

No final de 1960, o NCC *(National Computing Center)* apresentou a sua definição para o ciclo de vida de um projecto ou definição de um sistema. Este método deu mais tarde origem a um dos mais utilizados MDTs - o método em cascata *(waterfall)* [Roy70]. O método em cascata apresenta uma estrutura sistemática e sequencial para o desenvolvimento de software, definindo um conjunto de actividades a executar em cada uma das fases propostas. Apesar de ser vastamente utilizado, nem que seja ao nível das intenções, o método em cascata apresenta algumas limitações no que diz respeito ao "congelar" dos requisitos numa fase inicial do projecto e à transição rígida existente entre cada fase. A impossibilidade da transição para fases anteriores, característica do método em cascata, deu origem ao seu nome - *waterfall*. Em 1988 Boehm apresentou outro método de referência no grupo dos MDTs - o método em espiral [Boe88].

De 1980 a 1990, vários métodos de desenvolvimento foram propostos e publicados surgindo conceitos como prototipagem, *Rapid Application Development* (RAD), desenvolvimento iterativo e incremental e desenvolvimento orientado aos objectos. Estes conceitos deram mais tarde origem aos chamados MDAs.

Os MDTs foram os primeiros métodos de desenvolvimento a surgir. Estes vieram colmatar a lacuna existente no desenvolvimento de software, no que se refere à necessidade de identificar e descrever formalmente um conjunto de actividades que facilitasse o desenvolvimento e gestão de projectos de software. Até ao aparecimento dos MDTs, o desenvolvimento de software era realizado de forma *ad hoc*, sem método e seguindo uma abordagem de *code-and-fix*. Como referido anteriormente na página 18, deste tipo de abordagem advêm alguns problemas que dificultam uma implementação bem sucedida dos projectos de software.

Os MDTs foram a primeira resposta ao problema apresentado e baseiam-se num conjunto sequencial de tarefas, ou actividades, tais como definição de requisitos, implementação da solução, realização de testes e integração ou *deployment*.

2.7.1 Características

Apesar do grande número de métodos propostos, os MDTs partilham um conjunto de aspectos que os caracterizam. Essas características incluem a importância dada à análise de requisitos, ao uso e estruturação de documentação e à decomposição de todo o ciclo de

vida num conjunto de fases estáticas e bem delineadas - práticas derivadas do método em cascata. Os MDTs partilham, de forma implícita ou explícita, algumas das características do método em cascata. Evidência deste facto é a divisão do processo de desenvolvimento em fases como análise de requisitos, concepção da solução, implementação e manutenção. Mais recentemente têm surgido versões incrementais e evolutivas dos MDTs.

Os MDTs fazem uso de grande quantidade de documentação e artefactos conceptuais, tais como diagramas, no sentido de planear o desenvolvimento, monitorizar o progresso e garantir os níveis de qualidade. Esta utilização de grandes quantidades de informação documentada permite um melhor controlo de todo o processo de desenvolvimento, principalmente em situações onde o tamanho da equipa do projecto é composta por vários elementos, ou em situações em que o nível de maturidade e experiência dos elementos da equipa é reduzido. Outra característica que destaca os MDTs é o facto de estes se basearem no pressuposto de que existe, no início do projecto, um problema bem definido e que todos os requisitos do sistema são conhecidos.

A grande vantagem dos MDTs, defendida pelos tradicionalistas, reside na capacidade de replicar a sua aplicação uma vez que todas as fases do processo se encontram documentadas, assim como os resultados das suas aplicações. No entanto é necessário relembrar que *"nenhum método é passível de ser aplicado a todos os projectos"* [Coc02] e que *"não existe uma silver bullet"* [FPB95].

2.7.2 Limitações

As características dos MDTs levantam algumas limitações na sua aplicabilidade. Uma das principais características dos MDTs que tem sido referida ao longo deste capítulo é o recurso excessivo à documentação. Em alguns casos esta característica pode ser considerada uma mais valia, nomeadamente no que se refere ao controlo das fases do projecto e integração de novos elementos nas equipas dos projecto. No entanto alguns problemas advêm da utilização excessiva de documentação.

Uma das grandes diferenças entre os MDTs e os MDAs diz respeito à capacidade em se adaptarem às constantes alterações existentes no decorrer de um projecto, o que muitas vezes determina o sucesso ou insucesso de um projecto. Os MDTs caracterizam-se por, na prática, efectuarem uma análise detalhada dos requisitos no início do projecto, reduzindo assim a possibilidade de alteração dos mesmos. A constante alteração no que é hoje uma oportunidade de negócio, economicamente viável, e o ambiente volátil em que os projec-

tos de software são desenvolvidos tornam difícil a utilização de métodos de previsão ou a reunião de um conjunto estável de requisitos. Esta característica é assim apontada como a maior limitação dos MDTs. Assim, de forma eventualmente tendenciosa, Martin Fowler e Jim Highsmith, fundadores do manifesto ágil, referem que *"é mais produtivo promover a adaptação à mudança que tentar prevê-la"* [FH01].

As constantes alterações no decorrer do projecto levantam outro problema, ou limitação, na aplicação dos MDTs. A necessidade e pressuposto dos MDTs em definir todos os requisitos do projecto no início do mesmo leva a que o resultado final não corresponda, na maior parte das vezes, às verdadeiras necessidades do cliente. Estudos mostram que, em média, 45% das funcionalidades implementadas nos projectos de software não têm qualquer valor de negócio ou utilidade para o cliente, não sendo por isso utilizadas [sta02]. Este é o motivo que leva os "agilistas" a defenderem que o concepção do sistema deva ser o mais simples possível, com o objectivo de não adicionar complexidade desnecessária, e que se baseia num desenvolvimento incremental e iterativo acompanhado activamente pelo cliente.

2.7.3 Método em cascata - *Waterfall*

O método em cascata, também conhecido por método sequencial, foi o primeiro MDT publicado e sugere uma abordagem linear e sequencial[2]. Assim, o processo de desenvolvimento é visto como um conjunto de fases sequenciais desde a análise de requisitos até à fase de suporte. Embora criticado pela forma desactualizada como este método encara o desenvolvimento de software, este método continua a ser bastante utilizado. A figura 2.3 ilustra as diferentes fases do método em cascata.

A principal característica do método em cascata é o facto de não existir sobreposição das fases ou retrocesso para fases anteriores, o que dificulta a correcção de erros em fases anteriores. As características sequenciais e lineares do método não permitem o início de uma fase sem que a anterior tenha terminado e sido revista. Ainda assim, estas características, permitem um maior controlo de todo o projecto. Como MDT, o método em cascata coloca também grande ênfase na documentação. Assim, o final de cada uma das fases tem como resultado um conjunto de documentos que servem normalmente de base para a fase seguinte.

[2]Embora o método em cascata original proposto por Winston Royce [Roy70] suportasse ciclos, na prática as empresas aplicam o método de forma sequencial e linear.

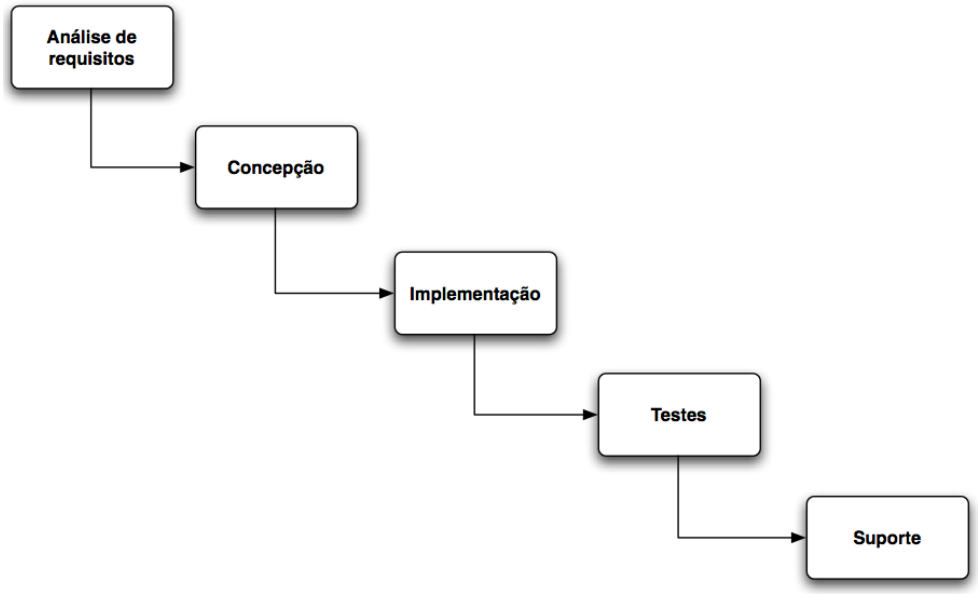

Figura 2.3: Método em cascata

O método em cascata é composto por cinco fases: 1) análise de requisitos, 2) concepção, 3) implementação, 4) testes e 5) suporte, que serão descritas de seguida [Pre01].

Análise de requisitos

Na fase de análise de requisitos são reunidas todas as funcionalidades a implementar, especificadas pelo cliente. Nesta fase é dada grande importância à interacção com o cliente, devendo por isso o analista dominar a área na qual será desenvolvido o projecto. O analista deverá ter também em consideração todos os aspectos de aceitação por parte do cliente, nomeadamente questões de funcionalidade, *performance* e interface com o utilizador final. No final desta fase é elaborado um documento contendo formalmente todos os requisitos definidos e acordados com o cliente.

Concepção

A fase de concepção deve-se focar em quatro atributos distintos do produto de software: 1) estrutura de dados, 2) arquitectura do software, 3) interface e interacções com o utilizador final e 4) algoritmos utilizados. Nesta fase, os requisitos definidos na fase anterior são traduzidos numa representação, como por exemplo o *Unified Modeling Language* (UML), cuja qualidade seja possível de avaliar antes de se dar início à implementação. Tal como os requisitos, também a concepção e arquitectura do sistema são documentados.

Implementação

Nesta fase dá-se início a implementação do produto de software. Assim, o documento de análise de requisitos e o documento que especifica a concepção da arquitectura a implementar, elaborados nas fases anteriores, são traduzidos numa linguagem de programação.

Testes

Após a implementação do produto do software dá-se início à fase de testes. Esta tem por objectivo verificar que todas as funcionalidades e requisitos especificados pelo cliente se encontram implementados, assegurando os níveis de qualidade e interfaces de interacção acordados com o cliente.

Suporte

Após a finalização dos testes o produto encontra-se pronto para entrega ao cliente. No entanto, a experiência mostra que na maior parte dos casos o produto entregue é sujeito a alterações devido a erros, alteração dos requisitos ou requisitos que foram mal interpretados e por isso incorrectamente implementados. Adicionalmente poderá ser necessário efectuar alterações para suportar interacção com novos sistemas externos ou sistemas operativos. A manutenção ou suporte de software aplica cada uma das fases anteriores novamente, desta vez a um produto existente e não a um novo produto.

Apesar de ser bastante utilizado nos projectos de software, o método em cascata tem sido alvo de algumas críticas intimamente relacionadas com as suas características tradicionais:

- O facto de ser um método sequencial e linear, não havendo por isso sobreposição de tarefas, pode criar algum atraso no desenvolvimento. Também o tempo gasto nas fases iniciais é considerável e atrasa o início do desenvolvimento, criando assim alguma frustração nos programadores;
- A inflexibilidade inerente a este método dificulta a alteração dos requisitos em fases intermédias do projecto, impossibilitando assim a adaptação à mudança;
- O contacto tardio que o cliente tem com o produto final. O cliente apenas tem contacto com o produto implementado no final do projecto. Dada a dificuldade em especificar totalmente os requisitos no início de um projecto, o resultado do mesmo não corresponde muitas vezes ao pretendido pelo cliente.

Com o objectivo de colmatar as críticas apontadas ao método em cascata, variações deste têm vindo a ser apresentadas [McC96]. No entanto, o método em cascata original continua a ser o mais utilizado nas empresas de software.

2.7.4 Método em espiral - *Spiral*

O método em espiral foi proposto por Boehm [Boe88] em 1988. Este método, inovador para a época, incorpora conceitos do ciclo de vida clássico de um projecto, prototipagem e desenvolvimento incremental. Adicionalmente este método inclui um conceito importante, a análise de risco. Apesar de importante, este conceito não é contemplado no método em cascata. A figura 2.4 representa o método proposto por Boehm.

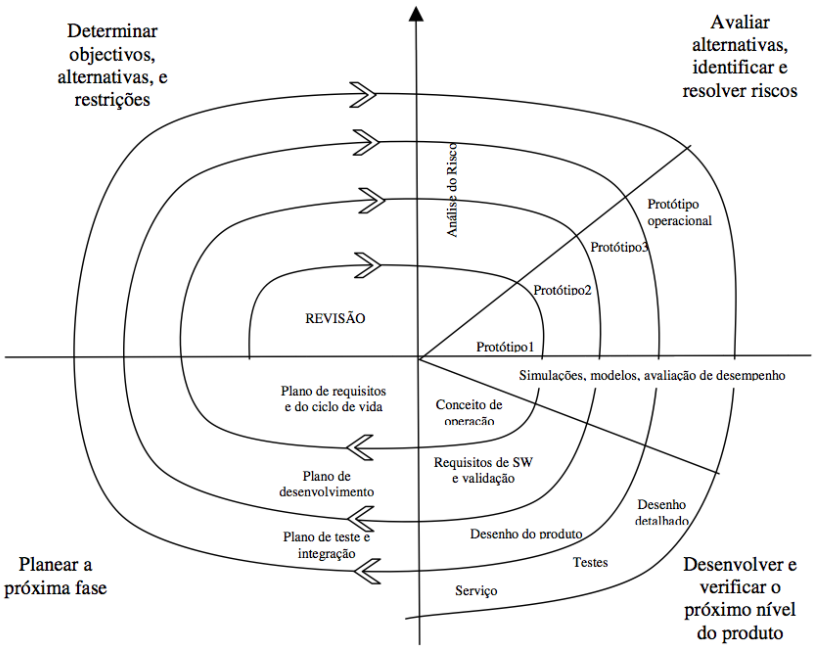

Figura 2.4: Método em espiral [Dua02]

A espiral que descreve o método proposto por Boehm é dividida em quatro partes fundamentais: 1) definição e determinação dos objectivos, 2) avaliação e redução dos riscos, 3) desenvolvimento e validação e 4) planificação [Boe88].

1. **Definição e determinação dos objectivos** - Nesta fase são identificados os objectivos para cada uma das fases do projecto;

2. **Avaliação e redução dos riscos** - Nesta fase são identificados os principais riscos

existentes. Esses riscos são analisados e são identificadas formas e obtida informação que possibilite a redução desses riscos;

3. **Desenvolvimento e validação** - Implementação do protótipo ao qual são adicionadas novas funcionalidades a cada ciclo do projecto. No final do projecto, o resultado será um protótipo operacional;

4. **Planificação** - Revisão do estado actual do projecto e planificação do próximo ciclo da espiral.

Segundo Boehm [Boe88], cada ciclo da espiral começa com a identificação dos seguintes pontos:

- Objectivos para o ciclo actual;
- Alternativas de implementação para o protótipo a desenvolver nesse ciclo;
- Riscos e problemas que advêm dessas alternativas.

Resumindo, o objectivo é identificar a cada ciclo da espiral e de forma incremental os riscos e objectivos/requisitos do projecto.

O método proposto por Boehm não é infalível já que este depende da capacidade dos programadores em identificar correctamente os riscos, bem como da sua aplicabilidade a determinado tipo de projectos. O método em espiral possui maior vantagem quando aplicado a projectos de grandes dimensões e maior complexidade, onde os objectivos são incertos ou encontram-se confusos. Utilizando o método em espiral é possível entregar ao cliente um conjunto de protótipos ao longo do projecto até chegar ao produto final.

Capítulo 3

Métodos de Desenvolvimento Ágeis

Como referido nos capítulos anteriores, ao longo dos últimos anos a Engenharia de Software tem vindo a evoluir sucessivamente. A evolução rápida e o aparecimento de novas tecnologias, o aumento da complexidade dos projectos de software, a incerteza na definição dos requisitos e a redução no *time-to-market* dos produtos são factores que têm contribuído para o insucesso verificado em alguns projectos de software [Jal02, sta03]. Face a estas novas dificuldades e com o objectivo de responder de forma eficaz às constantes alterações nos projectos de software surgiram os MDAs. Estes novos métodos de desenvolvimento destacam-se pela forma inovadora como abordam o desenvolvimento do software e pela ênfase e importância que colocam no factor Humano, colocando as pessoas antes dos processos.

Um estudo recente indica que a utilização de MDAs permite um aumento de produtividade na ordem dos 20%, uma redução de 5-7% nos custos de produção e uma redução de 25-50% no tempo de finalização dos projectos [Rei02]. Estes dados reflectem uma análise efectuada a um conjunto de empresas com níveis de CMMI distintos. Os projectos analisados dizem respeito a projectos de investimento/piloto relativamente pequenos, com equipas de dez ou menos elementos, duração inferior a um ano e orçamento relativamente reduzido. Estes projectos eram maioritariamente aplicações com um *time-to-market* reduzido. No que diz respeito à equipa de desenvolvimento envolvida nos projectos, esta era composta por elementos experientes mas relativamente jovens, factor que talvez tenha influenciado a motivação e a fácil adaptação a novos métodos de desenvolvimento.

Neste capítulo será efectuada uma introdução aos MDAs apresentando as suas ca-

racterísticas e limitações no que se refere à aplicabilidade a determinados tipos de projecto. Adicionalmente serão apresentadas as dificuldades que podem surgir na adopção destes novos métodos de desenvolvimento em empresas de software. Neste capítulo será também apresentado o manifesto ágil, o documento que serviu para formalizar este grupo de métodos de desenvolvimento, e os valores e princípios que o constituem. A apresentação dos valores e princípios que integram o manifesto ágil servem de base para a análise efectuada no capítulo 4, com a qual se pretende identificar "se" e "de que forma" os doze princípios definidos no manifesto ágil se encontram presentes num grupo de MDAs previamente seleccionados.

3.1 História

O conceito de MDAs surge em consequência do aumento da complexidade dos projectos de software, da dificuldade na definição dos requisitos e da constante pressão dos prazos pela entrega de um produto funcional e lucrativo. Em 1990 um grupo de profissionais da indústria de software concluiu que a definição de requisitos em fases iniciais de um projecto de software e o consequente "congelamento" dos mesmos não constituía a melhor abordagem para desenvolver software. Como resultado dá-se início à criação de um conjunto de novos métodos de desenvolvimento tendo por base o desenvolvimento incremental, uma técnica introduzida em 1975 [BT75], sendo mais tarde denominados como métodos de desenvolvimento ágeis (MDAs). Em 2001, dezassete entusiastas destes novos métodos de desenvolvimento reuniram-se para discutir esta temática. Como resultado surge o manifesto ágil que é uma compilação de valores que podem ser observados na maioria dos MDAs. Adicionalmente o manifesto ágil propõe um conjunto de doze princípios que suportam os valores apresentados. Estes princípios são, citando Cockburn, *'...resultado da experiência directa e reflexão sobre essa mesma experiência'* [Coc02].

Os MDAs não são um "anti-método" ou ausência de método. De facto, os defensores dos MDA pretendem

> *(...) restaurar a credibilidade da palavra (método) (...)*

e propõem-se a fazer uso

> *(...) da modelação, mas não meramente para arquivar os diagramas num repositório de uma empresa, (...) da documentação, mas sem gastar grandes quantidades de papel em documentação à qual não se dá qualquer suporte, (...) e planificação, reconhecendo no entanto os limites de a efectuar em ambientes turbulentos* [FH01] .

3.2 Características

São vários os artigos onde as características dos MDAs são analisadas e discutidas [Mil01, AWSR03, Mil03]. As características que serão seguidamente apresentadas resultam da análise desses mesmos artigos e da selecção das características que, na opinião do autor desta dissertação, são aquelas que melhor definem os MDAs e os distanciam dos MDTs.

Modulares

A modularidade dos MDAs permite dividir o processo de desenvolvimento num conjunto de actividades a serem executadas pelos membros da equipa. Com o objectivo de aumentar a agilidade do processo de desenvolvimento e a sua capacidade de se adaptar a alterações, podem ser adicionadas ou removidas actividades ao processo de desenvolvimento sem que isso crie impacto negativo ou atrase o processo de desenvolvimento como um todo.

Iterativos

Os MDAs efectuam uma divisão do processo de desenvolvimento em pequenos ciclos. Ao contrário dos MDTs, que têm início com a especificação dos requisitos pelo cliente e terminam com a validação do produto desenvolvido por parte do próprio cliente, o desenvolvimento seguido quando se adopta um MDA não é efectuado de forma sequencial. Os MDAs efectuam ciclos de desenvolvimento de curta duração, no final dos quais é efectuada uma revisão aos requisitos definidos e planificada a próxima iteração do desenvolvimento. Esta característica dos MDAs possibilita a rápida verificação do código e eventuais correcções.

Incrementais

O desenvolvimento incremental caracteriza os MDAs. No final de cada ciclo de desenvolvimento são criadas pequenas *releases* funcionais às quais novas funcionalidades vão sendo adicionadas. Esta característica permite que as actividades do processo de desenvolvimento, em particular a implementação, sejam efectuadas em paralelo. Após a implementação de uma nova funcionalidade, esta é testada e integrada na próxima *release*.

Convergentes

A criação de prioridades nos requisitos do projecto e a criação de *releases* funcionais no final de cada iteração possibilitam adicionar, a cada *release*, novas funcionalidades. As

funcionalidades que são adicionadas ao produto, a cada iteração, são definidas pelo cliente juntamente com a equipa do projecto. Desta forma, cada nova *release* converge para um produto final que corresponde às verdadeiras necessidades do cliente.

Lightweight

Os MDAs são considerados *lightweighted* no sentido que não defendem:

- A necessidade de uma especificação de requisitos exaustiva;
- A concepção detalhada, em fases iniciais do projecto, da arquitectura a implementar;
- A criação e manutenção exaustiva de documentação.

A remoção de todas as tarefas desnecessárias ou a redução do esforço aplicado a tarefas menos prioritárias conferem aos MDAs uma maior agilidade e uma redução no tempo de desenvolvimento de um projecto de software.

Adaptativos

Nos MDAs é possível remover, adicionar ou alterar requisitos sem que tal crie um impacto negativo na produtividade ou prazos do projecto. Esta característica, de adaptação às alterações existentes nos projectos de software, apenas é possível pelas práticas e conceitos inerentes aos MDAs. Prioritização dinâmica dos requisitos, desenvolvimento incremental e um plano de projecto modular são práticas e características que possibilitam a fácil adaptação às alterações.

Cooperativos

A necessidade da existência de uma relação estrita entre a equipa do projecto e o cliente, a utilização de práticas como *pair programming*[1] e a desvalorização dos documentos como via de comunicação entre os membros da equipa do projecto, conferem uma maior colaboração e cooperação entre os seus elementos e com o cliente.

People-oriented

Os MDAs consideram as pessoas como o factor mais importante no desenvolvimento, favorecendo as relações inter-pessoais e a criatividade individual de cada membro da equipa do projecto. Com o objectivo de facilitar e promover as relações inter-pessoais são criadas equipas com um número reduzido de elementos e as condições necessárias à motivação da equipa. Práticas comuns nos MDAs são *workshops* de reflexão e *stand-up-meatings* diários, que possibilitam a cada membro da equipa manifestar preocupações ou necessidades.

[1]Pair programming é uma prática introduzida pelo XP, que consiste em colocar dois programadores a trabalhar em simultâneo no mesmo computador.

3.3 Limitações

A maior limitação dos MDAs apontada na literatura [CH01, BT03], ainda que refutada por alguns "agilistas", prende-se com as dificuldades da aplicação dos MDAs em equipas com um número elevado de elementos. Cockburn e Highsmith concluem no seu livro que *"o desenvolvimento ágil de software é mais difícil em equipas numerosas (...). À medida que aumenta o tamanho, coordenar a comunicação torna-se um problema dominante"* [CH01].

Uma das práticas advogadas pelos MDAs é a entrega rápida de software funcional. O software deve ser entregue em sucessivas *releases* funcionais, implementadas em ciclos de desenvolvimento, de duração relativamente curta (tipicamente de 2 a 4 semanas). A importância dada à entrega rápida de resultados pode também constituir um problema quando não é efectuada uma gestão correcta do projecto. Esta prática, quando aplicada a sistemas com maior complexidade, pode levar à re-implementação de algumas funcionalidades se a arquitectura desenhada anteriormente não for escalável [Boe02]. Este problema pode ser acrescido pelo facto dos MDAs se focarem apenas no essencial.

Como foi já referido, os MDTs são baseados num conjunto sequencial de tarefas ou actividades estaticamente definidas, dando grande importância à definição de requisitos no início do projecto. Este tipo de métodos assenta no pressuposto de que é possível replicar em vários projectos o sucesso conseguido em projectos anteriores, utilizando os mesmos conceitos e práticas. Assim, todo o código, fases do projecto, decisões de implementação, revisões de código e produto final encontram-se bem documentados. No entanto, grande parte dos MDAs não suporta este tipo de práticas durante os seus ciclos de vida, utilizando técnicas de *pair programming* e técnicas informais de revisão que ainda não se provaram adequadas a projectos críticos e de grandes dimensões. A importância dada pelos MDAs ao conhecimento implícito (*tacit knowledge*) existente nas equipas de desenvolvimento é também uma limitação dos MDAs importante de referir. Esta prática pode introduzir erros na concepção ou na implementação de um sistema, que não serão facilmente detectados devido à falta de documentação [LBB+02].

Os MDAs são direccionados para o contacto com o cliente dando grande importância ao envolvimento deste nas diferentes fases do projecto. No entanto, esta prática pode revelar-se também uma limitação caso exista um maior número de clientes envolvidos, ou caso o representante do cliente não possua o conhecimento necessário ou autorização para definir ou modificar requisitos. Nestas situações um controlo e especificação detalhada conseguida através da utilização de documentos, utilização de um plano do projecto de-

talhado e revisão da concepção e implementação poderiam reduzir os riscos existentes.

3.4 Dificuldades na adopção de MDAs

"Agile is not for everyone" [CH01]

O sucesso na adopção de MDAs, descrito por várias empresas [SA05, Gre01, Bar06], levou a que muitas outras mostrassem também o seu interesse por estes métodos. No entanto, a adopção de MDAs implica alterações a vários níveis dentro da organização, existindo também algumas condicionantes à sua adopção.

Algumas dificuldades podem surgir na adopção dos MDAs, como por exemplo [Amb05]: *atitude tradicional, postura anti-ágil, raciocínio limitado, resistência à mudança, especialização, desactualização, mentalidade orientada à documentação e integração súbita.*

3.4.1 Atitude tradicional

Muitos profissionais da área das TICs acostumaram-se às abordagens tradicionais, tornando-se pouco receptivos a novas e revolucionárias abordagens. Isto deve-se ao facto de que, nos últimos anos, o desenvolvimento de software foi dominado pelos MDTs e pelas abordagens mais convencionais. Assim, a maioria dos profissionais das TICs quer identificar todos os requisitos do projecto nas fases iniciais do mesmo. Na transição do paradigma dos MDTs para os MDAs é necessário evidenciar as vantagens destes últimos e realizar formação e actualização dos profissionais envolvidos. Pessoas pouco receptivos à introdução de novos métodos de desenvolvimento podem dificultar a mesma.

3.4.2 Postura anti-ágil

O número de profissionais da indústria de software que investem tempo no estudo de novos métodos é reduzido. Em especial, a introdução de MDAs em contextos industriais ainda encontra muitos obstáculos. Tal problema pode surgir por dois motivos. Muitos consideram os MDAs como um método de *code-and-fix* disfarçado ou simplesmente adoptam uma postura anti-ágil. Assim, é necessário introduzir os MDAs de forma incremental, possibilitando a adaptação dos envolvidos.

3.4.3 Raciocínio limitado

Muitos profissionais na área das TICs reduzem o campo de escolha a duas opções, adoptando uma abordagem de "tudo ou nada". Por exemplo, a maioria das empresas terá a sensação de que a escolha ideal se encontra entre adoptar MDTs ou MDAs. No entanto existe pelo menos mais uma opção que é a combinação de MDAs e MDTs. Esta combinação poderá ser mais apropriada para determinados projectos, a avaliar [BT03].

3.4.4 Resistência à mudança

Dúvidas quanto às capacidades pessoais de adoptar MDAs ou quanto às alterações no ambiente de trabalho podem conduzir a uma resistência ou medo à mudança. Uma vez que o medo está geralmente associado à perda e falha, na tentativa de defesa, é possível que alguns profissionais se apresentem renitentes quanto à vantagem de adoptar novos métodos de desenvolvimento.

3.4.5 Especialização

Vários anos a desenvolver software, tendo por base os MDTs, levou à especialização das pessoas em determinadas áreas da engenharia de software. Isto origina profissionais com grandes capacidades numa ferramenta ou prática de desenvolvimento em particular, mas com conhecimentos reduzidos ou inexistentes em outros aspectos do desenvolvimento. Exemplo disto é o caso de gestores de projecto sem conhecimento das tecnologias utilizadas pela sua equipa ou programadores sem competências mínimas em concepção ou modelação da arquitectura. Por forma a colmatar este problema, os profissionais das TIs devem ser incentivados e formados no sentido de, além de especializados em determinada área, possuírem também conhecimentos básicos de aspectos técnicos e de negócio associados ao desenvolvimento de software.

3.4.6 Desactualização

Muitos profissionais, por se terem especializado numa determinada área, estagnaram a sua capacidade de aprendizagem. Assim, grande parte destes profissionais não se encontra actualizada no que diz respeito às últimas novidades na área do desenvolvimento de software. Cortes no orçamento das empresas para cursos de formação e actualização técnica contribuem para essa situação. No sentido de aumentar a eficiência no desenvolvimento de software é do interesse da empresa investir na formação dos seus colaboradores, possibilitando que os mesmos tenham algum tempo disponível para investigar, estudar e manter-se actualizados.

3.4.7 Mentalidade orientada à documentação

É generalizada a noção de que, no sentido de produzir software com qualidade, é necessário adoptar uma abordagem extremamente orientada à documentação, produzindo documentação compreensiva e detalhada. É necessário ter em consideração que os MDAs não colocam de lado toda a documentação, mas sim a documentação desnecessária. De facto, advêm alguns problemas da utilização excessiva de documentação. Os documentos são um meio de comunicação unilateral e desta forma possuem alguma limitação pois, para o leitor, a interpretação poderá ser ambígua. Sem a presença do autor do documento, a análise e leitura de alguns pontos do documento pode ser mal interpretada, não existindo a possibilidade de clarificar esses mesmos pontos.

3.4.8 Integração súbita

É bastante perigoso, não só na adopção de MDAs mas de qualquer alteração nos métodos de desenvolvimento de uma empresa, efectuar uma integração de novos conceitos numa só fase. A transição para novos métodos de desenvolvimento, nomeadamente os ágeis, deve por isso ser realizada de forma faseada e progressiva.

3.5 Manifesto ágil

Os MDAs tiveram origem com o manifesto ágil em 2001 (*http://agilemanifesto.org/*). Em Fevereiro de 2001 dezassete profissionais da área das TICs e defensores de diversos métodos de desenvolvimento inovadores, que mais tarde viriam a integrar o grupo dos MDAs, reuniram-se e assinaram um documento simbólico denominado manifesto ágil. Este manifesto reúne os valores e princípios defendidos univocamente pelos membros da "Aliança Ágil"[2]. No início do manifesto é possível ler-se

"Nós estamos a descobrir formas melhores de desenvolver software (...)"

A partir da frase anterior é possível identificar qual o principal objectivo do grupo de profissionais responsáveis pelo que é hoje conhecido como movimento ágil. Assim, os defensores destas *"novas formas de desenvolver software"* definiram os seguintes valores no manifesto ágil, os quais se tornaram na pedra basilar dos MDAs:

Indivíduos e interacções são mais importantes que processos e ferramentas;

Software executável é mais importante que documentação completa e detalhada;

[2]Designação do grupo de defensores das metodologias ágeis e signatários do manifesto ágil

Colaboração do cliente é mais importante do que negociação de contratos;

Respostas rápidas a alterações é mais importante do que seguir o plano inicial.

Adicionalmente foram também definidos, no manifesto, os seguintes princípios ágeis que corroboram os valores apresentados. Estes princípios têm dois objectivos principais: (1) ajudar a uma melhor compreensão do que são os MDAs e (2) orientar as equipas dos projectos a determinar se estão de facto a utilizar um MDA.

1. A maior prioridade é a satisfação do cliente através da disponibilização atempada e contínua de software com valor;
2. Devem-se aceitar as alterações de requisitos, mesmo quando isso acontece em fases tardias do desenvolvimento. Os MDAs utilizam as alterações para fornecer vantagem competitiva ao cliente;
3. Deve-se disponibilizar software operacional com frequência, em intervalos que podem variar entre as duas semanas e os dois meses, embora com preferência pelos prazos mais curtos;
4. Os *stakeholders* e os programadores têm que trabalhar diariamente em conjunto ao longo do projecto;
5. Os projectos devem ser executados por indivíduos motivados. A partir daqui, há que fornecer-lhes o ambiente e o suporte que precisam, além de confiar neles para a realização do trabalho;
6. O método mais eficiente e eficaz de transmitir informação a uma equipa de desenvolvimento e dentro da mesma é a conversação frente-a-frente;
7. O software operacional é a principal métrica de progresso;
8. Os MDAs promovem o desenvolvimento sustentado. Os patrocinadores, os especialistas em desenvolvimento e os utilizadores devem ser capazes de manter indefinidamente um ritmo constante;
9. A atenção contínua à excelência técnica permite aumentar a agilidade;
10. A simplicidade - enquanto arte de minimização da quantidade de trabalho por fazer - é essencial;
11. As melhores arquitecturas, os melhores requisitos e os melhores desenhos de arquitecturas emergem de equipas que se auto-organizam;

12. A intervalos de tempo regulares, a equipa deve reflectir sobre formas de se tornar mais eficaz, procedendo seguidamente a ajustes no seu comportamento com esse fim em mente.

Este manifesto, e todos os valores e princípios nele definidos, representam a filosofia por detrás dos MDAs e que devem ser visíveis, na teoria, em todas as práticas existentes nos diversos MDAs. Este estudo será efectuado no capítulo seguinte. De seguida serão apresentados com mais detalhe os valores e princípios definidos no manifesto ágil e que foram introduzidos anteriormente.

3.5.1 Os Valores

Indivíduos e interacções

Uma boa interacção entre os indivíduos envolvidos e empenhados em terminar uma tarefa ou actividade é um factor fundamental para o sucesso da mesma. Muitas das vezes um erro pode passar imperceptível quando a validação é efectuada pela mesma pessoa que o cometeu inicialmente. Conscientes deste facto, os defensores dos MDAs valorizam mais a interacção entre os indivíduos do que os processos e ferramentas, recorrendo a práticas como o *pair programming*, a ambientes de trabalho que promovam a interacção e a técnica de *peer-reviewing*.

Quando se efectua o desenvolvimento de software seguindo um método tradicional, a capacidade e autorização para decidir sobre factores de implementação ou solução de problemas encontra-se, na prática, centralizada. Este ponto comum de decisão pode ser constituído por uma ou mais pessoas, no entanto raramente representa a totalidade da equipa do projecto. Esta característica acrescida pelo facto das tarefas atribuídas serem raramente partilhadas por mais que um indivíduo, dificulta a comunicação entre os membros da equipa, fazendo com que a partilha de ideias e pensamento conjunto seja escasso ou inexistente. Os MDAs tentam eliminar este ponto central de decisão com o objectivo de melhorar a comunicação e valorizar a interacção entre os membros das equipas. O que isto significa na prática é que passam a existir equipas de pequenas dimensões, onde cada pessoa ou grupo de pessoas é responsável por lidar com os problemas e decisões de implementação que surgem no decorrer do projecto. Apesar de existir um ponto central, responsável pela coordenação e gestão do projecto, as decisões são tomadas em conjunto. Embora seja apontada como uma vantagem dos MDAs, a característica apresentada anteriormente limita a aplicação destes métodos a projectos de grandes dimensões. Nestes casos, é uma boa prática dividir as equipas de grandes dimensões em várias equipas,

reduzindo assim o número de canais de comunicação e facilitando a interacção entre os membros da mesma.

Este valor - indivíduos e interacções são mais importantes que processos e ferramentas - definido no manifesto ágil é visível em vários MDAs, como por exemplo no eXtreme Programming (XP) e no Scrum [MR05]. O XP, um dos mais conhecidos e utilizados MDAs, advoga a criação de equipas pequenas com um máximo de dez elementos. Desta forma é possível gerir facilmente os mecanismos de comunicação existentes. Uma das práticas existentes no XP que dá ênfase a este valor definido no manifesto é o recurso à técnica de *pair programming*. No entanto, e embora facilitando a interacção, esta prática (*pair programming*) exige que todos os membros da equipa sejam capazes de trabalhar em conjunto, não existindo espaço para individualismo ou isolamento.

No caso do Scrum, outro MDA que privilegia o "pensamento em grupo", a importância dada às interacções e às pessoas é visível no tipo de reuniões efectuadas e também no tamanho ideal de equipas. Detalhes sobre as práticas do Scrum serão introduzidas numa secção para esse efeito.

Software executável

Grande parte da documentação criada nos projectos de software diz respeito a documentos de suporte ao projecto, e.g., documentos de requisitos, manual do utilizador, planos de actividades do projecto e documentos que descrevem a arquitectura utilizada. Estes documentos servem para controlo interno do projecto mas também para fornecer, ao cliente, a informação necessária sobre a progressão do projecto e sobre o produto final. No entanto, a documentação apresentada é muitas vezes demasiado técnica, factor que dificulta a compreensão por parte do cliente. A maior parte dos MDTs comete o erro de recorrer em demasia à documentação que, como já foi mencionado, pode ser visto como um problema pela característica de comunicação unidireccional, possível ambiguidade na leitura, difícil manutenção e tempo despendido na criação de documentação. No método em cascata, descrito no capítulo anterior, a documentação é o resultado de cada uma das fases do método. Existe por isso um grande esforço em criar documentação sendo assim muito fácil cometer o erro de exagerar na informação presente nesses documentos, aumentando o esforço necessário para manter esses mesmos documentos.

Qualquer programador entende a necessidade da existência de código documentado. Uma pequena parte do código não documentado pode ser bastante complicada de inter-

pretar. Para grande parte dos programadores a documentação continua a ser um estigma e talvez por isso seja difícil encontrar código bem documentado e com documentação externa para suporte adicional. No entanto, a solução para este problema não é "convencer" ou impor a criação de documentos mas sim rever a ênfase que é dada a estes.

No sentido de melhorar a comunicação com o cliente, reduzir o tempo despendido na criação de documentos difíceis de manter e permitir aos programadores iniciarem o desenvolvimento mais cedo, os MDAs valorizam o software executável em detrimento da utilização de documentação, mesmo que esta seja de compreensão fácil. Apesar desta característica dos MDAs é necessário salientar que, embora reduzam o ênfase dado à criação de documentação, os MDAs não descartam por completo a utilização da mesma. No caso do XP esta característica, bem como as restantes características dos MDAs, é levada ao extremo[3]. No XP a documentação é o próprio código (suportado por comentários adicionais) e a concepção da solução cuja manutenção e actualizações serão efectuadas em simultâneo com as alterações efectuadas ao código implementado - *evolutionary design* [MR05]. Para os defensores do XP a criação de código simples e auto-descritivo reduz significativamente a quantidade de documentação necessária num projecto.

Colaboração do cliente

Este valor definido no manifesto ágil sugere que deve ser aplicado um maior esforço para trabalhar directamente com o cliente, envolvendo-o no desenvolvimento do produto, em vez de desperdiçar tempo com acordos burocráticos e negociações contratuais. Este valor distingue novamente a forma como o desenvolvimento de software é abordado pelos MDAs. Nos MDTs o produto resultante de um projecto de software é encarado como resultado de um acordo contratual entre o cliente e a entidade responsável por fornecer determinado serviço. Desta forma, os únicos contactos com o cliente são efectuados apenas no início de um projecto, para definição dos requisitos, e no final do mesmo, para os testes de aceitação. Este tipo de abordagem resulta muitas vezes em software que não satisfaz os requisitos do cliente em funcionalidades implementadas que não possuem qualquer tipo de valor de negócio para o cliente e, em casos mais críticos, em projectos cancelados.

Pelos motivos apresentados anteriormente, os MDAs colocam grande importância no contacto e interacção com o cliente, considerando o *feedback* contínuo, por parte do cliente, um factor e condicionante fundamental para o sucesso de um projecto de software. Mais uma vez é necessário salientar que os MDAs não negligenciam a definição de requisitos no

[3]Daí o nome eXtreme Programming

início de um projecto, no entanto estes não são "congelados" nesta fase embrionária de um projecto e são ajustados ao longo do ciclo de vida do projecto, com o auxílio do cliente ou representante deste. Desta forma é possível concluir o projecto com a garantia que os requisitos definidos pelo cliente são cumpridos e que todas as funcionalidades implementadas possuem valor de negócio para o cliente. Exemplo de um estudo que compara a transição de um contacto com o cliente tipicamente tradicional para um contacto directo e contínuo, característica dos MDAs, é descrito em *"Agile customer engagement: a longitudinal qualitative case study"* [HF06]. Neste artigo são visíveis as vantagens e dificuldades que advêm deste tipo de transição.

Respostas rápidas a alterações

Como último valor definido no manifesto ágil temos a valorização de respostas rápidas a alterações. Este talvez seja o valor que mais diferencia os MDAs dos MDTs. Como foi já referido, os MDTs destacam-se pelo facto de se cingirem estritamente a um plano pré-definido onde as alterações constantes, que caracterizam os projectos de software, não são geralmente contempladas nesse plano mas sim previstas. Estas alterações provocam geralmente a revisão do plano, o que exige alteração a documentos e aceitação dessas alterações por parte do cliente. Todas estas adversidades criam entropia que resulta em atrasos e aumenta a dificuldade em manter código, documentação e requisitos. No caso dos MDAs não se pretende prever todas as alterações possíveis de suceder. Como alternativa, os MDAs sugerem que é mais fácil e produtivo a adaptação e resposta rápida às alterações que a tentativa, na maior parte das vezes frustrante, de prever essas mesmas alterações. Este valor definido no manifesto ágil encontra-se intimamente ligado com o valor apresentando anteriormente, a colaboração do cliente, já que é o cliente que comunica as alterações a efectuar no projecto e participa activamente na escolha dos requisitos a implementar a cada iteração do desenvolvimento.

Nos MDAs os requisitos mantêm-se inalterados durante cada iteração do desenvolvimento. Assim, a criação de um plano e detalhe deste apenas diz respeito à iteração seguinte e é planificada no final da fase anterior. Desta forma, durante a revisão efectuada pelo cliente na preparação da iteração seguinte, os requisitos podem ser revistos e as prioridades estabelecidas, o que permite alterar o plano sem que tal implique um impacto significativo em todo o projecto.

A gestão dos requisitos é efectuada de forma muito semelhante em todos os MDAs através da selecção e criação de prioridades dos requisitos. Esta selecção de requisitos que

serão implementados a cada iteração do projecto, e prioritização dos mesmos, é efectuada pelo cliente em conjunto com a equipa do projecto. De seguida os requisitos seleccionados são formalizados. A forma de formalização varia entre os MDAs, sendo que, por exemplo, no Scrum temos o *product backlog* e no XP as *user stories*. Estas e outras práticas dos MDAs serão revistas no próximo capítulo desta dissertação.

3.5.2 Os Princípios

Princípio 1 - *A maior prioridade é a satisfação do cliente através da disponibilização atempada e contínua de software com valor*

Grande parte dos métodos de desenvolvimento existentes assume que a capacidade de entregar um produto com valor comercial para o cliente é atingido através da capacidade de seguir e cumprir o plano do projecto. As características voláteis dos projectos de software demonstram que tal não é verdade. Os requisitos e as funcionalidades têm de ser reavaliados continuamente, durante o ciclo de vida do projecto, de forma a ser possível determinar quais desses requisitos continuam ou não a representar valor de negócio para o cliente. Assim, a capacidade de seguir um plano predefinido tem pouca contribuição para o sucesso de um projecto de software.

Os MDAs são orientados para o cliente. O princípio apresentado, definido no manifesto ágil, salienta que a maior prioridade do MDAs é a satisfação do cliente. Neste princípio é possível destacar três frases que demonstram de que forma os MDAs se propõem a aumentar a satisfação do cliente. [Koc05].

> *(...) satisfação do cliente através da disponibilização* ***atempada*** *(...) de software (...)*

Adicionalmente, são muitos os métodos de desenvolvimento que depositam confiança na capacidade dos profissionais em identificar, em fases iniciais, todos os requisitos de um projecto. Desta forma, uma parte considerável do tempo é gasto em análises abstractas no início do projecto. Por sua vez os MDAs optam por criar as bases mínimas para o arranque do projecto, iniciando de imediato a criação de um produto funcional. Ainda que nas primeiras iterações do ciclo de vida do projecto seja produzido um produto com o mínimo de funcionalidades, este proporciona ao cliente e à equipa do projecto um exemplo concreto do esforço que será necessário aplicar e um protótipo do produto a implementar. Desta forma, detalhes sobre as funcionalidades, requisitos e decisões de implementação são deliberados sobre o software funcional desenvolvido nas diferentes iterações do projecto.

A cada iteração, ao protótipo inicial, vão sendo incluídas ou removidas funcionalidades, aumentando desta forma o valor de negócio e comercial que o produto terá para o cliente.

> *(...) satisfação do cliente através da disponibilização (...)* ***contínua*** *de software*

Os MDAs caracterizam-se também por efectuar entregas contínuas do software implementado a cada iteração do ciclo de vida do projecto. Entregas regulares e contínuas de um produto funcional permitem ao cliente efectuar ajustes em detalhes que só identificáveis em fases mais avançadas do projecto. A cada nova iteração podem surgir novos detalhes, dúvidas, requisitos e funcionalidades a serem implementadas.

> *(...) satisfação do cliente através da disponibilização (...) de software com* ***valor***

Entende-se por *"software com valor"* um produto cujas funcionalidades e requisitos correspondam as necessidades do cliente e possuam valor comercial e de negócio para este. Esta característica, de criar e implementar software com valor para o cliente, é alcançada através da relação estreita entre o cliente e a equipa do projecto. Como foi já mencionado, os MDAs caracterizam-se pelo facto do cliente participar activamente na identificação das funcionalidades a serem implementadas a cada iteração do projecto.

Princípio 2 - ***Devem-se aceitar as alterações de requisitos, mesmo quando isso acontece em fases tardias do desenvolvimento***

A capacidade de adaptação às alterações existentes num projecto de software é, na teoria, uma característica comum a todos os MDAs. Isto porque todos os MDAs são definidos sobre o pressuposto de que todo o projecto de software é passível de ser alterado durante o seu ciclo de vida [Koc05]. Assim, deve ser esperado o surgimento de possíveis alterações de forma a ser possível responder eficientemente a essas mesmas alterações. Essa capacidade de adaptação às alterações surge nos MDAs com o sentido de dar uma vantagem competitiva às equipas dos projectos de software, relativamente à utilização dos MDTs que se limitam a seguir o plano e tentar prever à partida todas as alterações que possam surgir.

Ao contrário dos MDTs, os MDAs sugerem que as alterações aos requisitos devem ser aceites ainda que estas surjam em fases tardias do projecto, tendo sempre em consideração as consequências que advêm deste tipo de abordagem [FH01]. Este princípio, definido no manifesto ágil, é visível na forma como os MDAs tratam os requisitos de um projecto. Os

MDAs advogam a definição incremental dos requisitos de um projecto. Assim, os requisitos vão sendo refinados, ganhando detalhe, ao longo do projecto. A forma como a questão da definição dos requisitos é abordada varia entre os diferentes MDAs. Por exemplo, em XP é utilizada uma lista de *User Stories* e em Scrum é utilizado um *product backlog*. Estes conceitos serão detalhados mais à frente.

Este princípio não surge, no entanto, sem qualquer tipo de dificuldade ou limitação no que se refere à sua aplicação. Na prática, a aplicação deste princípio num projecto de software depende da capacidade e flexibilidade dos membros da equipa do projecto. Em casos onde a equipa do projecto possui poucos conhecimentos técnicos ou experiência prática, a capacidade de responder rapidamente às alterações num projecto pode ter o efeito indesejado.

Princípio 3 - ***Deve-se disponibilizar software operacional com frequência, em intervalos que podem variar entre as duas semanas e os dois meses, embora com preferência pelos prazos mais curtos***

Os MDAs valorizam a entrega de software executável ao cliente em vez de documentação, no final de cada iteração do projecto. Assim é possível, a cada iteração, refinar os requisitos, efectuar possíveis alterações e acrescentar ao produto funcionalidades de forma incremental. Desta forma é dada ao cliente a possibilidade de interagir com *releases* incrementais daquilo que será mais tarde o produto final. Este contacto directo com diferentes versões, que convergem para o produto final ao longo do tempo, permite ao cliente ter percepção daquilo que está a ser realmente implementado e das funcionalidades que vão sendo incorporadas no produto. Este conceito possibilita ao cliente decidir quais os requisitos e funcionalidades que representam maior valor de negócio para este. Consegue-se assim, no final do projecto de software, um produto cujas funcionalidades implementadas são o que era realmente pretendido pelo cliente. No entanto é necessário ter em consideração que o que se pretende com a definição deste princípio não é a colocação de cada uma das *releases* em produção, pois tal acção seria inviável. O objectivo é meramente possibilitar ao cliente uma maior interacção com a evolução do produto, possibilitando ajustes e planificação das iterações seguintes do desenvolvimento.

Este princípio está directamente relacionado com o primeiro princípio do manifesto ágil pois ambos defendem, directa ou indirectamente, a " (...) disponibilização atempada e contínua de software com valor".

Princípio 4 - *Os* stakeholders *e os programadores têm que trabalhar diariamente em conjunto ao longo do projecto*

Entende-se por *stakeholders* todos os profissionais envolvidos no projecto e que não integram a equipa técnica propriamente dita, em contraste com os programadores [Koc05]. Este princípio vem salientar a importância e a necessidade da interacção entre a equipa de desenvolvimento, nomeadamente os programadores, e outras pessoas envolvidas no projecto, em especial o cliente ou um representante deste.

Como já foi mencionado anteriormente, nos MDAs não é efectuado um levantamento de todos os requisitos nas fases iniciais do projecto, considerando a volatilidade destes nessas fases iniciais. Assim, nos MDAs é efectuado apenas o levantamento dos requisitos necessários para o arranque do projecto, sem que seja necessário detalhar esses mesmos requisitos. Ao longo do desenvolvimento e ciclo de vida do projecto esses requisitos são enriquecidos e refinados. Por si só esta técnica não é praticável, pois a equipa do projecto não é detentora das necessidades e objectivos comercias do cliente. Com o objectivo de eliminar esta lacuna, os MDAs reforçam a comunicação existente entre a equipa do projecto e os restantes intervenientes (*stakeholders*). A palavra "diariamente", visível neste princípio, vem reforçar a necessidade de existência de uma boa comunicação.

Princípio 5 - *Os projectos devem ser executados por indivíduos motivados. A partir daqui, há que fornecer-lhes o ambiente e o suporte que precisam, além de confiar neles para a realização do trabalho*

Os MDAs defendem, como se pode ler no manifesto ágil, que *"indivíduos e interacções são mais importantes que processos e ferramentas"*. A importância e confiança que estes métodos de desenvolvimento colocam nas pessoas, como membros da equipa de um projecto, é visível neste princípio. Segundo os "agilistas", as pessoas são um factor preponderante para o sucesso de um projecto de software.

Independentemente do tipo de linguagem de programação, processo ou paradigma, existe um factor que é comum a todos os projectos. Esse factor são as pessoas. Os MDAs não se limitam a impor um processo e um conjunto de actividades, aos quais os membros da equipa têm de se restringir. Este tipo de abordagem não apresenta qualquer tipo de problema em situações estáveis mas quando surgem alterações ao plano, este tipo de decisão centralizada, dificulta a adaptação à mudança. Os MDAs, pela confiança que é depositada nas pessoas assumem que estas são capazes de tomar decisões importantes

quando as situações assim o exigem.

Este princípio evidencia também a importância em existir um ambiente que promova a motivação e o bem estar geral da equipa. Isto também distancia os MDAs dos MDTs. Os MDTs assumem que a motivação intrínseca é rara, assim são criadas sequências rígidas de actividades que condicionam e forçam o trabalho aos membros da equipa.

Princípio 6 - ***O método mais eficiente e eficaz de transmitir informação a uma equipa de desenvolvimento e dentro da mesma é a conversação frente-a-frente***

A utilização da documentação como processo de comunicação possui limitações pela sua característica de comunicação unilateral e ambiguidade. Torna-se assim necessária a utilização de técnicas de comunicação directas. Como foi já referido os MDAs destacam como meio de comunicação preferencial a comunicação "frente-a-frente". Esta característica contrasta com os MDTs, que dão preferência à documentação como meio de comunicação utilizado e como resultado de cada fase do ciclo de vida do projecto. No entanto torna-se necessário relembrar que os MDAs não descartam totalmente a utilização de documentos reconhecendo a necessidade da comunicação escrita. Os MDAs limitam-se a salientar a importância da comunicação "frente-a-frente" e a reduzir o esforço depositado na criação de documentação, muitas vezes desnecessária.

Apesar de serem visíveis as vantagens existentes numa comunicação "frente-a-frente", existem também desvantagens e limitações que não podem ser ignoradas. Exemplo dessas limitações é o facto de não existir uma persistência do que é transmitido ou acordado durante a comunicação. Desta forma este tipo de comunicação apenas é possível quando existe elevado grau de confiança entre os diversos membros da equipa. Destaca-se assim mais uma das características dos MDAs - a confiança depositada nas pessoas. Outro dos problemas, consequência da não persistência numa comunicação "frente-a-frente", ocorre aquando da entrada de novos elementos nas equipas. Neste tipo de situações torna-se necessária a existência de um meio capaz de transmitir o conhecimento aos novos elementos. Ainda que uma comunicação "frente-a-frente" seja o meio mais eficaz de transmitir conhecimento, em grandes empresas, este tipo de comunicação e acompanhamento nem sempre é possível. Nestes casos é necessário recorrer a documentos que contenham informação e conhecimento necessários.

Princípio 7 - *O software operacional é a principal métrica de progresso*

Pelas características intangíveis do produto resultante de um projecto de software, torna-se muitas vezes complicado avaliar o progresso desse mesmo projecto. A forma mais eficaz de avaliar o progresso de um projecto de software é avaliar as funcionalidades e requisitos implementados. A avaliação do progresso de um projecto de software através das funcionalidades implementadas permite a definição objectiva de *milestones* e fornece uma visão real do estado actual do projecto. Adicionalmente, este tipo de avaliação do progresso de um projecto de software permite à respectiva equipa refinar o conhecimento sobre as funcionalidades a serem implementadas e um melhoramento no cálculo do esforço associado à implementação de funcionalidades futuras. No entanto, as características inerentes aos MDTs dificultam este tipo de avaliação pelo facto de:

- fecharem os requisitos em fases iniciais de um projecto. O facto dos requisitos, num projecto de software, estarem sujeitos a constantes alterações torna difícil a avaliação das funcionalidades implementadas;
- seguirem um ciclo de vida de um projecto que é sequencial. Não existem versões incrementais ao longo do projecto que permitam a avaliação das funcionalidades implementadas até ao momento.

Os MDAs, por outro lado, possibilitam este tipo de controlo do progresso de um projecto de software. As características incrementais e a prática de desenvolvimento iterativo, com criação de versões funcionais ao longo do ciclo de vida do projecto de software, colocam os MDAs numa posição única e ideal que facilita este tipo de controlo.

Princípio 8 - *Os MDAs promovem o desenvolvimento sustentado. Os patrocinadores, os especialistas em desenvolvimento e os utilizadores devem ser capazes de manter indefinidamente um ritmo constante*

Os MDAs são, na sua essência, iterativos e incrementais. Cada método proposto, dentro do grupo dos MDAs, advoga iterações relativamente curtas durante as quais são criadas versões funcionais daquilo que será mais tarde o produto final. Cada uma dessas versões acrescenta novas funcionalidades e requisitos implementados que se reflectem, para o cliente, em valor de negócio. Esta característica dos MDAs faz com que seja possível manter um ritmo contínuo e regular durante todo o ciclo de vida do projecto de software. Os MDAs advogam, inclusivamente, que apenas seja cumprido o horário de trabalho dito normal de forma a não sobrecarregar os elementos das equipas dos projectos. Profissionais cansados e sem motivação não são capazes de desenvolver eficientemente. A falta

de qualidade, visível em situações de *stress*, leva na maior parte das vezes à criação de software com pouca qualidade e "atalhos" que provocam efeitos colaterais e indesejados no ritmo do projecto.

As características dos MDAs permitem um desenvolvimento contínuo evitando assim as noites e fins-de-semana de trabalho com o objectivo de cumprir os prazos estipulados.

Princípio 9 - *A atenção contínua à excelência técnica permite aumentar a agilidade*

O princípio apresentado aponta a excelência técnica como um pré-requisito para a existência de agilidade. Apenas mantendo uma atenção constante à qualidade técnica do desenvolvimento é possível a criação de uma arquitectura capaz de suportar alterações rápidas a requisitos e funcionalidades [Koc05]. Pelas características dos MDAs, que foram anteriormente apresentadas, a implementação de código simples, modular e bem documentado é um factor preponderante para manter a agilidade de um projecto sem comprometer a qualidade do mesmo.

O conceito generalizado de excelência técnica aparece frequentemente relacionado com custos excessivos, arquitecturas estáveis e uma concepção da arquitectura que tenta contemplar possíveis problemas que possam surgir no futuro. Assim tenta-se alcançar uma maior qualidade no produto final e consequentemente a excelência técnica através de uma análise exaustiva em fases iniciais dos projectos e através de testes e *refactoring* em fases finais. No caso dos MDAs o objectivo é a criação, desde o início e de forma contínua até ao final do projecto, de código com qualidade tendo sempre em atenção a excelência técnica na implementação deste. Desta forma existe uma procura contínua pela qualidade.

Princípio 10 - *A simplicidade - enquanto arte de minimização da quantidade de trabalho por fazer - é essencial*

O desenvolvimento de software pode ser abordado de inúmeras formas. Nesta dissertação foi já feita referência aos inúmeros de métodos, paradigmas e técnicas que têm vindo a ser apresentados ao longo dos anos. No caso dos MDAs estes defendem a simplicidade no desenvolvimento de software. Pretende-se desta forma minimizar a quantidade de actividades desnecessárias, efectuadas e implementadas ao longo de um projecto de software. Desta forma será possível contrariar os resultados apresentados no capítulo 1, que apontam para uma média de 45% de funcionalidades implementadas sem qualquer valor de negócio para o cliente final.

No entanto, a utilização do conceito de simplicidade e remoção das tarefas ou actividades desnecessárias deve ser efectuada cuidadosamente, de forma a não comprometer o projecto de software. Ao eliminar determinada tarefa ou actividade é necessário ter em consideração as suas ramificações. Adicionalmente, algumas actividades que podem parecer à primeira vista desnecessárias são muitas vezes imprescindíveis para mitigar possíveis problemas.

Princípio 11 - ***As melhores arquitecturas, os melhores requisitos e os melhores desenhos de arquitecturas emergem de equipas que se auto-organizam***

Este princípio faz referência à necessidade de uma equipa de projecto coesa e toda ela responsável pelo sucesso, ou insucesso, do projecto de software. Assim, os MDAs procuram a criação de equipas auto-organizadas cujos membros sejam capazes de tomar decisões importantes quando as situações assim o exigirem. Esta filosofia representa um movimento contra a forma tradicional de gerir projectos, onde existe um ponto centralizado de decisão. Esta responsabilidade partilhada por todos os membros da equipa e a liberdade de decisão são, segundo os MDAs, uma forma de alcançar não só a motivação individual mas também a excelência técnica.

Princípio 12 - ***A intervalos de tempo regulares, a equipa deve reflectir sobre formas de se tornar mais eficaz, procedendo seguidamente a ajustes no seu comportamento com esse fim em mente***

É possível efectuar ajustes nas métricas associadas aos métodos de desenvolvimento que integram o grupo dos MDAs, como por exemplo, o tamanho de cada iteração e periodicidade na geração de relatórios. Adicionalmente, dependendo da tipologia do projecto, algumas práticas inerentes a determinado método de desenvolvimento podem não ser aplicáveis. Assim, este princípio sugere que as equipas de desenvolvimento se reunam, periodicamente, para reflectir de que forma é possível tornar o desenvolvimento de software mais eficiente. Esta prática tem por objectivo levar a um contínuo melhoramento das práticas utilizadas no projecto e como evitar no futuro erros passados.

Capítulo 4

Apresentação dos MDAs a Analisar

Neste capítulo são apresentados em detalhe os métodos XP e Scrum, que são os MDAs alvo da análise apresentada nesta dissertação. Os motivos que contribuíram para a selecção destes métodos serão apresentados no capítulo 5. Para cada um dos métodos seleccionados (XP e Scrum) será efectuada uma pequena introdução e serão apresentadas as práticas que cada um deles advoga, o ciclo de vida e as funções a desempenhar pelos membros da equipa.

4.1 eXtreme Programming

O eXtreme Programming foi desenvolvido em 1990 por Kent Beck, Ron Jeffries e Ward Cunningham enquanto trabalhavam no projecto Chrysler Comprehensive Compensation System (C3), da empresa Chrysler. Durante a realização deste projecto, estes três profissionais da área das TICs, concluíram que a comunicação, simplicidade, *feedback* frequente e atempado e coragem são os pré-requisitos para melhorar o processo de desenvolvimento de software [Pau01]. De seguida é apresentada uma breve descrição destes quatro valores:

Comunicação é um dos factores mais importantes no sucesso de um projecto [FPB95]. Grande parte dos problemas existentes em projectos de software advêm da ineficácia na comunicação. A única forma de manter a agilidade de um projecto de software, respondendo eficientemente às alterações, é através da comunicação e colaboração entre os membros da equipa;

Simplicidade na concepção da arquitectura a implementar. O objectivo é não tentar prever todos os problemas que possam surgir mas sim possuir um sistema simples,

onde seja possível efectuar alterações sem comprometer essa mesma simplicidade ou funcionalidades já implementadas;

Feedback frequente e atempado permite à equipa do projecto e ao cliente, como membro integrante dessa mesma equipa, efectuar a gestão do progresso do projecto, dos requisitos e das funcionalidades implementadas;

Coragem na tomada de decisões e ao assumir a responsabilidade pelos actos individuais e suas eventuais implicações num projecto de software.

Tendo por base os quatro valores apresentados, em 1999 foi apresentado um novo método de desenvolvimento na conferência *Technology of Object-Oriented Languages and Systems* - o XP. O XP surge com o objectivo de mitigar os problemas causados pelos longos períodos de desenvolvimento que caracterizavam até então os projectos de software [Bec99].

Não é possível falar das características do XP sem efectuar uma repetição das características dos MDAs apresentadas no capítulo anterior. Ainda assim é possível destacar o XP dos restantes MDAs pela forma extrema como aborda alguns conceitos, como por exemplo o envolvimento do cliente e a curta duração dos ciclos de desenvolvimento. Esta característica de levar 'ao extremo' os conceitos apresentados contribuíram para o nome do método.

4.1.1 Práticas

O XP, como qualquer outro método de desenvolvimento, tem como objectivo a criação eficiente de software. Com o objectivo de facilitar essa tarefa o XP apresenta um conjunto de doze práticas que devem ser aplicadas ao desenvolvimento de software. Nas secções seguintes serão apresentadas essas práticas, de acordo com o que é apresentado por Kent Beck [Bec99], efectuando uma pequena descrição das mesmas.

The Planning Game

Esta prática advoga uma interacção estreita entre o cliente e a equipa técnica onde cada uma das partes é responsável pela definição e identificação de um conjunto de métricas como o âmbito do projecto, definição de prazos, estimativa de esforço e impedimentos tecnológicos.

A necessidade de uma relação estreita entre o cliente e a equipa técnica do projecto é explicada pela existência de conhecimento partilhado. Apenas o cliente detém o conhecimento sobre os requisitos a serem implementados, enquanto que apenas a equipa técnica detém o conhecimento necessário para apresentar ao cliente estimativas sobre o esforço a ser utilizado na implementação desses mesmos requisitos. A equipa técnica é também responsável por elucidar o cliente quanto às consequências da utilização de uma ou outra tecnologia. Isto ocorre nas situações em que, por exemplo, o cliente define como pré-requisito a utilização de uma linguagem de programação que possa aumentar o risco de implementação e a dificuldades na manutenção do código ou na integração com outros sistemas externos.

Pequenas *Releases*

O objectivo da utilização desta prática é a colocação rápida em produção de pequenas partes do produto final. Cada uma destas partes - *releases* - deve ser o mais pequena possível e conter os requisitos de maior valor para o cliente. Cada nova *release* apresentada ao cliente é constituída pelo conjunto das funcionalidades que integraram a *release* anterior e das funcionalidades implementadas na nova iteração. Esta prática permite ao cliente, ao longo do ciclo de vida do projecto, ter contacto com o produto que se encontra em desenvolvimento, possibilitando à equipa técnica ter *feedback* por parte do cliente sobre a validade do que se encontra a ser implementado.

Metáforas

O recurso à utilização de metáforas, para descrever determinada funcionalidade, permite criar uma visão comum do funcionamento do produto, aproximando o cliente da equipa técnica. Desta forma é possível reduzir a utilização de expressões técnicas, muitas vezes de difícil compreensão para o cliente, na definição das *user stories* 55. Esta prática visa facilitar a comunicação entre o cliente e a equipa técnica. No entanto, a utilização desta prática pode revelar-se um desafio, pois requer que seja criado um desenho de alto nível da arquitectura que seja simultaneamente de simples compreensão pelo cliente, descreva o sistema a desenvolver e possua as informações suficientes para que os programadores sejam capazes de o traduzir numa implementação dessa mesma arquitectura.

Concepção Simples

Como foi já mencionado, os MDAs sugerem que é mais fácil e produtivo a adaptação e resposta rápida a alterações do que a tentativa de prever essas mesmas alterações. Com o objectivo de promover este conceito, o XP apresenta como prática a seguir a concepção

simples. Assim, toda a concepção da arquitectura e código implementado devem ser o mais simples possível.

Escrita de Testes e *Test-Driven Development*

Esta prática obriga à escrita de testes para verificação e validação do software implementado. Cabe aos clientes a tarefa de criar um conjunto de testes funcionais e de aceitação, que validam as funcionalidades implementadas do ponto de vista do negócio, e aos programadores a escrita de testes unitários, que verificam as funcionalidades implementadas do ponto de vista técnico. No XP uma funcionalidade implementada só se encontra completa e pronta para integrar uma *release* quando todos os testes, criados até ao momento, executam com sucesso. A escrita de testes deve acompanhar, ou até preceder, a implementação das funcionalidades (*Test-Driven Development*)

Refactoring

A prática de *refactoring* tem como objectivo a simplificação do código implementado, a remoção de ambiguidade e redundância no código. A aplicação desta prática permite aumentar a flexibilidade e a qualidade do código implementado. Após ser aplicado o *refactoring* a uma funcionalidade já implementada, os testes implementados até ao momento devem ser executados novamente.

Pair Programming

A qualidade técnica associada a uma concepção simples e à criação de testes rigorosos pode revelar-se uma actividade difícil de adoptar e manter [Goe03]. Com o objectivo de reduzir esta dificuldade e permitir uma constante revisão do código implementado, o XP introduz a prática de *pair programming*. Esta prática consiste em ter sempre dois programadores a trabalhar em simultâneo na mesma máquina. Cada um dos programadores desempenha um papel específico. No *pair programming* um dos elementos do par é responsável pela escrita do código, enquanto que o outro elemento é responsável pela verificação e revisão do código e por verificar a existência de casos que não tenham sido contemplados nos testes implementados, facilitando assim a tarefa de verificação. Esta prática mostra-se particularmente fundamental em situações em que um dos elementos do par não possui conhecimentos suficientes para realizar determinada tarefa.

Posse Colectiva do Código

Qualquer elemento da equipa de um projecto XP possui autorização para efectuar alterações ao código. No XP todos os elementos possuem, ainda que parcialmente, conhecimento sobre todos os componentes que integram o produto final e são encorajados a melhorar o código sempre que surge uma oportunidade nesse sentido. Desta forma a responsabilidade do código implementado é colectiva.

Integração Contínua

Após a implementação de uma nova funcionalidade ou de ajustes efectuados ao código existente (correcção de erros, *refactoring*, ...) deve ser criada uma actualização à *release* actual que reflicta e integre essas mesmas alterações. Esta integração deverá ser efectuada pelos elementos da equipa responsável pelas alterações e, idealmente, deverá existir uma máquina dedicada para esse efeito. Após a integração do código numa actualização da *release* actual esta deve ser testada e todos os testes devem executar com sucesso. Uma vez que se tem a garantia que a última actualização da *release* se encontrava 100% funcional, caso algum dos testes falhe, é da responsabilidade de quem efectua a integração a realização de possíveis correcções ao código integrado.

40 horas semanais

A regra do XP é clara - nenhum elemento da equipa do projecto deve efectuar horas extraordinárias duas semanas consecutivas [BA04]. Se tal acontece deverá então ser tratado como um problema de gestão do projecto. O motivo desta prática é facilmente compreendido, profissionais cansados não são capazes de ser criativos e de criar software com qualidade.

Cliente *On-Site*

No caso do XP o contacto contínuo com o cliente é mais uma vez uma das características dos métodos ágeis que é elevada ao extremo. O XP refere-se ao cliente como sendo o utilizador final ou um representante deste. O XP advoga a presença do cliente *on-site*. Isto significa que o cliente, ou um seu representante, deve estar sempre presente durante todo o ciclo de vida, trabalhando directamente com a equipa do projecto.

Utilização de *Coding Standards*

Os *coding standards* definem um conjunto de regras que descrevem de que forma o código deve ser implementado, que características da linguagem de programação devem ser aplicadas e que ferramentas devem ser utilizadas. Esta prática suporta a prática anteriormente apresentada que se refere à posse colectiva do código. A não utilização de *coding standards* aumenta a possibilidade de existir falta de concordância entre os elementos da equipa e código pouco legível, o que pode levar à falta de cooperação entre os elementos da equipa afectando assim a qualidade do projecto.

A utilização de *coding standards* permite que:

- Os programadores se sintam familiarizados quando trabalham com código produzido pelos restantes elementos;
- Elementos que venham a integrar a equipa do projecto sejam capazes de compreender facilmente o código implementado.

4.1.2 Ciclo de vida

O ciclo de vida do XP consiste em seis fases: 1) Exploração, 2) Planificação, 3) Iterações para a *release*, 4) Produção, 5) Manutenção e 6) Morte do projecto.

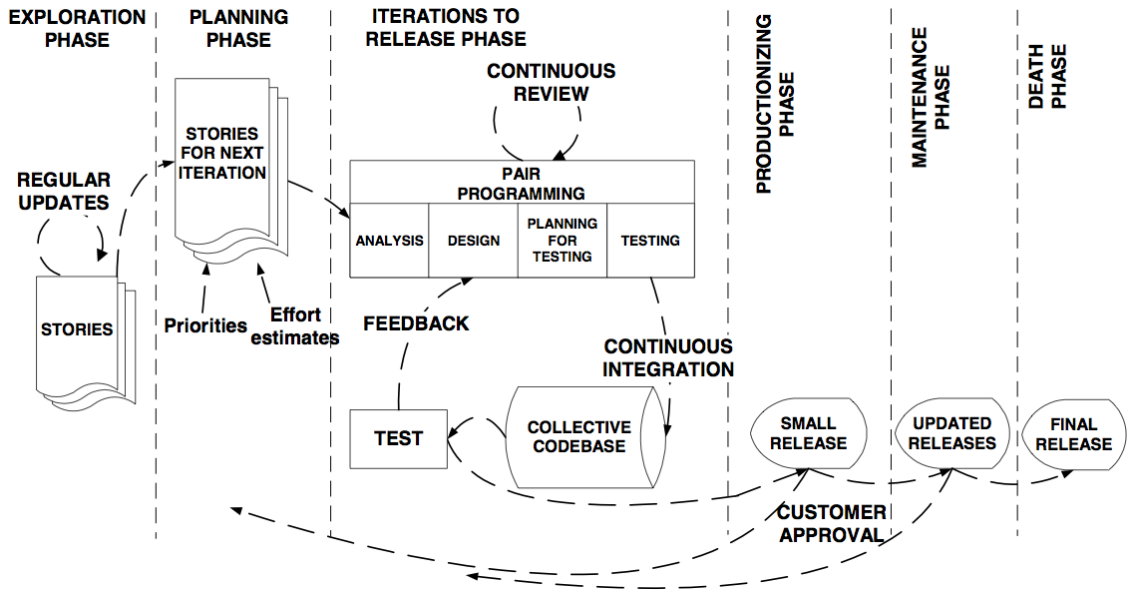

Figura 4.1: Ciclo de vida do eXtreme Programming [ASRW02]

Exploração

A fase de exploração tem dois objectivos principais:

- A escrita, por parte do cliente, de *user stories*, que definem os requisitos e as funcionalidades a implementar;
- A experimentação das tecnologias e ferramentas a utilizar no desenvolvimento do projecto, por forma a ambientar os elementos da equipa a essas mesmas tecnologias e ferramentas;

Nesta fase o cliente, no papel de utilizador final do produto a desenvolver, é responsável pela criação de *user stories*. *User stories* [Coh04] são um dos principais artefactos do XP na especificação de requisitos. Uma *user story* define sucintamente e de forma abstracta uma funcionalidade/requisito a implementar, contendo apenas a informação necessária para que os programadores sejam capazes de estimar de forma razoável o tempo necessário para realizar essa mesma implementação. Este artefacto do XP é concretizado recorrendo à utilização de *story cards*. A figura 4.2 representa a definição de uma *user story* recorrendo à utilização de *story cards*.

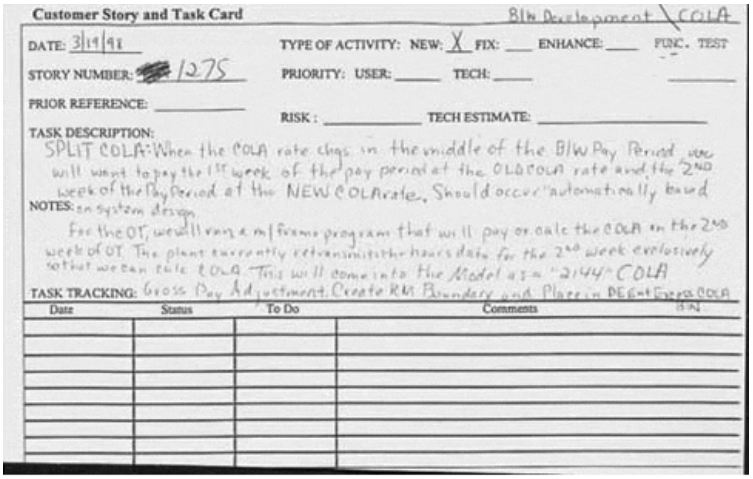

Figura 4.2: Exemplo de uma *story card* [BA04]

No que diz respeito ao segundo objectivo desta fase, os elementos da equipa devem, durante o período desta fase, explorar as tecnologias a utilizar no desenvolvimento do produto e alternativas à arquitectura a utilizar. Isto é efectuado através da criação de um sistema rudimentar, semelhante ao que será utilizado no produto, implementado de formas

diferentes por mais que um par de programadores (com recurso ao *pair programming*). Desta forma, novas ideias e conceitos surgem de forma natural e sem grande esforço. Mais uma vez é possível identificar no XP uma das características dos MDAs - o pensamento conjunto.

Adicionalmente, devem ser testados os níveis de desempenho das tecnologias a utilizar. Embora não seja possível medir com exactidão os níveis de *performance* das tecnologias a utilizar, recorrendo a uma proto-arquitectura, desenvolvida nesta fase, é possível inferir resultados e obter estimativas. No entanto é necessário ter em consideração o tempo dispendido na exploração das tecnologias e possíveis arquitecturas. Caso o tempo necessário para experimentar uma determinada tecnologia ultrapasse o prazo de uma semana, essa tecnologia deverá ser considerada como crítica ou inviável [BA04].

A fase de exploração termina quando for recolhido material (i.e., *user stories*), em quantidade e qualidade suficientes para a criação de uma primeira *release* e que permita aos elementos da equipa estimar o esforço necessário para implementar as funcionalidades a incluir nessa mesma release.

Planificação

A fase de planificação diz apenas respeito à próxima *release* do produto a implementar. Nesta fase, o plano do projecto para a próxima iteração é acordado entre o cliente e a equipa do projecto. Durante esta fase são também acordados os prazos a cumprir e são criadas prioridades para as *user stories*, definidas pelo cliente.

Iterações para a *release*

Nesta fase o prazo acordado para a criação de uma versão final do produto é dividido, igualmente, em iterações com uma duração de uma a quatro semanas. Na primeira iteração, o objectivo é a criação de uma primeira *release* que defina a arquitectura, ou esqueleto da arquitectura, sobre a qual novas funcionalidades serão implementadas. Assim, é sugerido que a escolha das funcionalidades a implementar numa primeira iteração "obriguem" à criação dessa mesma arquitectura [BA04]. Durante as iterações seguintes o cliente fica responsável por seleccionar um conjunto de *user stories* a implementar.

Em paralelo com a implementação das funcionalidades, a integrar na *release*, são criados pelo cliente e implementados pela equipa de desenvolvimento os casos de teste a

aplicar. O objectivo é a execução desses mesmos testes cada vez que uma nova funcionalidade é implementada e integrada na *release*.

Durante esta fase é necessário manter uma constante atenção a desvios no plano do projecto. Estes desvios podem obrigar a que, durante uma iteração desta fase, *user stories* sejam adicionadas ou removidas.

No final desta fase do projecto, deverá ser possível a criação de uma *release* que integre todas as funcionalidades, *user stories*, requeridas pelo cliente na fase de planificação. Desta forma é possível ao cliente acompanhar o estado de evolução do produto e analisar se o mesmo corresponde às suas necessidades. Deverá também existir um conjunto de testes, possíveis de replicar, que cobrem todas essas mesmas funcionalidades. No final desta fase é possível colocar a *release* em produção.

Produção

Esta fase encontra-se intimamente relacionada com o valor apresentado anteriormente e que aponta o *feedback* frequente e atempado como factor fundamental, para o sucesso de um projecto de software. Nesta fase é efectuada a análise da *performance* do sistema que é posteriormente apresentada e entregue ao cliente, juntamente com a *release* criada na fase anterior. Os resultados são analisados pelo cliente e a *release* é revista. Na fase de produção, devido à interacção e necessidade de uma avaliação por parte do cliente, a equipa deverá estar preparada para a implementação de novos testes de aceitação e *performance* de forma a comprovar a validade da *release*.

Adicionalmente pode surgir a necessidade de efectuar pequenas alterações às funcionalidades que integram a *release*. No entanto estas alterações devem ser efectuadas de forma ponderada devido ao impacto que podem causar nesta fase do projecto. Estas alterações serão efectuadas na fase seguinte do projecto - fase de manutenção. Caso não seja possível efectuar essas mesmas alterações durante a iteração em curso estas deverão ser remetidas para as iterações seguintes, dependendo dos objectivos, orçamento e predisposição por parte do cliente.

Manutenção

Nesta fase serão efectuadas as alterações propostas pelo cliente, na fase anterior do projecto. As alterações nesta fase do projecto são críticas no sentido em que pode comprometer toda a arquitectura. Assim, é necessária uma maior atenção ao código

implementado/re-implementado e garantir a execução com sucesso dos casos de teste implementados até ao momento. O código correspondente a novas alterações ou funcionalidades implementadas nesta fase do projecto deve ser, após a execução com sucesso dos casos de teste associados, imediatamente colocado em produção.

No final desta fase será entregue ao cliente uma actualização da *release*.

Morte do projecto

A morte do projecto, ou finalização, ocorre quando não existem mais *user stories* a implementar e quando factores como a *performance*, resiliência e estabilidade do sistema foram aprovados pelo cliente. Nesta fase é criada toda a documentação relativa ao projecto requerida pelo cliente. Esta documentação deverá incluir uma descrição simplificada da arquitectura implementada e das funcionalidades que integram o produto entregue.

4.1.3 Cargos e funções

Como em qualquer tipo de actividade efectuada por um grupo de indivíduos, também nos métodos de desenvolvimento se torna imprescindível a definição de um conjunto de cargos e funções, especificando qual o âmbito de cada uma. Por este motivo, e contrastando um pouco com a flexibilidade advogada pelos MDAs, todos estes métodos de desenvolvimento definem um grupo de cargos estruturado e específico, que devem ser atribuídos a um conjunto de indivíduos qualificados. A existência de cargos e funções a desempenhar num método de desenvolvimento não tem como objectivo uma simples organização lógica ou estrutural mas sim possibilitar a atribuição de responsabilidades, específicas a cada um dos diferentes cargos, e a aceitação dessas mesmas responsabilidades pelas pessoas que os executam.

Nesta secção do documento serão apresentados os cargos propostos no XP bem como as responsabilidades de quem os executa. Embora possam existir semelhanças entre os cargos definidos no XP e noutros métodos de desenvolvimento, pelas características inovadoras com que este método aborda o desenvolvimento de software, torna-se necessário que as pessoas que desempenham esses mesmos cargos possuam um conjunto de características e qualificações adicionais.

Programador

Como elemento principal de um projecto de software, o programador é responsável pela tradução dos requisitos a implementar numa linguagem de programação. Tendo por base

o conceito *test-driven development*, torna-se necessária a criação de testes unitários e a manutenção dos mesmos. A criação e implementação desses testes unitários é também da responsabilidade dos programadores.

Um dos valores sobre os quais assenta o XP é a simplicidade. Simplicidade na concepção da solução e implementação dessa mesma solução. No XP, as decisões de implementação ficam a cargo dos técnicos e pessoas especializadas envolvidas no projecto - os programadores. Assim, torna-se necessário que estes sejam capazes de simplificar o código implementado sem que isso comprometa a sua qualidade.

Como em qualquer tipo de método de desenvolvimento de software, tradicional ou ágil, a capacidade e níveis de conhecimento técnico da pessoa que desempenha este cargo são qualidades indispensáveis. No entanto, nos MDAs é dada grande importância à comunicação, outro dos valores do XP, e a falta dela foi já apontada como uma das principais razões que leva ao insucesso num projecto de software. Desta forma, uma boa capacidade de comunicação é uma qualidade fundamental num programador do XP. Adicionalmente, a utilização de práticas como o *pair programming* obrigam a que os programadores sejam capazes de trabalhar em conjunto, partilhando funções e responsabilidades, e auxiliar os restantes membros da equipa. Outra das práticas do XP que apela às capacidades dos programadores é o *refactoring*. O *refactoring* só é possível de aplicar caso os programadores possuam excelentes conhecimentos técnicos, sejam capazes de efectuar ajustes ao código sem comprometer as funcionalidades já implementadas e testadas e sejam críticos quanto às suas próprias decisões de implementação.

Cliente

O cliente é responsável pela criação de *user stories* que, como foi já mencionado, representam as funcionalidades e requisitos a serem implementados pelos programadores. Geralmente, nos projectos de software, o cliente sabe o que é necessário implementar e o programador sabe como o implementar. Adicionalmente, cabe ao cliente a tarefa de escrever os testes funcionais e de aceitação que serão executados com o objectivo de validar as funcionalidades implementadas. Desta forma, o cliente deve ser capaz de identificar as funcionalidades a testar e evitar a criação de testes redundantes.

Tester

O *tester* num projecto do XP tem como responsabilidade auxiliar o cliente na escrita dos testes funcionais e de aceitação e manter o ambiente de testes do projecto. Este elemento

da equipa deve executar regularmente os testes já implementados, por forma a manter a consistência do produto e garantir que o *refactoring* ao código e implementação de novas funcionalidades não comprometem as que foram já implementadas.

Os resultados da execução dos testes devem ser publicados e partilhados com todos os membros da equipa. O cliente, como membro integrante da equipa do projecto deve ter também acesso a esta informação.

Tracker

O *tracker* é responsável por recolher informação relativa à evolução do projecto e efectuar um parecer sobre as estimativas e esforços necessários para implementar cada uma das *user stories* definidas pelo cliente. A estimativa do esforço necessário continua a pertencer ao programador, no entanto esta deve e pode ser ajustada pelo *tracker* com o objectivo de tornar mais exactas essas mesmas estimativas. Consequentemente, o *tracker* deve ter vasta experiência em projectos de software.

É da responsabilidade do *tracker* manter o registo dos resultados dos testes efectuados. Devem também ser registados todos os erros e problemas que surgem durante o projecto, bem como as pessoas responsáveis por efectuar as correcções e a informação sobre os testes que são criados para testar essas mesmas correcções.

No que diz respeito às suas capacidades, um *tracker* deve ser, como já foi referido, capaz de efectuar estimativas o mais precisas possível. Adicionalmente, deve ser capaz de recolher a informação necessária sobre o projecto, sem perturbar excessivamente o respectivo processo de desenvolvimento. Deve ser capaz de inquirir os membros da equipa e transmitir informação sobre o decorrer do projecto, sem exercer uma pressão que leve os programadores a omitir factos e possíveis problemas.

Treinador

O treinador é o elemento da equipa responsável pelo projecto e por orientar os restantes elementos da equipa durante o ciclo de vida do XP. Todos os elementos de uma equipa do XP devem ter, no mínimo, conhecimento sobre as fases específicas do método de desenvolvimento. O treinador deve possuir conhecimentos pormenorizados sobre o método, que práticas podem mitigar um conjunto de problemas encontrados, de que forma é utilizado o XP por outras equipas, quais os valores e princípios do XP e de que forma estes se en-

contram relacionados com a tentativa de resolver os problemas que existem actualmente nos projectos de software [BA04].

Consultor

É possível que, em determinados projectos, os elementos da equipa não possuam os conhecimentos técnicos necessários sobre uma determinada matéria. Nestas situações é necessário integrar na equipa um profissional que detenha esse mesmo conhecimento. Este profissional desempenha, no XP, o papel de consultor do projecto. Desta forma, cabe ao consultor orientar e auxiliar a equipa do projecto na resolução de problemas específicos.

Big Boss

O *big boss* é o gestor de um projecto XP e responsável por providenciar e alocar todos os recursos necessários. Este deve acompanhar a evolução de todo o projecto e é responsável por, caso seja necessário, intervir de forma a assegurar o sucesso desse mesmo projecto.

4.2 Scrum

Em 1986, Takeuchi e Nonaka realizam um estudo, publicado na *Harvard Business Review*, denominado *The New Product Development Game* [TN86]. Neste estudo é constatado que obtiveram melhores resultados os projectos onde foram utilizadas equipas multi-disciplinares[1], com um número reduzido de elementos [Sut06], e auto-organizadas. Tendo por base as ideias e conceitos apresentados por Takeuchi e Nonaka, Jeff Sutherland criou em 1993 um novo método de desenvolvimento de produtos que foi, 2 anos depois, formalizado e adaptado para o desenvolvimento de software, sob a designação de Scrum [Sch95]. O Scrum é um dos poucos MDAs cuja aplicação tem vindo a ser tentada em projectos com maiores dimensões, que aqueles a que os MDAs são tradicionalmente aplicados [BT03].

4.2.1 Práticas

Nas secções seguintes serão apresentadas as práticas introduzidas pelo Scrum. Estas práticas, à semelhança das práticas introduzidas pelo XP, têm como objectivo facilitar e promover a criação eficiente de software.

[1] Deriva do original *cross-functional*

Product Backlog

O primeiro passo num projecto do Scrum é a reunião de todas as funcionalidades e requisitos a implementar numa lista criada para o efeito, a lista de *product backlog*. Cabe ao *product owner* a criação e manutenção dessa mesma lista. Para cada elemento que integra a lista de *product backlog* é definida uma prioridade e estimado o esforço necessário para o executar ou implementar. As prioridades associadas a cada elemento são definidas pelo *product owner* e devem reflectir o valor de negócio que cada elemento representa para o cliente. As estimativas do esforço associado a cada elemento da lista de *product backlog* ficam a cargo do *product owner* e da *scrum team*.

A lista de *product backlog* é continuamente actualizada e gerida pelo *product owner* de modo a reflectir alterações que possam surgir, novas ideias a implementar, problemas de índole técnica e actualizações nas estimativas efectuadas pelos programadores. No entanto, a gestão e manutenção eficiente da lista de *product backlog* apenas é possível através do *feedback* por parte da *scrum team*. A comunicação aparece assim, mais uma vez, como um factor preponderante no sucesso de um projecto de software.

Estimativa de Esforço

Estimativa de esforço é o nome dado ao processo iterativo durante o qual o esforço para cada elemento da lista de *product backlog* é iterativamente estimado de forma a tornar-se cada vez mais preciso. Este processo tem em consideração a informação que vai sendo disponibilizada ao longo do projecto.

Sprint

O *sprint* é considerada a parte ágil do Scrum e consiste num ciclo de desenvolvimento durante o qual a *scrum team* se organiza de forma a produzir um novo incremento que será integrado numa nova versão do produto, criada no final de cada *sprint*. Cada *sprint* tem uma duração de duas a quatro semanas, sendo que este período pode ser ajustado de acordo com as diversas variáveis de um projecto de software.

Cada *sprint* tem início com uma reunião de planificação do *sprint*, durante a qual são acordados os objectivos para o *sprint* e seleccionados os elementos da lista de *product backlog* que serão implementados neste ciclo de desenvolvimento. Durante um *sprint* são realizadas diversas reuniões diárias, *sprint daily meetings*.

No final de um *sprint* é realizada uma reunião de revisão do *sprint* na qual os resultados e a nova *release*, integrando agora as novas funcionalidades e requisitos implementados, são apresentados aos clientes, utilizadores e ao *product owner*. Após um *sprint* deve ser também efectuada uma reunião de retrospectiva, com o objectivo de identificar problemas ou impedimentos que possam ter surgido e definir formas de os mitigar.

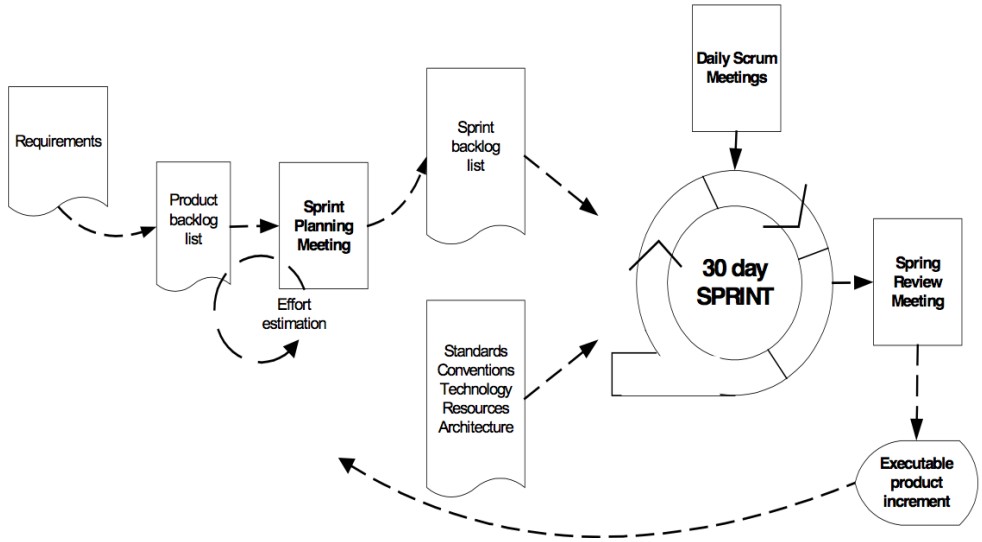

Figura 4.3: *Sprint* [ASRW02]

Reunião Diária

Estas reuniões são organizadas diariamente, durante cada *sprint*, com o objectivo de analisar a progressão do projecto, identificando o trabalho realizado desde a última reunião e o trabalho a realizar até à próxima. Estas reuniões servem também para garantir que cada elemento da *scrum team* está ciente do trabalho que se encontra a ser realizado pelos restantes membros. Cada reunião diária deve ser realizada a um horário fixo e pré-acordado e devem ter uma duração aproximada de quinze minutos.

Reunião de Planificação do *Sprint*

A reunião de planificação do *sprint* é organizada pelo *scrum master* e é dividida em duas fases. Durante a primeira fase da reunião, o *scrum master*, o cliente, o *product owner* e a *scrum team* seleccionam um conjunto de funcionalidades e requisitos, da lista de *product backlog*, que serão implementados durante o ciclo de desenvolvimento. Desta forma são definidos os objectivos globais para o *sprint*. A escolha destes elementos da lista de

product backlog deve ter em consideração a prioridade dos mesmos, permitindo assim que os elementos com maior valor de negócio para o cliente sejam implementados nas fases iniciais do projecto.

Na fase seguinte da reunião de planificação do *sprint* participam apenas o *scrum master* e a *scrum team*. Nesta fase, os elementos da lista de *product backlog* seleccionados na fase anterior são decompostos em tarefas, a serem realizadas pelos elementos da equipa. Essas tarefas integram a lista de *sprint backlog*. A cada uma dessas tarefas é também associada uma prioridade, sendo esta definida pela *scrum team*. É com base neste nível de prioridade que a *scrum team* planeia o *sprint* e as tarefas que cada elemento da equipa se compromete a executar.

Sprint Backlog

A lista de *sprint backlog* é constituída pelo conjunto de actividades a serem realizadas no decorrer de um *sprint*. Cada actividade ou conjunto de actividades representa um elemento da lista de *product backlog* que deve ser implementado de forma a integrar a *release* criada no final do *sprint*.

Reunião de Revisão do *Sprint*

No final de cada *sprint* é realizada uma reunião de revisão do *sprint*. Nesta reunião participam todos os intervenientes no projecto, sendo efectuada uma demonstração das funcionalidades que integram a nova *release*. Esta reunião não tem duração máxima e deve durar o tempo necessário para exemplificar e explicar as novas funcionalidades a todos os intervenientes. Durante a reunião devem ser colocadas as dúvidas que possam surgir e dado o *feedback*, por parte do cliente, quanto à qualidade e ao cumprimento dos objectivos propostos para o *sprint*.

Reunião de Retrospectiva do *Sprint*

A reunião de retrospectiva do *sprint* funciona à semelhança das típicas *leassons learned*, prática existente, na teoria, em qualquer projecto de software. Estas reuniões são realizadas no final de cada *sprint*, após a reunião de revisão do *sprint*. O seu objectivo é promover a discussão entre os elementos da equipa do projecto, sobre problemas com os quais a equipa possa ter sido confrontada e de que forma estes podem ser colmatados em *sprints* futuros.

4.2.2 Ciclo de vida

O ciclo de vida do Scrum, representado pela figura 4.4, é constituído por três fases: 1) *Pre-Game*, 2) Desenvolvimento e 3) *Post-Game*. Cada uma destas fases será seguidamente apresentada.

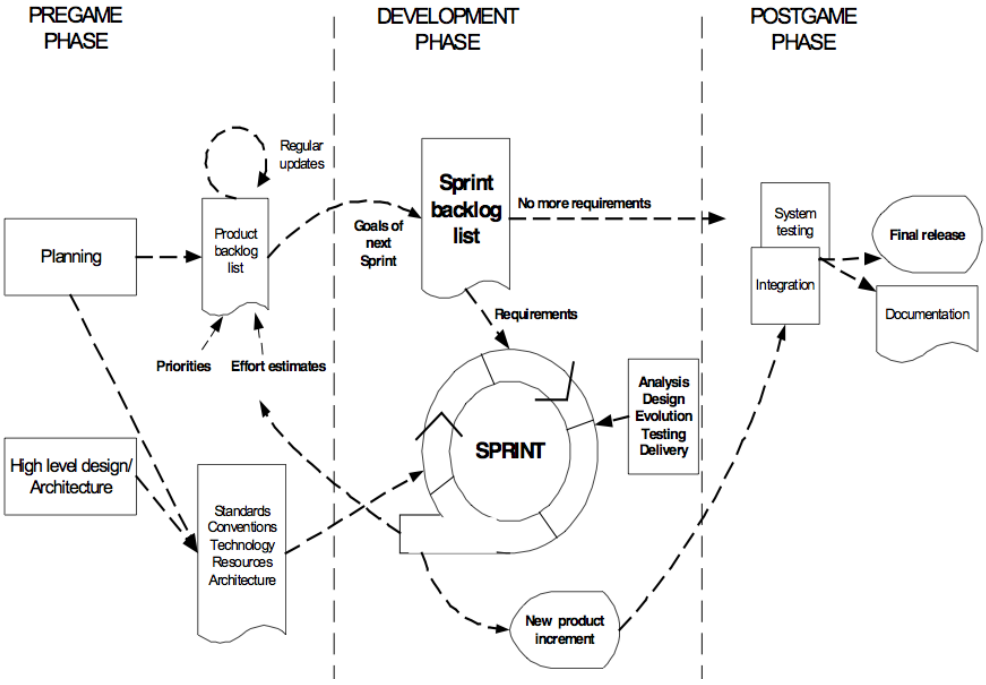

Figura 4.4: Ciclo de vida do Scrum [ASRW02]

Fase de *Pre-Game*

A fase de *pre-game* é constituída por duas sub-fases, a **planificação** e o **desenho de alto-nível da arquitectura**, cada uma com objectivos distintos:

- A fase de **planificação** tem por objectivo a definição de prazos, previsão de custos associados ao projecto, definição da equipa do projecto, selecção das ferramentas a utilizar e a avaliação e controlo do risco associado ao projecto. Nesta fase é também efectuado o levantamento de requisitos que vão integrar a lista de *product backlog*. Os elementos que constituem esta lista podem ser definidos pelo cliente, departamentos de *marketing*, departamentos de suporte ao cliente ou até mesmo da própria equipa de desenvolvimento, e descrevem os requisitos funcionais, não funcionais, de sistema ou produto. Cada requisito da lista de *product backlog* tem

associado um nível de prioridade. A gestão da lista e das prioridades associadas a cada entrada é gerida pelo *product owner* do projecto. A lista de *product backlog* é continuamente actualizada, adicionando ou removendo elementos, acrescentando informação mais detalhada aos elementos já existentes, modificando prioridades e actualizando o esforço associado à implementação dos elementos que a constituem.

- Na fase de **desenho de alto-nível da arquitectura** deve ser analisada e comceptualizada a arquitectura que irá servir de base ao desenvolvimento do produto a implementar. Esta concepção é baseada nos elementos que integram, até ao momento, a lista de *product backlog*. Caso o projecto não seja a criação de um novo produto mas a modificação ou actualização de um sistema já existente, esta fase serve para analisar o impacto causado pela implementação dos elementos da lista de *product backlog* na arquitectura existente. No final desta fase é realizada uma reunião de revisão para analisar as propostas apresentadas e seleccionar a arquitectura a utilizar.

Fase de Desenvolvimento

No Scrum, a fase de desenvolvimento é tratada como uma "caixa-negra" onde o imprevisível é expectável [ASRW02]. As diversas variáveis de um projecto, tais como prazos, qualidade, requisitos, recursos, tecnologias utilizadas e práticas de desenvolvimento, são observadas e controladas através de um conjunto de práticas introduzidas pelo Scrum durante os *sprints* (representado em detalhe pela figura 4.3). Na fase de desenvolvimento, o produto é implementado em *sprints* que correspondem a pequenos ciclos de desenvolvimento. Cada *sprint* inclui as fases tradicionais do desenvolvimento de software: análise e gestão de requisitos, implementação, testes e entrega do produto. Um *sprint* tem a duração média de uma semana a um mês, podendo existir entre três a oito *sprints* antes que o produto a implementar se encontre pronto para distribuição. No final da fase de desenvolvimento, as funcionalidades implementadas, durante o ciclo actual, são integradas numa *release* e é iniciada uma nova iteração do Scrum com a revisão da lista de *product backlog*. A passagem para a fase de *post-game* é realizada quando os objectivos acordados para o projecto forem alcançados com sucesso.

Fase de Post-Game

A fase de *post-game* diz respeito à preparação e criação de uma *release* final para entrega ao cliente. Esta fase ocorre quando, durante um projecto Scrum, todas as variáveis (e.g., prazos, requisitos, âmbito do projecto, qualidade e orçamento) analisadas e acordadas foram cumpridas com sucesso. Nesta fase são realizados testes de integração, testes

de sistema e criados os manuais de utilizador e material para formação. Nesta fase é comunicada a finalização do projecto a todos os intervenientes.

4.2.3 Cargos e funções

Nesta secção serão apresentados os cargos existentes no Scrum e as responsabilidades de quem os executa. O Scrum apresenta quatro cargos principais e todas as responsabilidades de gestão do projecto se encontram divididas entre eles: 1) *scrum master*, 2) *product owner*, 3) *scrum team* e 4) cliente.

Scrum Master

O *scrum master* é um dos mais importantes elementos da equipa do Scrum e deve efectuar tudo o que estiver ao seu alcance para auxiliar a equipa do projecto a terminar o projecto com sucesso. O seu papel é o de um líder, sendo este o elemento que detém o mais vasto conhecimento do Scrum, práticas utilizadas, artefactos produzidos, funções a serem desempenhadas por cada elemento da equipa, princípios e valores dos MDAs. É da responsabilidade do *scrum master* assistir a *scrum team* e o *product owner* das seguintes formas [Sch04]:

- Remover qualquer barreira existente entre o desenvolvimento e o *product owner*, possibilitando assim um maior contacto entre este e a *scrum team*;
- Auxiliar o *product owner* a maximizar o retorno sobre investimento (ROI)[2];
- Auxiliar a *scrum team*, promovendo a criatividade e a pro-actividade;
- Melhorar, sempre que possível, a produtividade e o desempenho da equipa de desenvolvimento (*scrum team*);
- Melhorar continuamente as práticas de engenharia e ferramentas utilizadas durante o projecto;
- Gerir a informação relativa ao desempenho e progresso da equipa do projecto, tornando disponível essa mesma informação a todos os intervenientes.

Pelo facto de ser um cargo transversal a todo o projecto, o *scrum master* deve possuir conhecimentos de Engenharia de Software, em particular gestão de projectos, concepção de arquitecturas e teste.

[2]Do inglês *Return of Investment*, representa o índice financeiro que mede o retorno de determinado investimento realizado e contabilizado em meses nos quais ele será amortizado para começar a gerar lucros.

Product Owner

O *product owner* é responsável por representar os interesses de todos os intervenientes no projecto. O *product owner* cria as bases para o início do projecto reunindo os requisitos inicias, definindo os objectivos para o ROI e os planos para as diversas *releases*. O *product owner* é também responsável pela criação e manutenção da lista de *product backlog*, assegurando que possuem uma maior prioridade os elementos da lista que representam maior valor de negócio para o cliente. O *product owner* participa também, em conjunto com a *scrum team*, na estimativa do esforço necessário para implementar cada elemento da lista de *product backlog*. O elemento que irá desempenhar este cargo é seleccionado em conjunto pelo *scrum master* e pelo cliente.

Scrum Team

A *scrum team* é a equipa de desenvolvimento responsável pela implementação do produto final, sendo constituída por 5 a 10 elementos incluindo analistas, programadores e *testers*. Em projectos de maiores dimensões onde a equipa do projecto possui um maior número de elementos, devem ser criadas várias equipas, de menores dimensões, onde cada equipa fica responsável por determinado aspecto do projecto. No Scrum, a equipa de desenvolvimento deve ser multi-disciplinar, possuindo todos os conhecimentos necessários para terminar um projecto, e auto-organizadas, com elevado grau de autonomia e responsável pelas suas próprias decisões [DB07].

A *scrum team* deve interagir continuamente com o *product owner*, fornecendo *feedback* relativo à estimativa de prazos, possíveis problemas técnicos e soluções e alternativas na concepção da arquitectura e na codificação do produto a desenvolver.

Cliente

O cliente participa nas tarefas relacionadas com a lista de *product backlog*, auxiliando o *product owner* na atribuição de prioridades. Adicionalmente o cliente é responsável por fornecer *feedback* sobre as *releases* apresentadas na várias reuniões de revisão, realizadas no final de cada *sprint*.

Capítulo 5

Análise e Classificação do XP e Scrum

Neste capítulo é descrita a análise comparativa realizada a um grupo de MDAs. Os MDAs seleccionados para este estudo foram o XP e o Scrum. A escolha destes métodos prende-se com o facto de estes serem os mais referidos na literatura [Gre01, BT03, MR05, ASRW02], mas principalmente por serem apontados como sendo os mais utilizados actualmente em empresas de software [Con06, VN07, Ser07, PHS+08]. Outra das razões que levou à escolha destes dois métodos para integrar o grupo em análise é a diferença existente entre o *focos* de cada um dos métodos - o XP é orientado para a parte de implementação de um projecto de software, sendo que o Scrum apresenta práticas e conceitos orientados para a parte de gestão dos projectos de software.

5.1 O Método de análise utilizado

O estudo comparativo de métodos de desenvolvimento revela-se uma tarefa difícil de concretizar. Os motivos que contribuem para essa dificuldade são:

- A dificuldade em encontrar contextos similares de aplicação dos métodos de desenvolvimento analisados (antes e depois da aplicação do método);
- O facto de não existirem dois projectos de software iguais (motivo directamente relacionado com o anterior);
- A dificuldade de proceder a uma validação científica das ideias propostas;
- O facto dessa comparação ser sempre influenciada pelas experiências ou intuições do(s) autor(es) da comparação.

Com o objectivo de reduzir esta dificuldade, em estudos semelhantes [SO92, ASRW02] foi utilizado um método de comparação "quase formal" (*quasiformal*), proposto por Henk G. Sol em *A Feature Analysis of Information Systems Design Methodologies: Methodological Considerations* [Sol83], o qual será também utilizado nesta dissertação. Neste artigo, Sol apresenta cinco abordagens possíveis para uma comparação "quase formal":

1. Descrever o método de desenvolvimento ideal e utilizá-lo como base de comparação para os métodos de desenvolvimento analisados;
2. Reunir um conjunto de atributos relevantes e utilizá-los como base de comparação para os métodos de desenvolvimento analisados;
3. Formular uma hipótese sobre os requisitos de um método de desenvolvimento e derivar os resultados a partir de evidências empíricas;
4. Definir uma meta-linguagem como meio de representação e um conjunto de critérios utilizados para descrever qualquer método de desenvolvimento;
5. Relacionar os atributos de cada método com um conjunto de problemas específicos.

Na análise realizada optou-se por utilizar a segunda abordagem apresentada. No que se refere aos atributos utilizados nessa análise, optou-se por seleccionar um subconjunto de quatro das onze Áreas de Conhecimento (ACs) definidas no SWEBOK: 1) Requisitos de Software, 2) Construção de Software, 3) Teste do Software e 4) Gestão da Engenharia de Software. O SWEBOK (*http://www.swebok.org/overview/*) é um projecto conjunto da *Software Engineering Coordinating Committee* e da *IEEE Computer Society*, com o objectivo de reunir e uniformizar as actividades e conceitos inerentes à Engenharia de Software e existentes em projectos de software. A comparação dos dois métodos tendo por base este subconjunto de ACs tem como objectivo identificar *se* e *de que forma* estes são abordados pelos MDAs analisados. Pretende-se desta forma inferir sobre o grau de abrangência do método ao conjunto de ACs seleccionado. Neste caso, um maior grau de abrangência do MDA identifica-o como sendo mais completo. A limitação deste estudo a um subconjunto de quatro das onze ACs definidas no SWEBOK prende-se com a limitação temporal para a realização da presente dissertação de mestrado.

Adicionalmente foi também seleccionado um quinto atributo, com o qual se pretende identificar a relação existente entre os princípios apresentados no manifesto ágil e as práticas introduzidas por cada um dos MDAs analisados. O objectivo da comparação entre os MDAs analisados sob este atributo é a análise dos respectivos graus de agilidade.

A tabela 5.1 resume os atributos seleccionados.

Atributo	***Descrição, sub-atributos e princípios***
Requisitos de Software	Identifica de que forma o método analisado aborda os requisitos num projecto de software, nomeadamente no que se refere aos seguintes sub-atributos: • Fundamentos de Requisitos de Software; • Processo de Requisitos; • Levantamento de Requisitos; • Análise de Requisitos; • Validação de Requisitos;
Construção de Software	Identifica de que forma o método analisado aborda a implementação de software, nomeadamente no que se refere aos seguintes sub-atributos: • Minimização da Complexidade; • Capacidade de Resposta a Possíveis Alterações; • Implementação Orientada à Verificação; • Utilização de *Standars*.
Teste do Software	Identifica de que forma o método analisado efectua a validação do produto desenvolvido e a abordagem adoptada para os testes, nomeadamente no que se refere aos seguintes sub-atributos: • Fundamentos de Testes do Software; • Níveis de Teste; • Medição de Métricas Associadas aos Testes; • Processo de Testes.
Gestão da Engenharia de Software	Identifica de que forma o método analisado aborda a gestão de um projecto de software, nomeadamente no que se refere aos seguintes sub-atributos: • Iniciação e Definição do Âmbito; • Planificação do Projecto de Software; • Aprovação e Execução do Plano do Projecto; • Finalização do Projecto; • Revisão e Avaliação.
Relação Princípios Ágeis - Práticas Advogadas	Identifica a relação existente entre os princípios apresentados no manifesto ágil e as práticas introduzidas pelo MDA.

Tabela 5.1: Conjunto de atributos e respectivos sub-atributos ou princípios para análise comparativa de MDAs

5.2 Classificação sob os atributos seleccionados

Nesta secção do documento será apresentada a classificação do XP e do Scrum sob os atributos seleccionados. Como foi referido, a abordagem utilizada consiste na selecção de um conjunto de atributos a ser utilizado como base de comparação. Estes atributos são constituídos por um subconjunto das ACs definidas no SWEBOK e pela relação entre os princípios definidos no manifesto ágil e as práticas advogadas pelos MDAs analisados.

Para classificar a existência de práticas ou conceitos nos métodos analisados que suportem o subconjunto das ACs seleccionadas e para classificar a relação entre os princípios

do manifesto ágil e as práticas, introduzidas por cada um dos métodos, será utilizado o critério apresentado na tabela 5.2.

NS	Não satisfaz	O método não apresenta práticas ou conceitos que suportam o sub-atributo ou princípio analisado
SP	Satisfaz Parcialmente	O método advoga práticas ou conceitos que suportam o sub-atributo ou princípio analisado, no entanto alguns aspectos deste não são contemplados
S	Satisfaz	O método advoga práticas ou conceitos que suportam totalmente o sub-atributo ou princípio analisado

Tabela 5.2: Critério de classificação na análise comparativa efectuada

A dificuldade existente em quantificar o suporte, ou não, de determinado sub-atributo ou princípio por parte das práticas apresentadas por cada método analisado, levou à escolha de um sistema qualitativo de classificação. Embora simples, na opinião do autor desta dissertação, um sistema qualitativo de classificação satisfaz as necessidades deste estudo. A comparação dos MDAs, segundo os atributos apresentados na tabela 5.1, será apresentada nas secções seguintes deste capítulo.

5.2.1 Requisitos de Software

A AC Requisitos de Software encontra-se relacionada com o levantamento, a especificação, a análise e a validação dos requisitos de um projecto de software. É do conhecimento geral, dentro da indústria das TICs, que os projectos de software são bastante vulneráveis quando as actividades anteriormente apresentadas são negligenciadas [ABDM04], o que torna relevante uma análise dos métodos de desenvolvimento sob este atributo.

No atributo de Requisitos de Software, pretende-se analisar de que forma os métodos analisados abordam os requisitos de um projecto de software, em particular os sub-atributos apresentados na tabela 5.3.

A análise de cada um dos métodos sob este atributo será apresentada nas duas secções seguintes do documento.

eXtreme Programming

O XP apresenta um conjunto de práticas e conceitos que suportam a definição, levantamento e análise de requisitos, como por exemplo a prática de *Planning Game*, a fase

Sub-atributo	*Descrição*
1. Fundamentos de Requisitos de Software	Refere-se à existência de uma definição clara do que são requisitos de software, efectuando a distinção entre os diferentes tipos de requisitos (produto vs. sistema, funcionais vs. não-funcionais, ...)
2. Processo de Requisitos	Refere-se à existência de um processo para levantamento, especificação, análise e validação de requisitos, especificando os intervenientes, considerações práticas, modelos de gestão de requisitos, ...
3. Levantamento de Requisitos	Refere-se à recolha e levantamento dos requisitos, especificando o meio utilizado e a origem desses mesmos requisitos
4. Análise de Requisitos	Refere-se em particular à existência de conceitos, práticas e técnicas para detecção e resolução de conflitos e classificação de requisitos
5. Validação de Requisitos	Refere-se à existência de conceitos, práticas ou técnicas que permitam a posterior validação dos requisitos implementados

Tabela 5.3: Sub-atributos do atributo Requisitos de Software

de exploração e a definição de *user stories*. Adicionalmente, o XP identifica todos os intervenientes e artefactos envolvidos na definição, levantamento, análise e implementação dos requisitos de software. No entanto, o XP não efectua qualquer tipo de distinção, clara e objectiva, sobre os diferentes tipos de requisitos existentes (produto vs. sistema, funcionais vs. não-funcionais). Assim, embora apresentando os conceitos básicos, as actividades e os intervenientes relacionados com os requisitos de software, por não apresentar uma distinção clara dos diferentes tipos de requisitos o XP **satisfaz parcialmente** o sub-atributo *1. Fundamentos de Requisitos de Software*.

Analisando o ciclo de vida do XP sob a perspectiva de levantamento, especificação, análise e validação de requisitos, este tem início com a fase de exploração onde o cliente reúne um conjunto de *user stories*, que representam os requisitos e funcionalidades a implementar. Como já foi mencionado, os requisitos num projecto de software são muitas vezes incertos, pelo que o conjunto de *user stories* definido pelo cliente não é estático, estando assim sujeito a eventuais alterações. De seguida, cabe aos programadores estimar o tempo e o esforço necessários para a implementação de cada uma das *user stories*. Nesta fase é da responsabilidade dos programadores identificar as *user stories* cuja implementação excede o período de duas semanas, sendo de seguida negociado com o cliente a possível divisão de cada uma dessas *user stories*. Caso exista alguma ambiguidade ou dúvida relativamente às *user stories*, a presença *on-site* do cliente facilita a comunicação e permite a resolução rápida do problema. Após a estimativa de esforço, o cliente selecciona as *user stories* a serem implementadas e integradas na próxima *release*. Cada *user storie* a ser implementada é dividida num conjunto de actividades ou tarefas, a serem executadas por cada par de programadores, e que incluem a implementação de testes unitários para verificação do código implementado. Após a implementação e integração de uma funcionalidade numa *release*, o cliente define um conjunto de testes funcionais e

de aceitação com o objectivo de validar as funcionalidades implementadas e assegurar a correcta implementação dos requisitos definidos por esse mesmo cliente. O processo acima descrito, permite concluir que o XP **satisfaz** o sub-atributo *2. Processo de Requisitos.*

O principal meio de levantamento de requisitos, utilizado no XP, é a criação de *user stories* por parte do cliente, sendo este incluido como membro da equipa do projecto. O cliente trabalha, em conjunto com os restantes membros da equipa, na criação de *user stories*, na definição de testes funcionais e na aceitação e na prioritização de requisitos. No XP, os requisitos não são representados através de um documento monolítico, mas sim através de uma colecção de *user stories*, testes funcionais e aceitação e testes unitários que são definidos de forma incremental [Dun01]. Analisar se a forma apresentada no XP para levantamento de requisitos é ou não válida ou eficiente não se encontra no âmbito deste trabalho. No entanto, o simples facto do XP apresentar um meio para levantamento de requisitos **satisfaz** o sub-atributo *3. Levantamento de Requisitos.*

No contexto deste estudo, define-se como análise de requisitos a detecção e resolução de conflitos entre requisitos e a classificação dos mesmos segundo uma métrica pré-definida. No XP não são visíveis quaisquer técnicas para detecção e resolução de conflitos nos requisitos especificados. Isto deve-se ao facto do XP assumir a existência de apenas um cliente ou representante deste. Esta assumpção, associada ao facto da existência de um cliente *on-site*, faz com que não exista a necessidade deste tipo de técnicas. No entanto, é necessário referir que a inexistência deste tipo de técnicas dificulta a aplicação do XP nos casos em que:

- o cliente ou representante deste não possui as qualificações ou capacidades necessárias para especificar correctamente os requisitos;
- os requisitos, especificados por cada um dos vários clientes envolvidos no projecto, podem ser mutuamente exclusivos.

No que se refere à classificação de requisitos, como foi já referido, o cliente é responsável por atribuir prioridades a cada *user story* por ele definida, de forma a garantir que são integradas nas primeiras *releases* aquelas que são mais críticas para o negócio. Desta forma é possível minimizar os riscos do projecto já que, caso ocorra alguma alteração ao plano, é sempre possível a entrega de um produto que integra as *user stories* de maior valor de negócio para o cliente. Devido à inexistência de técnicas para detecção e resolução de conflitos entre requisitos, é possível afirmar que o XP **satisfaz parcialmente** o sub-atributo

4. Análise de Requisitos.

No XP, a validação dos requisitos é concretizada através da definição de um conjunto de testes funcionais e de aceitação por parte do cliente.

A síntese da análise efectuada ao XP sob o atributo Requisitos de Software é apresentada na tabela 5.4.

Sub-atributo	***Classificação***	***Características do XP que satisfazem o sub-atributo***
Fundamentos de Requisitos de Software	**SP**	• Definição de conceitos relacionados com os requisitos de software; • Não é efectuada uma distinção entre os diferentes tipos de requisitos.
Processo de Requisitos	**S**	• Definição de um processo para levantamento, especificação, análise e validação de requisitos, definindo explicitamente os intervenientes e actividades a realizar.
Levantamento de Requisitos	**S**	• Cliente *On-site*; • *User Stories*.
Análise de Requisitos	**SP**	• Classificação de requisitos através da definição de prioridades; • Inexistência de técnicas para detecção e resolução de conflitos entre requisitos.
Validação de Requisitos	**S**	• Testes funcionais e de aceitação, escritos pelo cliente.

Tabela 5.4: Síntese da análise efectuada ao XP sob o atributo Requisitos de Software

Scrum

No Scrum são visíveis os conceitos básicos relacionados com os requisitos de software. O Scrum define um conjunto de fases e práticas que suportam o levantamento, a especificação e a análise de requisitos. Adicionalmente, os elementos que integram a lista de *product backlog* podem surgir dos mais variados *stakeholders* envolvidos no projecto (e.g. o cliente ou programadores) e podem representar funcionalidades do sistema ou do produto, requisitos funcionais e não-funcionais, actividades de investigação ou erros e defeitos a corrigir. Assim, é também possível identificar uma distinção entre os diferentes tipos de requisitos. Desta forma, o Scrum **satisfaz** o sub-atributo *1. Fundamentos de Requisitos de Software.*

O processo de requisitos do Scrum tem início com a fase de *pre-game*, onde é elaborada a lista de *product backlog*. Durante esta fase, o cliente em colaboração com o *product owner* é responsável pela atribuição de prioridades a cada elemento da lista. No início de cada *sprint* é realizada uma reunião de planificação do *sprint* onde é seleccionado um subconjunto dos elementos da lista de *product backlog* para ser implementado durante o *sprint*. Cada um desses elementos é dividido, pela *scrum team*, em actividades ou tarefas a realizar, as quais constituem a lista de *sprint backlog*. Desta forma, o Scrum apresenta também um processo definido para o levantamento, especificação, análise e validação de requisitos. Assim, é possível afirmar que o Scrum **satisfaz** o sub-atributo *2. Processo de Requisitos*.

No Scrum, os principais meios utilizados no levantamento de requisitos são o contacto directo com o cliente na fase de *pre-game* (planificação) e a possibilidade de qualquer uma das várias entidades envolvidas no projecto adicionar um novo elemento à lista de *product backlog*. Assim, o Scrum **satisfaz** o sub-atributo *3. Levantamento de Requisitos*.

No que diz à respeito à classificação de requisitos, ambos os métodos analisados sugerem a criação de prioridades associadas aos requisitos, *user stories* (XP) e itens da lista *product backlog* (Scrum). No caso do Scrum, é o *product owner*, e não o cliente, o responsável por definir as prioridades dos requisitos que integram a lista de *product backlog* e que irão mais tarde integrar a lista de *sprint backlog*. A lista de *sprint backlog* é um artefacto exclusivo da *scrum team*, que é a única responsável pela gestão da mesma. Como tal, as prioridades associadas a cada elemento que integra a lista de *sprint backlog* são definidas pela *scrum team* e não pelo *product owner*. Adicionalmente e com objectivo de estabilizar ligeiramente as fases referentes à implementação, não é permitida qualquer alteração aos itens da lista de *sprint backlog* durante o decorrer de um *sprint*. No que se refere à existência de práticas e técnicas para detecção e resolução de conflitos, à semelhança do XP, também o Scrum não apresenta qualquer técnica ou prática com este fim. Desta forma, o Scrum **satisfaz parcialmente** o sub-atributo *4. Análise de Requisitos*.

Por não apresentar quaisquer conceitos, práticas ou técnicas relacionadas com a validação dos requisitos definidos pelos diferentes intervenientes, o Scrum **não satisfaz** o sub-atributo *5. Validação de Requisitos*.

A síntese da análise efectuada ao Scrum sob o atributo Requisitos de Software é apre-

sentada na tabela 5.5.

Sub-atributo	*Classificação*	*Características do Scrum que satisfazem o sub-atributo*
Fundamentos de Requisitos de Software	**S**	• Definição de conceitos relacionados com os requisitos de software; • Distinção entre os diferentes tipos de requisitos.
Processo de Requisitos	**S**	• Definição de um processo para levantamento, especificação, análise e validação de requisitos, definindo explicitamente os intervenientes e actividades a realizar.
Levantamento de Requisitos	**S**	• Contacto directo com o cliente em fases iniciais do projecto; • Possibilidade de qualquer entidade envolvida no projecto adicionar um novo elemento à lista de *product backlog*.
Análise de Requisitos	**SP**	• Classificação de requisitos através da definição de prioridades; • Inexistência de técnicas para detecção e resolução de conflitos entre requisitos.
Validação de Requisitos	**NS**	-

Tabela 5.5: Síntese da análise efectuada ao Scrum sob o atributo Requisitos de Software

5.2.2 Construção de Software

A AC de *Construção de Software* refere-se aos detalhes de implementação num projecto de software, através da utilização de técnicas de verificação, validação, integração e capacidade de resposta a possíveis alterações ao código já implementado [ABDM04]. Esta AC é constituída por três sub-ACs, das quais foi seleccionada para esta análise a sub-área referente aos fundamentos da construção de software. Esta sub-área abrange quatro sub-atributos da implementação de software: *1. Minimização da Complexidade*, *2. Antecipação de Possíveis Alterações*, *3. Implementação Orientada à Verificação* e *4. Utilização de Standars*. Pelas características inerentes aos MDAs, no que se refere à capacidade de aceitar as alterações face à tentativa de prever as alterações, o sub-atributo *2. Antecipação de Possíveis Alterações* foi substituído por *2. Capacidade de Resposta a Possíveis Alterações*.

Assim, uma análise sob esta perspectiva pretende identificar de que forma os métodos analisados, XP e Scrum, abordam os detalhes de implementação num projecto de software, em particular os sub-atributos apresentados na tabela 5.6.

Sub-atributo	***Descrição***
1. Minimização da Complexidade	Refere-se à utilização de técnicas e práticas com o objectivo de minimizar a complexidade do código implementado
2. Capacidade de Resposta a Possíveis Alterações	Refere-se à utilização de técnicas e práticas com o objectivo de promover a capacidade de resposta a possíveis alterações
3. Implementação Orientada à Verificação	Refere-se à utilização de técnicas e práticas na implementação do código que permitam a detecção rápida de falhas ou erros durante a escrita do código e durante a execução de testes
4. Utilização de *Standars*	Refere-se à utilização de standards durante a implementação do código

Tabela 5.6: Sub-atributos do atributo Construção de Software

A análise de cada um dos métodos sob este atributo será apresentada nas duas secções seguintes do documento.

eXtreme Programming

No caso do XP é possível afirmar que este se apresenta orientado para a fase de implementação, sugerindo explicitamente um conjunto de práticas que devem ser aplicadas durante essa mesma fase. Estas práticas têm como objectivo a criação rápida de código simples, funcional, com elevada qualidade e capaz de ser rapidamente ajustado a possíveis alterações de requisitos.

Analisando pormenorizadamente o XP sob o atributo de Construção de Software, podemos verificar que **satisfaz** o sub-atributo *1. Minimização da Complexidade* não só pelas práticas introduzidas pelo método mas também pelos valores sobre os quais este assenta. Como referido na secção 4.1, o XP assenta sobre quatro valores sendo um deles a simplicidade, por oposição à complexidade. No que se refere às práticas introduzidas pelo XP, existem essencialmente duas que abordam este sub-atributo: a prática de concepção simples e a de *refactoring*. A prática de concepção simples tem por objectivo impedir a introdução de complexidade desnecessária no código implementado, facilitando assim a manutenção e compreensão do mesmo. Beck descreve a forma correcta de concepção e implementação no XP como sendo aquela que [BA04]:

- Permite a execução com sucesso de todos os testes;
- Não possui lógica duplicada;
- Reflecte todas as intenções dos programadores;
- Possui o menor número de classes e métodos implementados.

A prática de *refactoring* permite (1) minimizar a complexidade existente no código implementado, através da remoção de complexidade desnecessária, com o objectivo de tornar o código mais legível, (2) eliminar redundância e funcionalidades duplicadas e (3) aumentar a modularidade e escalabilidade, promovendo desta forma a reutilização de código. Assim, é possível afirmar que o XP **satisfaz** o sub-atributo *1. Minimização da Complexidade.*

O sub-atributo *2. Capacidade de Resposta a Possíveis Alterações* reflecte a agilidade e resposta rápida a alterações, que são duas das principais características dos MDAs. Através de um conjunto de práticas, como a posse colectiva de código, o *refactoring* e a integração contínua, o XP maximiza a sua capacidade de responder a alterações que possam surgir nos requisitos do projecto, pelo que **satisfaz** totalmente este sub-atributo. Em particular, a prática de posse colectiva de código permite que qualquer elemento da equipa do projecto realize alterações ao código, resultantes da alteração de requisitos. Assim, evitam-se esperas desnecessárias na alteração ao código. A prática de *refactoring*, influencia indirectamente a capacidade de resposta a alterações no decorrer de um projecto de software. Como foi dito anteriormente, através de *refactoring* pretende-se a clarividência do código, a eliminação de redundância e a maximização da modularidade. A tarefa de efectuar alterações ao código implementado é significativamente facilitada quando este é legível e simples. A prática de integração contínua suporta a capacidade de resposta a possíveis alterações, garantindo a integridade do produto e que alterações efectuadas ao código existente e a implementação de novas funcionalidades não comprometem o trabalho já realizado.

A existência de práticas como o *test-driven development* e *pair programming* no XP permite a criação de uma infra-estrutura sólida de suporte à construção e à verificação de software, constituída por um conjunto de testes cuja implementação acompanha ou precede a própria implementação do código. A conjugação destas duas práticas permite a detecção rápida de falhas ou erros durante a escrita do código e durante a execução de testes, pelo que **satisfaz** o sub-atributo *3. Implementação Orientada à Verificação.*

A utilização de normas, durante a implementação de um projecto de software, tem por objectivo enquadrar e orientar todos os membros da equipa de desenvolvimento, facilitando a compreensão do código por parte de outros elementos que não o autor. A utilização desta prática deve abranger não só a utilização de normas na implementação (*coding standards*), prática utilizada no XP, mas também a utilização de normas na comunicação, na criação de interfaces, na criação de documentação, na concepção da arquitec-

tura e no levantamento de requisitos. Uma vez que a prática de *coding standards* utilizada no XP apenas se refere à implementação, pode-se dizer que o XP **satisfaz parcialmente** o sub-atributo *4. Utilização de* Standars.

A síntese da análise efectuada ao XP sob o tópico Construção de Software, é apresentada na tabela 5.7.

Sub-atributo	***Classificação***	***Características do XP que satisfazem o sub-atributo***
Minimização da Complexidade	**S**	• Valores do XP; • Concepção Simples; • *Refactoring*.
Capacidade de Resposta a Possíveis Alterações	**S**	• Posse Colectiva de Código; • *Refactoring*; • Integração Contínua.
Implementação Orientada à Verificação	**S**	• *Test-Driven Development*; • Pair Programming.
Utilização de *Standars*	**SP**	• Utilização de *Coding Standards*.

Tabela 5.7: Síntese da análise efectuada ao XP sob o atributo Construção de Software

Scrum

Pelas práticas do Scrum apresentadas anteriormente, no capítulo 4, é possível afirmar que este se apresenta como um método para gestão de projectos de software, podendo ser também aplicado a outros tipos de projectos. O Scrum surge com o objectivo melhorar os conceitos de Engenharia de Software utilizados na gestão de projectos, que visam identificar problemas e impedimentos no processo de desenvolvimento e nas práticas de implementação utilizadas. Assim, práticas de implementação não são contemplados no Scrum, deixando ao critério das suas equipas multi-disciplinares a escolha destas mesmas práticas a utilizar na concepção da arquitectura, na implementação, na configuração do produto, nos testes e na integração. Assim, o Scrum não contempla nenhum dos sub-atributos da AC Construção de Software.

A sintese da análise efectuada ao Scrum sob o atributo apresentado, Construção de Software, é apresentada na tabela 5.8.

Sub-atributo	*Classificação*	*Características do Scrum que satisfazem o sub-atributo*
Minimização da Complexidade	**NS**	-
Capacidade de Resposta a Possíveis Alterações	**NS**	-
Implementação Orientada à Verificação	**NS**	-
Utilização de *Standars*	**NS**	-

Tabela 5.8: Síntese da análise efectuada ao Scrum sob o atributo Construção de Software

5.2.3 Teste do Software

O teste do software é uma actividade desempenhada com o objectivo de validar e verificar a qualidade do produto, identificando possíveis defeitos ou problemas no código implementado. O teste do software consiste na verificação e na validação do comportamento de um programa, através da comparação entre o resultado obtido da execução de um conjunto finito de testes e o comportamento esperado [ABDM04]. Sobre o atributo de Teste do Software, pretende-se analisar que soluções e práticas o XP e o Scrum apresentam para a definição e realização de testes, em particular para os sub-atributos apresentados na tabela 5.9.

Sub-atributo	*Descrição*
1. Fundamentos de Testes do Software	Refere-se à existência de uma definição clara dos conceitos relacionados com o teste do software e à existência de relações explicitas entre as actividades de teste e as restante actividades do método analisado
2. Níveis de Teste	Refere-se à existência de diversos níveis de teste, definindo explicitamente os objectivos e âmbito dos diferentes testes, e.g. testes unitários, testes de integração, testes de *performance*, testes de aceitação, ...
3. Medição de Métricas Associadas aos Testes	Refere-se à realização de medições e análise das métricas resultantes da execução dos testes
4. Processo de Testes	Refere-se às considerações práticas relacionadas com os testes e à existência de um processo definido para a escrita e execução dos mesmos

Tabela 5.9: Sub-atributos do atributo Teste do Software

A análise de cada um dos métodos segundo este atributo será apresentada nas duas secções seguintes do documento.

eXtreme Programming

O XP define explicitamente conceitos relacionados com o teste de software, objectivos e âmbito dos diversos níveis de teste, actividades e funções específicas de cada elemento envolvido no teste de software, como pode ser verificado através de uma análise ao ciclo de vida do XP (ver secção 4.1.2). A análise ao ciclo de vida do XP permite também verificar que são definidas fases específicas para a execução de testes, existindo assim uma

relação explicita entre as actividades de teste e as restantes actividades do XP. Assim, é possível afirmar que o XP **satisfaz** o sub-atributo *1. Fundamentos de Testes do Software*, apresentando conceitos e terminologias específicas do teste de software e especificando a relação entre as actividades relacionadas com o teste de software e as restantes actividades do projecto.

Em [BA04], um dos criadores do XP identifica o seguinte conjunto de testes a executar durante o ciclo de vida de um projecto XP, cada um com âmbitos e objectivos distintos:

Testes de Unitários são escritos pelos programadores antes ou durante a implementação e têm por objectivo a verificação do código implementado;

Testes de Integração são executados no final de uma iteração para criação de um nova *release*, aquando da integração das novas funcionalidades com o código já existente. Estes testes suportam a prática de integração contínua;

Testes de Desempenho são executados com o objectivo de levar ao limite o desempenho de um sistema, como por exemplo a simulação do pior caso de utilização no que se refere à quantidade de informação processada;

Testes Funcionais e de Aceitação são escritos pelo cliente, em conjunto com o *tester*, e têm como objectivo a validação das funcionalidades implementadas.

Adicionalmente, Beck [BA04] refere que outros tipos de testes podem ser aplicados num projecto XP (e.g testes de usabilidade, compatibilidade, regressão), de acordo com as necessidades e especificidade do projecto. Desta forma é claramente visível a divisão dos testes em níveis distintos no âmbito e objectivo, pelo que o XP **satisfaz** o sub-atributo *2. Níveis de Teste.*

O XP também **satisfaz** o sub-atributo que se refere à realização de medições e análise das métricas resultantes da execução dos testes, já que o XP define um cargo específico para a gestão e execução dos testes - o *Tester* (ver secção 4.1.3). Para além de auxiliar o cliente na escrita dos testes funcionais e de aceitação e execução periódica dos testes criados, este elemento da equipa do projecto é também responsável por reunir e analisar os resultados da execução dos testes, partilhando esta informação com os restantes membros da equipa. Estes dados são de extrema importância para avaliar a progressão do projecto.

O XP define um processo explícito para a escrita e a execução dos diferentes tipos de testes. Durante a fase de exploração do ciclo de vida do XP (ver secção 4.1.2), o cliente

é responsável pela escrita de *user stories* que definem o conjunto de funcionalidades a implementar. Em paralelo com a escrita das *user stories* deve ser definido um conjunto de testes funcionais ou de aceitação, com o objectivo de validar cada uma das *user stories* escritas pelo cliente, ficando a definição deste conjunto de testes a cargo do próprio cliente. Na prática a definição de testes funcionais e de aceitação pode-se revelar uma tarefa relativamente complexa. Desta forma, a definição de testes funcionais e de aceitação deve ser uma tarefa do cliente auxiliado pelo *tester*. Os testes unitários devem preceder e acompanhar a implementação das funcionalidades, segundo a filosofia do *test-driven development*. Segundo Beck, os programadores devem criar testes unitários à medida que novas funcionalidades são implementadas seguindo os seguintes critérios [BA04]:

- Quando a interface de um método é pouco clara, a implementação do teste deve preceder a implementação do método;
- Quando a interface do método é clara mas a implementação se revela mais complexa que o esperado, a implementação do teste deve preceder a implementação do método;
- Caso seja detectada uma situação pouco usual na qual o comportamento do código implementado deva permanecer inalterado, deve ser implementado um teste de forma a contemplar a situação detectada;
- Sempre que for encontrado um problema na implementação, este deve ser resolvido e um teste que isole esse mesmo problema deve ser implementado.

Um maior número de testes permite aumentar o grau de confiança dos programadores, sempre que seja necessário efectuar alterações ao código, e garantir que a alteração ou implementação de novas funcionalidades não compromete as funcionalidades implementadas anteriormente. A existência de um sistema automatizado de testes é recomendado e, dependendo do grau de criticidade, os testes devem abranger uma determinada percentagem do código implementado. Como referido na secção 4.1.1, uma funcionalidade implementada só se encontra 100% completa e pronta para integrar uma *release* quando todos os testes criados até a esse momento executam com sucesso. Os testes de integração são executados no final de cada iteração para criação de uma nova *release*, onde as novas funcionalidades implementadas são integradas e testes de regressão são executados. Caso algum dos testes executados falhe, a nova funcionalidade não é integrada na nova *release*. Na fase de produção, do ciclo de vida do XP, é executado um conjunto de testes de *performance*, cujos resultados são posteriormente entregues ao cliente para validação. O processo acima descrito define um conjunto de considerações práticas relacionadas com a escrita e execução de testes, num projecto XP. Assim, conclui-se que o XP **satisfaz** o

sub-atributo *4. Processo de Testes.*

A síntese da análise efectuada ao XP sob o atributo Teste do Software é apresentada na tabela 5.10.

Sub-atributo	***Classificação***	***Características do MDA que satisfazem o sub-atributo***
Fundamentos de Testes do Software	**S**	• Definição de conceitos relacionados com testes e actividades subjacentes; • Ciclo de vida com fases específicas de teste; • Definição de artefactos específicos para definição, execução e análise de testes.
Níveis de Teste	**S**	• Testes de Unitários; • Testes de Integração; • Testes de *Performance*; • Testes Funcionais e de Aceitação.
Medição de Métricas Associadas aos Testes	**S**	• Recolha e análise dos resultados da execução dos testes; • Disponibilização da análise dos resultados a todos os membros da equipa; • *Tester*.
Processo de Testes	**S**	• Processo de testes devidamente especificado; • Práticas relacionadas com a escrita e execução de testes devidamente detalhadas.

Tabela 5.10: Síntese da análise efectuada ao XP sob o atributo Teste do Software

Scrum

O Scrum não apresenta explicitamente nenhuma prática ou conceito que suporte a realização de testes unitários, de sistema ou de validação por parte do cliente. Analisando o ciclo de vida do Scrum é possível identificar que este contempla a realização de testes durante a implementação dos requisitos que integram a lista de *sprint backlog* e a realização de testes de sistema na fase de *post-game*, antes da entrega final de uma *release ao cliente*. No entanto, nenhuma consideração é feita quanto à forma como estes testes são realizados, delegando essas decisões para a equipa de cada projecto.

A sintese da análise efectuada ao Scrum sob o atributo Teste do Software é apresentada na tabela 5.11.

Sub-atributo	*Classificação*	*Características do MDA que satisfazem o sub-atributo*
Fundamentos de Testes do Software	**NS**	-
Níveis de Teste	**NS**	-
Medição de Métricas Associadas aos Testes	**NS**	-
Processo de Testes	**NS**	-

Tabela 5.11: Síntese da análise efectuada ao Scrum sob o atributo Teste do Software

5.2.4 Gestão da Engenharia de Software

A gestão da Engenharia de Software pode ser definida como a aplicação de todas as actividades de gestão - e.g. planificação, coordenação, monitorização, controlo, avaliação e controlo de riscos e comunicação com o cliente - com o objectivo de assegurar a criação e manutenção de software de forma sistemática e disciplinada [IEE90]. A AC apresentada diz respeito às actividades de gestão de um projecto de software. Sob este atributo pretende-se identificar de que forma cada um dos métodos analisados aborda os sub-atributos apresentados na tabela 5.12.

Sub-atributo	*Descrição*
1. Iniciação e Definição do Âmbito	Refere-se à existência de actividades que dão início a um projecto de software, as quais incluem a definição dos objectivos do projecto, a análise e negociação de requisitos, a análise do ROI e a identificação das necessidades de formação e de sub-contratados
2. Planificação do Projecto de Software	Refere-se à criação dos planos do projecto, a definição dos artefactos a entregar ao cliente, a análise de custos e esforço, a alocação de recursos e a avaliação e controlo de riscos
3. Aprovação e Execução do Plano do Projecto	Refere-se à aceitação do plano por parte de todos os intervenientes no projecto de software, a execução do plano do projecto e a definição do processo de monitorização e controlo do progresso do projecto
4. Finalização do Projecto	Refere-se à existência de actividades de finalização de um projecto de software, como a entrega do produto final ao cliente, a comunicação do fim do projecto a todos os intervenientes e a libertação dos recursos alocados
5. Revisão e Avaliação	Refere-se à determinação do grau de satisfação do cliente, avaliação do cumprimento dos requisitos, a avaliação da *performance* da equipa e o cumprimento dos objectivos

Tabela 5.12: Sub-atributos do atributo Gestão da Engenharia de Software

A análise de cada um dos métodos sob este atributo será apresentada nas duas secções seguintes do documento.

eXtreme Programming

A prática *planning game* do XP tem por objectivo a recolha de dados para permitir um início rápido do desenvolvimento. Durante a execução desta prática advoga-se o trabalho

conjunto entre o cliente e a equipa técnica, sendo cada uma das partes responsável pela definição do seguinte conjunto de métricas associadas ao projecto:

- O cliente é responsável por:
  - definir o **âmbito** do projecto;
  - definir as **prioridades** das *user stories* a implementar;
  - definir quais as ***user stories*** que devem integrar cada uma das *releases*;
  - definir as **datas e prazos** do projecto para lançamento de cada uma das versões.
- A equipa técnica é responsável por:
  - apresentar as **estimativas** de prazos;
  - apresentar ao cliente as **consequências** da utilização de uma ou outra tecnologia, após avaliar e controlar os riscos;
  - organizar a **atribuição de tarefas e alocação de recursos**;
  - definir e apresentar um **plano detalhado** do projecto ao cliente.

Questões como a aceitação do plano detalhado do projecto e a sua posterior execução são contempladas, ainda que de forma iterativa devido às características dos MDAs, nas fases de planificação e iterações para a *release* (figura 4.1). No que diz respeito à definição do processo de monitorização e controlo do progresso do projecto, o XP utiliza o software executável como métrica para avaliação desse mesmo progresso, definindo especificamente um elemento como responsável pela recolha da informação relativa à evolução do projecto - *tracker*. Assim, o XP **satisfaz** os sub-atributos:

1. Iniciação e Definição do Âmbito;
2. Planificação do Projecto de Software;
3. Aprovação e Execução do Plano do Projecto.

O XP **satisfaz** o sub-atributo *4. Finalização do Projecto* pela existência da fase de morte do projecto. Nesta fase do ciclo de vida do XP, como referido na secção 4.1.2, é entregue ao cliente o produto final juntamente com toda a documentação necessária e requerida por este.

Embora questões, como o cumprimento dos objectivos do projecto, a verificação e a validação das funcionalidades implementadas e a aceitação das mesmas por parte do cliente, sejam abordadas pelo XP, este não apresenta ou faz referência a qualquer tipo de

prática ou fase onde seja efectuada uma avaliação global do nível de desempenho da equipa ou do nível de satisfação do cliente. Assim, é possível afirmar que o XP **não satisfaz** o sub-atributo *5. Revisão e Avaliação.*

A síntese da análise efectuada ao XP sob o atributo Gestão da Engenharia de Software é apresentada na tabela 5.13.

Sub-atributo	***Classificação***	***Características do XP que satisfazem o sub-atributo***
Iniciação e Definição do Âmbito	**S**	• The Planning Game.
Planificação do Projecto de Software	**S**	• The Planning Game; • Fase de Planificação.
Aprovação e Execução do Plano do Projecto	**S**	• Fase de Planificação; • Fase de Iterações para a *release*; • Software executável como métrica para avaliação do progresso; • *Tracker*.
Finalização do Projecto	**S**	• Fase de Morte do Projecto.
Revisão e Avaliação	**NS**	-

Tabela 5.13: Síntese da análise efectuada ao XP sob o atributo Gestão da Engenharia de Software

Scrum

Embora o Scrum apresente o *product backlog* como artefacto fundamental na análise, definição e negociação de requisitos, questões como a definição dos objectivos do projecto, a análise do ROI e a identificação das necessidades de formação e sub-contratados não são abordadas. Adicionalmente, o Scrum define a equipa de desenvolvimento - *scrum team* - como sendo multi-disciplinar e possuindo todos os conhecimentos técnicos necessários para avançar com o projecto. Na prática, raras são as equipas de um projecto que possuem esta característica. Assim, o Scrum **satisfaz parcialmente** o sub-atributo *1. Iniciação e Definição do Âmbito.*

Um projecto Scrum tem início com a fase de *pre-game* (figura 4.4) que por sua vez é dividida em duas sub-fases, sendo uma delas a de planificação. Esta sub-fase inclui a definição de prazos, custos associados ao projecto, definição da equipa e ferramentas a utilizar e a avaliação e controlo de risco associado ao projecto. Assim, esta fase satisfaz o

sub-atributo *2. Planificação do Projecto de Software*. Adicionalmente, o Scrum introduz a prática de estimativa de esforço, consistindo num processo iterativo onde a estimativa do esforço necessário para implementar cada elemento da lista de *product backlog* é continuamente refinada. Complementando a planificação de alto-nível do projecto, realizada na fase de *pre-game*, o Scrum advoga também a realização de uma reunião de planificação do *sprint* com o objectivo de criar um plano do projecto agora com um âmbito mais reduzido e apenas aplicado ao *sprint*. Desta forma, o Scrum **satisfaz** o sub-atributo *2. Planificação do Projecto de Software*.

A *Aprovação e Execução do Plano do Projecto* é abordada no Scrum com a execução de *sprints*, após a aceitação do plano definido na reunião de planificação do *sprint*. Como processo de monitorização e controlo do progresso do projecto o Scrum utiliza, além do software executável, as listas de *product backlog* e *sprint backlog*. Estas contêm todas as tarefas ou funcionalidades a executar e implementar, respectivamente durante o projecto e durante um *sprint*. Adicionalmente, como referido na secção 4.2.1, durante a execução de um *sprint* são realizadas reuniões diárias com o objectivo de avaliar a progressão do projecto e analisar problemas que tenham surgido ou que possam eventualmente surgir. Desta forma, o Scrum **satisfaz** o sub-atributo *3. Aprovação e Execução do Plano do Projecto*.

O Scrum **satisfaz** o sub-atributo *4. Finalização do Projecto* de duas formas distintas e com âmbitos diferentes. Através da realização de uma reunião de revisão do *sprint* é apresentada ao cliente a *release* implementada durante um *sprint* e são executadas todas as tarefas de finalização desse mesmo *sprint*. No final da fase de *post-game* (figura 4.4) é entregue ao cliente o produto final.

A avaliação do grau de satisfação do cliente, do cumprimento dos requisitos, do desempenho da equipa e do cumprimento dos objectivos do projecto é realizada, embora apenas no âmbito de um *sprint*, durante a reunião de retrospectiva do *sprint*. Esta prática **satisfaz** o sub-atributo *5. Revisão e Avaliação*, embora deva ser tido em consideração que o objectivo não é avaliar o projecto globalmente mas antes cada *sprint* realizado.

A síntese da análise efectuada ao Scrum sob o atributo Gestão da Engenharia de Software é apresentada na tabela 5.14.

Sub-atributo	*Classificação*	*Características do Scrum que satisfazem o sub-atributo*
Iniciação e Definição do Âmbito	**SP**	• Product Backlog.
Planificação do Projecto de Software	**S**	• Fase de Pre-Game; • Estimativa de Esforço; • Reunião de Planificação do Sprint.
Aprovação e Execução do Plano do Projecto	**S**	• Sprint; • Software executável como métrica para avaliação do progresso; • Product Backlog; • Sprint Backlog; • Reuniões diárias.
Finalização do Projecto	**S**	• Reunião de Revisão do Sprint; • Fase de Post-Game.
Revisão e Avaliação	**S**	• Reunião de Retrospectiva do Sprint.

Tabela 5.14: Síntese da análise efectuada ao Scrum sob o atributo Gestão da Engenharia de Software

5.2.5 Relação Princípios Ágeis - Práticas Advogadas

Nesta secção será analisada a relação entre os princípios apresentados no manifesto ágil (*http://agilemanifesto.org/*) e as práticas advogadas por cada um dos MDAs analisados, XP e Scrum. O objectivo desta análise é identificar *se* e *de que forma* os princípios ágeis se reflectem nas práticas dos dois MDAs analisados.

De seguida são apresentados os resultados da análise do XP e Scrum sob o atributo apresentado, de forma tabular (Tabelas 5.15 e 5.16). Adicionalmente é apresentada uma pequena explicação do critério aplicado.

eXtreme Programming

Princípio 1 - A criação de pequenas *releases*, a intervalos regulares, permite a entrega contínua e atempada de software ao cliente. A prática *the planning game* exige uma relação estreita entre cliente e equipa técnica, delegando para aquele a responsabilidade da definição das *user stories* e das respectivas prioridades. Desta forma é possível garantir ao cliente a entrega de software com valor comercial e de negócio. A utilização conjunta destas duas práticas **satisfaz** totalmente o princípio 1, apresentado no manifesto ágil.

Princípio	*Classificação*	*Práticas*
1. A maior prioridade é a satisfação do cliente através da disponibilização atempada e contínua de software com valor	**S**	• Pequenas *releases* • The Planning Game
2. Devem-se aceitar as alterações de requisitos, mesmo quando isso acontece em fases tardias do desenvolvimento. Os MDAs utilizam as alterações para fornecer vantagem competitiva ao cliente	**S**	• Concepção simples • *Refactoring*
3. Deve-se disponibilizar software operacional com frequência, em intervalos que podem variar entre as duas semanas e os dois meses, embora com preferência pelos prazos mais curtos	**S**	• Pequenas *releases* • Integração Contínua
4. Os *stakeholders* e os programadores têm que trabalhar diariamente em conjunto ao longo do projecto	**S**	• Cliente *On-Site*
5. Os projectos devem ser executados por indivíduos motivados. A partir daqui, há que fornecer-lhes o ambiente e o suporte que precisam, além de confiar neles para a realização do trabalho	**S**	• The Planning Game • Posse Colectiva do Código • 40 horas semanais
6. O método mais eficiente e eficaz de transmitir informação a uma equipa de desenvolvimento e dentro da mesma é a conversação frente-a-frente	**S**	• *Pair Programming* • Cliente *On-Site* • Disposição do local de trabalho
7. O software operacional é a principal métrica de progresso	**S**	• Pequenas *releases* • Integração Contínua
8. Os MDAs promovem o desenvolvimento sustentado. Os patrocinadores, os especialistas em desenvolvimento e os utilizadores devem ser capazes de manter indefinidamente um ritmo constante	**SP**	• 40 horas semanais
9. A atenção contínua à excelência técnica permite aumentar a agilidade	**S**	• Testes • *Refactoring* • Utilização de *Coding Standards*
10. A simplicidade - enquanto arte de minimização da quantidade de trabalho por fazer - é essencial	**S**	• Concepção Simples • *Refactoring*
11. As melhores arquitecturas, os melhores requisitos e os melhores desenhos de arquiteturas emergem de equipas que se auto-organizam	**NS**	-
12. A intervalos de tempo regulares, a equipa deve reflectir sobre formas de se tornar mais eficaz, procedendo seguidamente a ajustes no seu comportamento com esse fim em mente	**NS**	-

Tabela 5.15: Relação princípios do manifesto - práticas do XP

Princípio 2 - O XP advoga que todos os componentes que constituem uma arquitectura e código que a implementa devem ser justificados como sendo imprescindíveis, não incluindo desta forma módulos, funcionalidades ou componentes numa tentativa de antecipar problemas futuros. Novas funcionalidades a serem implementadas devem ser incorporadas na arquitectura existente. Esta prática de concepção simples permite tornar o produto flexível a possíveis ajustes e alterações de requisitos em qualquer fase de um projecto. A utilização da prática de *refactoring* facilita a alteração a requisitos já implementados, garantindo a qualidade do código implementado. Desta forma o XP **satisfaz** totalmente o princípio 2 do manifesto ágil.

Princípio 3 - No XP é efectuada a divisão do plano e prazo acordados para a criação de uma versão final do produto em iterações com uma duração de uma a quatro semanas, no final das quais são entregues ao cliente *releases* funcionais que integram um pequeno conjunto de funcionalidades. Esta prática, associada a prática de integração contínua, permite a disponibilização frequente de software operacional ao cliente. Desta forma, o XP **satisfaz** totalmente o princípio 3 do manifesto ágil.

Princípio 4 - Um cliente, ou representante deste, *on-site* permite que este e os programadores realizem conversas informais, validem requisitos, participem em reuniões não agendadas e clarifiquem pontos pouco claros ou que possam suscitar dúvidas. A utilização desta prática permite que os *stakeholders* e os programadores trabalhem diariamente em conjunto, pelo que o XP **satisfaz** o princípio 4 do manifesto ágil.

Princípio 5 - A prática *the planning game* é uma actividade de colaboração entre todos os intervenientes no projecto, incluindo equipa técnica (programadores) e clientes. Cada uma das partes contribuí com uma perspectiva única sobre o produto a desenvolver. Assim, no XP, à semelhança dos restantes MDAs, a equipa técnica é tratada como elemento fundamental da equipa do projecto. A importância e confiança depositadas na equipa técnica do projecto confere um importante aspecto de motivação, assim como a posse colectiva de código. O facto de cada par de programadores ser responsável por todo código implementado e encorajado a efectuar quaisquer alterações necessárias, confere aos programadores a possibilidade de efectuarem decisões importantes para o sucesso do projecto, revelando ser também um factor de motivação. Assim, o XP **satisfaz** totalmente o princípio 5 do manifesto ágil.

Princípio 6 - Os MDAs apontam a comunicação frente-a-frente como sendo o método mais eficaz na transmissão de informação, como é descrito pelo princípio 6 do manifesto. Este método de comunicação permite colmatar os problemas típicos que

advêm da utilização de outros métodos de comunicação, como por exemplo a documentação. No caso do XP são introduzidas duas práticas que valorizam este método de comunicação, satisfazendo o princípio 6 - o *pair programming* e a existência de um cliente *on-site*.

A prática de *pair programming* resulta num dialogo contínuo entre dois programadores, trabalhando em conjunto numa mesma *workstation*. A existência de um cliente *on-site* permite que todos os aspectos do projecto possam ser discutidos frente-a-frente, com esse mesmo cliente. Desta forma é possível reduzir a ambiguidade e perda de informação na comunicação entre equipa do projecto e o cliente. Adicionalmente, o XP sugere ainda que a equipa do projecto trabalhe toda na mesma sala, de preferência num *open-space*[2], facilitando assim a interacção entre os membros da equipa. Embora não sendo uma prática do XP, esta característica suporta também o princípio 6. Assim, é possível afirmar que o XP **satisfaz** totalmente o princípio apresentado.

Princípio 7 - A criação de pequenas *releases* permite visualizar o estado actual do projecto, permitindo identificar o conjunto de requisitos que foram implementados até ao momento de criação da *release* ou o não cumprimento destes. A integração contínua garante que a implementação de novos requisitos, e posterior integração, não afecta o código anteriormente implementado. Assim, todos os requisitos e funcionalidades implementadas até um determinado momento reflectem o estado actual do progresso. Desta forma, o XP **satisfaz** totalmente o princípio 7 do manifesto.

Princípio 8 - A prática de 40 horas semanais indica que o trabalho extraordinário deve ser a excepção e não a regra. O objectivo é garantir um desenvolvimento sustentado, indo assim ao encontro deste princípio. Embora esta prática se apresente directamente relacionada com este princípio é da opinião do autor que a simples aplicação desta prática não garante um ritmo constante no desenvolvimento de um projecto de software. Assim, o XP **satisfaz parcialmente** o princípio 8.

Princípio 9 - As práticas relacionadas com a definição e a execução de testes, *refactoring* e a utilização de *coding standards* suportam o princípio 9. Todas estas práticas têm como objectivo garantir a consistência do que é implementado.

- A definição e a execução de testes têm como objectivo garantir a verificação e validação do produto;

[2]Nome que designa um local de trabalho caracterizado pela ausência de escritórios ou cubículos de trabalho. Este tipo de disposição do local de trabalho promove as relações inter-pessoais e a cooperação entre os funcionários.

- A aplicação de *refactoring* permite a remoção de ambiguidade, duplicação e a clarificação do código;
- A utilização de *coding standards* tem como objectivo a uniformização do código.

Esta contínua atenção à excelência técnica permite afirmar que o XP **satisfaz** o princípio 9.

Princípio 10 - Como foi já analisado na secção 5.2.2 deste documento, as práticas de concepção simples e *refactoring* suportam o princípio 10. Assim, o XP **satisfaz** o princípio 10.

Scrum

Princípio 1 - Como referido anteriormente, cada elemento da lista de *product backlog* tem associado uma prioridade. Essa prioridade, definida pelo *product owner*, deve reflectir o valor de negócio que cada elemento representa para o cliente e deve ser definido com o intuito de maximizar o ROI, garantindo assim a entrega de software com valor para o cliente. A característica iterativa do Scrum, comum a todos os MDAs, em conjunto com a realização de *sprints* permitem a disponibilização contínua e atempada (duas a quatro semanas) de software. Assim, o Scrum **satisfaz** o princípio 1 do manifesto ágil.

Princípio 2 - Embora durante a realização de um *sprint* não sejam permitidas alterações aos elementos que integram a lista de *sprint backlog*, em qualquer fase do projecto é permitido adicionar, remover ou alterar elementos da lista de *product backlog*. Alterações efectuadas a esta lista não têm impacto directo no *sprint* que se encontra em curso, a menos que estas inviabilizem ou tornem obsoleta a realização deste. Nestes casos o *sprint* é cancelado e uma nova reunião de planificação do *sprint* é realizada. A reunião de planificação do *sprint* tem assim um papel preponderante na aceitação de alterações durante um projecto de software, já que esta serve de base para a realização de cada *sprint*. Desta forma, o Scrum **satisfaz** o princípio 2 do manifesto.

Princípio 3 - Nos MDAs, o desenvolvimento de forma incremental permite disponibilizar frequentemente software operacional. No caso do Scrum esse ciclo de desenvolvimento, no final do qual é produzido um novo incremento que será entregue ao cliente software operacional sob a forma de uma nova *release* do produto, é concretizado através do *sprint*. No entanto, o Scrum não apresenta nenhuma prática que suporte a entrega de software operacional, apostando nas capacidades técnicas da

Princípio	***Classificação***	***Práticas***
1. A maior prioridade é a satisfação do cliente através da disponibilização atempada e contínua de software com valor	**S**	• Product Backlog • Sprint
2. Devem-se aceitar as alterações de requisitos, mesmo quando isso acontece em fases tardias do desenvolvimento. Os MDAs utilizam as alterações para fornecer vantagem competitiva ao cliente	**S**	• Reunião de planificação do sprint
3. Deve-se disponibilizar software operacional com frequência, em intervalos que podem variar entre as duas semanas e os dois meses, embora com preferência pelos prazos mais curtos	**SP**	• Sprint
4. Os *stakeholders* e os programadores têm que trabalhar diariamente em conjunto ao longo do projecto	**SP**	• Reunião de Planificação do Sprint • Product Backlog • Reunião de Revisão do Sprint
5. Os projectos devem ser executados por indivíduos motivados. A partir daqui, há que fornecer-lhes o ambiente e o suporte que precisam, além de confiar neles para a realização do trabalho	**NS**	-
6. O método mais eficiente e eficaz de transmitir informação a uma equipa de desenvolvimento e dentro da mesma é a conversação frente-a-frente	**S**	• Reunião de Planificação do Sprint • Reunião Diária • Reunião de Revisão do Sprint • Reunião de Retrospectiva do Sprint
7. O software operacional é a principal métrica de progresso	**S**	• Sprint • Reunião de Revisão do Sprint
8. Os MDAs promovem o desenvolvimento sustentado. Os patrocinadores, os especialistas em desenvolvimento e os utilizadores devem ser capazes de manter indefinidamente um ritmo constante	**NS**	-
9. A atenção contínua à excelência técnica permite aumentar a agilidade	**S**	• Reunião de Retrospectiva do Sprint
10. A simplicidade - enquanto arte de minimização da quantidade de trabalho por fazer - é essencial	**NS**	-
11. As melhores arquitecturas, os melhores requisitos e os melhores desenhos de arquitecturas emergem de equipas que se auto-organizam	**NS**	***NOTA:*** *Embora nenhuma prática do Scrum suporte o princípio 11, as características de uma equipa típica no Scrum satisfazem este princípio.*
12. A intervalos de tempo regulares, a equipa deve reflectir sobre formas de se tornar mais eficaz, procedendo seguidamente a ajustes no seu comportamento com esse fim em mente	**S**	• Reunião de Retrospectiva do Sprint

Tabela 5.16: Relação princípios do manifesto - práticas do Scrum

scrum team para garantir este aspecto do software. Desta forma, por não apresentar nenhuma prática que suporte a operacionalidade do software entregue, o Scrum **satisfaz parcialmente** o princípio 3.

Princípio 4 - O Scrum apresenta um conjunto de práticas que contam com a participação activa do cliente, durante um projecto. Exemplos disso são as práticas de reunião de planificação do *sprint*, gestão da lista de *product backlog* e a reunião de revisão do *sprint*. No entanto, o princípio 4 advoga um trabalho conjunto diário. Uma vez que a relação e o contacto com o cliente não é levado ao extremo, ao contrário do XP, na opinião do autor desta dissertação o Scrum **satisfaz parcialmente** o princípio 4.

Princípio 6 - Sendo um método de desenvolvimento para a gestão de projectos, o Scrum define um conjunto de práticas orientadas para a gestão e manutenção das diferentes variáveis de um projecto de software:

- Reunião de planificação do sprint;
- Reunião diária;
- Reunião de revisão do sprint;
- Reunião de retrospectiva do sprint.

Um factor comum a todas as práticas apresentadas é o facto de que toda a informação transmitida é realizada, em teoria, através de uma conversação frente-a-frente. Assim, o Scrum **satisfaz** o princípio 6.

Princípio 7 - Nos Scrum, a disponibilização de software operacional de forma incremental, no final de cada *sprint*, permite às diferentes entidades envolvidas no projecto ter uma percepção do estado actual e do progresso do mesmo. Assim, à semelhança do XP, também o Scrum utiliza o software operacional como principal métrica de progresso. A avaliação do estado e progresso do projecto ocorre no final de cada *sprint*, com a realização de uma reunião de revisão do *sprint*. Desta forma, a conjugação destas duas práticas **satisfaz** o princípio 7.

Princípios 9 e 12 - No final de cada *sprint* todos os elementos da equipa participam numa reunião de retrospectiva. O objectivo é identificar problemas e dificuldades que tenham surgido durante a realização do *sprint* e identificar possíveis soluções que impeçam a sua ocorrência em *sprints* futuros. Estas reuniões servem também para analisar o desempenho da equipa do projecto e identificar possíveis aspectos que possam ser melhorados. Esta prática permite um constante melhoramento do processo de desenvolvimento e da performance de cada elemento da equipa do projecto,

revelando uma atenção contínua à excelência técnica. Assim, o Scrum **satisfaz** os princípios 9 e 12.

5.3 Síntese dos resultados obtidos

Para o estudo realizado foram seleccionados cinco atributos que serviram de base de comparação entre o XP e o Scrum:

1. Requisitos de Software;
2. Construção de Software;
3. Teste do Software;
4. Gestão da Engenharia de Software;
5. Relação Princípios Ágeis - Práticas Advogadas.

Por forma a facilitar a análise, cada um dos atributos utilizados foi dividido num conjunto de sub-atributos. Os atributos 1) a 4) representam quatro das onze Áreas de Conhecimento presentes no SWEBOK, representando por isso os conceitos e características que deveriam estar, em teoria, presentes em todos os métodos de desenvolvimento. O último atributo representa a relação existente entre os princípios do manifesto ágil e as práticas advogadas pelo MDA analisado.

A quantificação dos resultados obtidos é dificultada pelo facto de não ser possível identificar com total precisão a maior ou menor relevância de cada um dos sub-atributos analisados para o sucesso de um projecto de software. Ainda assim, e assumindo que cada um dos sub-atributos analisados possui igual relevância, nesta secção do documento é apresentada uma síntese dos resultados obtidos.

Sob o atributo *Requisitos de Software* o gráfico a) da figura 5.1 revela que o XP **satisfaz** 60% e **satisfaz parcialmente** 40% dos sub-atributos analisados enquanto que o Scrum **satisfaz** 60%, **satisfaz parcialmente** 20% e **não satisfaz** 20% desses mesmos sub-atributos (gráfico b) da figura 5.1). Estas percentagens devem-se maioritariamente ao contacto estreito entre a equipa técnica do projecto e o cliente, visível no XP. A importância dada a esta relação reflecte-se no facto do XP abordar todos os sub-atributos de *Requisitos de Software*.

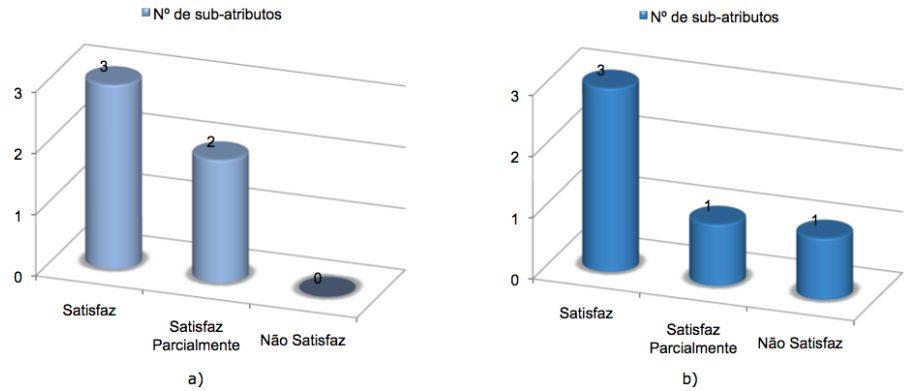

Figura 5.1: a) Resultados da análise ao XP sob o atributo Requisitos de Software; b) Resultados da análise ao Scrum sob o atributo Requisitos de Software

No que se refere ao atributo *Construção de Software*, o gráfico a) da figura 5.2 mostra que o XP **satisfaz** 75% dos sub-atributos analisados e **satisfaz parcialmente** 25% desses mesmos sub-atributos, em contraste com o Scrum que **não satisfaz** nenhum dos sub-atributos analisados (gráfico b) da figura 5.2). O facto do Scrum não sugerir explicitamente nenhuma prática direccionada para a implementação num projecto de software, distancia-o do XP sob o atributo *Construção de Software*.

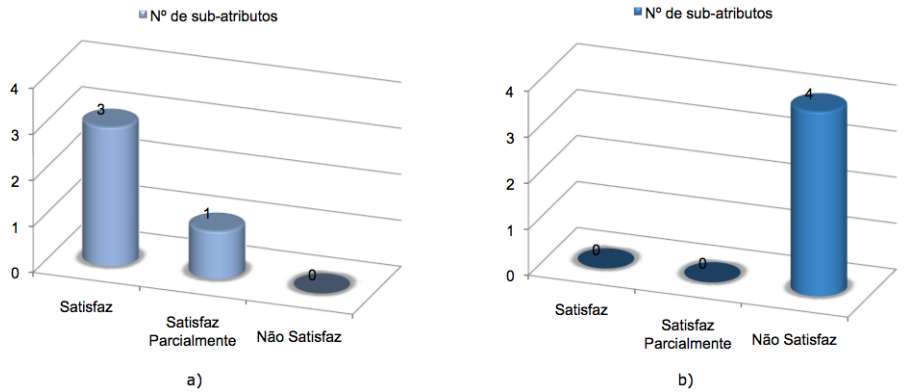

Figura 5.2: a) Resultados da análise ao XP sob o atributo Construção de Software; b) Resultados da análise ao Scrum sob o atributo Construção de Software

O XP caracteriza-se como sendo o MDA que mais ênfase dá aos testes efectuados. Tendo a sua base no *test-driven development*, este coloca os testes nas fundações do desenvolvimento. Os resultados obtidos da análise sob o atributo *Teste do Software* reflectem esta característica do XP e a importância que este coloca na criação e execução de testes

durante um projecto de software. O gráfico a) da figura 5.3 sintetiza os resultados obtidos, onde se pode observar que o XP **satisfaz** 100% dos sub-atributos de *Teste do Software*. No que se refere ao Scrum, este **não satisfaz** nenhum desses sub-atributos (gráfico b) da figura 5.3).

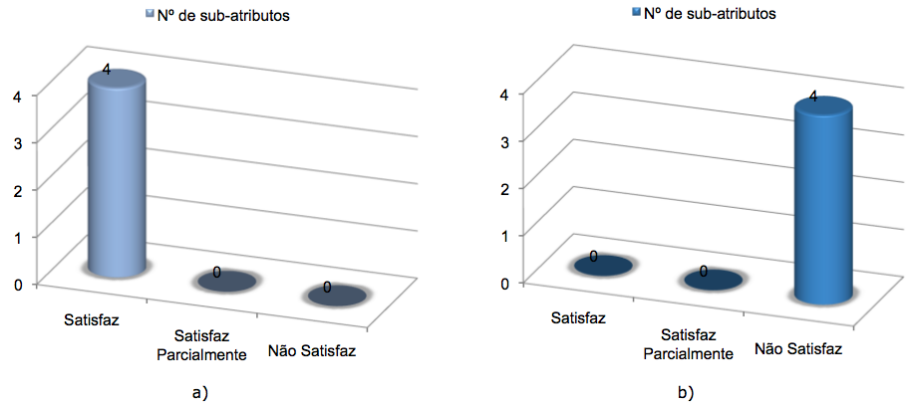

Figura 5.3: a) Resultados da análise ao XP sob o atributo Teste do Software; b) Resultados da análise ao Scrum sob o atributo Teste do Software

Na comparação, sob o atributo *Gestão da Engenharia de Software*, é possível verificar pelo gráfico a) da figura 5.4 que o XP **satisfaz** 80% dos sub-atributos analisados e **não satisfaz** 20% desses mesmos sub-atributos. Assim o XP revela-se consideravelmente completo no que refere ao suporte para a gestão de um projecto de software, não suportando apenas o sub-atributo que se refere à revisão e avaliação de um projecto. No caso do Scrum, como se pode verificar pelo gráfico b) da figura 5.4, **satisfaz** 80% dos sub-atributos analisados e **satisfaz parcialmente** 20% desses mesmos sub-atributos, apresentando-se no geral mais completo que o XP no que se refere à gestão de projectos de software.

Sob o atributo *Relação Princípios Ágeis - Práticas Advogadas*, os resultados mostram que o XP satisfaz 75% dos princípios apresentados no manifesto ágil (gráfico a) da figura 5.5), enquanto que o Scrum apenas satisfaz 50% desses mesmos princípios (gráfico b) da figura 5.5). Sendo a origem do XP anterior à criação do manifesto ágil, é claramente visível que o XP influenciou grande parte dos princípios que constituem o manifesto e os conceitos que caracterizam os MDAs. Assim, não é estranho o facto do XP satisfazer um maior número de princípios do manifesto quando comparado com o Scrum.

A tabela 5.17 resume os resultados obtidos.

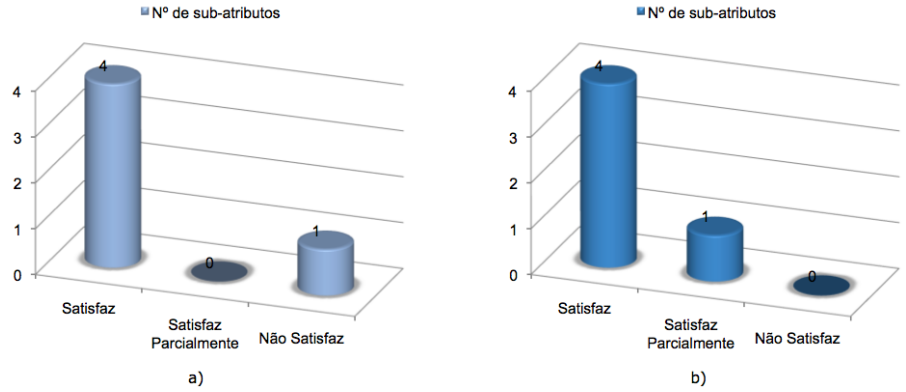

Figura 5.4: a) Resultados da análise ao XP sob o atributo Gestão da Engenharia de Software; b) Resultados da análise ao Scrum sob o atributo Gestão da Engenharia de Software

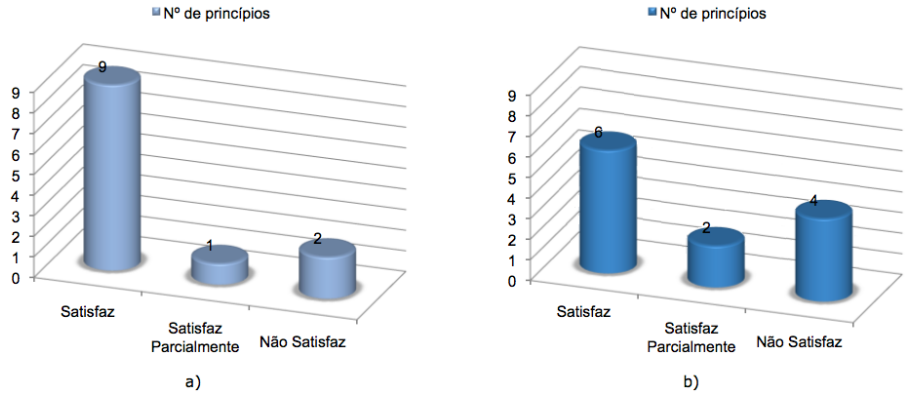

Figura 5.5: a) Resultados da análise ao XP sob o atributo Relação Princípios Ágeis - Práticas Advogadas; b) Resultados da análise ao Scrum sob o atributo Relação Princípios Ágeis - Práticas Advogadas

Atributo	**eXtreme Programming**			**Scrum**		
	S	***SP***	***NS***	***S***	***SP***	***NS***
Requisitos de Software	60%	40%	0%	60%	20%	20%
Construção de Software	75%	25%	0%	0%	0%	100%
Teste do Software	100%	0%	0%	0%	0%	100%
Gestão da Engenharia de Software	80%	0%	20%	80%	20%	0%
Relação Princípios Ágeis - Práticas Advogadas	75%	8%	17%	50%	17%	33%

Tabela 5.17: Síntese dos resultados obtidos

5.4 Ferramenta de apoio na escolha de MDAs

Com o objectivo de dar um carácter mais matemático ao estudo apresentado e auxiliar o leitor, desta dissertação, na avaliação do MDA que mais se adequa a uma determinada situação, foi criada uma grelha que permite a geração de um relatório com base neste mesmo estudo e num conjunto de dados introduzido pelo utilizador. De seguida é explicada essa mesma grelha e são apresentados alguns exemplos da sua utilização. A grelha criada encontra-se disponível para utilização em `http://rapidshare.com/files/166974426/msc_calc.xls`.

Esta grelha é constituída por quatro secções: 1) critérios de classificação, 2) sub-atributos, 3) atributos e 4) resultados da classificação. Os pesos associados aos critérios de classificação, sub-atributos e atributos são configuráveis pelo utilizador, influenciando a relevância de cada um desses elementos para o cálculo e consequentemente o resultado final do relatório. Estas secções são apresentadas na figura 5.6.

1.

Critério de Classificação	Peso
Satisfaz (S)	5
Satisfaz Parcialmente (SP)	3
Não Satisfaz (NS)	1

3.

Atributo	Peso	XP	Scrum
Requisitos de Software	16%	443	395
Construção de Software	20%	469	100
Teste do Software	18%	500	100
Gestão da Eng. De Soft.	34%	338	497
Rel. Princípios/Prácticas	12%	415	369
TOTAL:	100%	419	315

4.

2.

Atributo	Sub-Atributo	Peso	Total
Requisitos de Software	Fundamentos de Requisitos de Software	10	42
	Processo de Requisitos	10	
	Levantamento de Requisitos	10	
	Análise de Requisitos	2	
	Validação de Requisitos	10	
Construção de Software	Minimização da Complexidade	21	77
	Capacidade de Resposta a Possíveis Alterações	32	
	Implementação Orientada à Verificação	12	
	Utilização de Standards	12	
Teste do Software	Fundamentos de Testes do Software	10	40
	Níveis de Teste	10	
	Medição de Métricas Associadas aos Testes	10	
	Processo de Testes	10	
Gestão da Engenharia de Software	Iniciação e Definição do Âmbito	1	79
	Planificação do Projecto de Software	2	
	Aprovação e Execução do Plano do Projecto	12	
	Finalização do Projecto	32	
	Revisão e Avaliação	32	

Figura 5.6: Grelha de apoio na escolha de MDAs

Utilizando esta grelha é possível identificar, por exemplo, qual o MDA que mais se adequa a um ambiente de desenvolvimento onde os requisitos são incertos e os testes são

fundamentais e igualmente importantes. Assim, atribuindo um peso de, por exemplo, S=5, SP=3 e NS=1 aos critérios de classificação e considerando apenas os atributos *Requisitos de Software* e *Teste do Software* (assumindo um peso idêntico para cada sub-atributo), é possível verificar que o XP se destaca, de acordo com o estudo realizado, como sendo o MDA que mais se adequa a esta situação (figura 5.7).

Atributo	Peso	XP	Scrum
Requisitos de Software	50%	420	380
Construção de Software	0%	469	100
Teste do Software	50%	500	100
Gestão da Eng. De Soft.	0%	338	497
Rel. Princípios/Prácticas	0%	415	369
TOTAL:	***100%***	460	240

Figura 5.7: Grelha de apoio na escolha de MDAs - Exemplo de utilização 1

Se por outro lado se pretende aplicar um MDA num ambiente onde a existência de técnicas de gestão é prioritária, atribuindo os mesmos pesos aos critérios de classificação (S=5, SP=3 e NS=1) e analisando os MDAs apenas sob a perspectiva do atributo *Gestão da Engenharia de Software*, é possível identificar que o Scrum se destaca como sendo o MDA mais adequado (figura 5.8).

Atributo	Peso	XP	Scrum
Requisitos de Software	0%	420	380
Construção de Software	0%	469	100
Teste do Software	0%	500	100
Gestão da Eng. De Soft.	100%	338	497
Rel. Princípios/Prácticas	0%	415	369
TOTAL:	***100%***	338	497

Figura 5.8: Grelha de apoio na escolha de MDAs - Exemplo de utilização 2

Capítulo 6

Conclusões

A presente dissertação serve dois propósitos. O primeiro propósito é a sintetização da informação encontrada na literatura, referente aos MDAs, durante a realização deste trabalho. Desta forma foi reunido um conjunto de dados que descreve a importância da existência de um método de desenvolvimento em projectos de software, efectuada uma introdução aos MDAs e conceitos relacionados e apresentados em detalhe os dois MDAs mais referidos na literatura actual, o XP e o Scrum [Gre01, BT03, MR05, ASRW02].

O segundo propósito desta dissertação relaciona-se com a necessidade da realização de um estudo analítico que compare os diferentes MDAs existentes, salientando as semelhanças e diferenças existentes entre eles. Por forma a satisfazer este objectivo, foi seleccionado um conjunto de atributos que serviram como base de comparação entre os métodos analisados. Para este estudo foram utilizados o XP e o Scrum por serem os dois MDAs mais utilizados actualmente em empresas de software.

6.1 Reflexões

> *"It is predicted that a decade would not see any programming technique that would by itself bring an order-of-magnitude improvement in software productivity."* [FPB95]

Com o intuito de responder positivamente às actuais exigências de mercado das TICs, as empresas não podem ficar indiferentes às novas tecnologias, ferramentas e métodos de desenvolvimento.

A presente dissertação representa um contributo na selecção do MDA apropriado,

apresentando alguns dos aspectos e características dos MDAs analisados. No entanto, é possível concluir estes não são de longe a tão procurada "silver bullet". Apesar de inovadores e de abordarem a gestão e o desenvolvimento dos projectos de software de uma forma "radicalmente" diferente, os MDAs não apresentam ainda respostas concretas para muitos dos problemas que há muito consomem os profissionais da área de desenvolvimento de software.

Na opinião do autor desta dissertação, as dificuldades crescentes que se têm vindo a verificar nos projectos de software apenas poderão ser ultrapassadas através da experiência e excelentes competências técnicas dos profissionais que neles participam. Não existem projectos iguais e por isso as técnicas e práticas que se revelaram eficazes anteriormente, poderão não o ser no futuro. Assim, torna-se essencial um vasto conhecimento dos métodos de desenvolvimento existentes na altura de decidir qual o método de desenvolvimento a utilizar ou as práticas a aplicar, de acordo com as características de um projecto ou dos elementos que constituem a equipa de desenvolvimento. Desta forma uma solução possível seria, na opinião do autor desta dissertação, a utilização do método de desenvolvimento que mais se adequa ao tipo e características do projecto de software em questão, ou a aplicação conjunta de práticas advogadas pelos vários métodos de desenvolvimento.

6.2 Limitações deste trabalho e trabalho futuro

A principal limitação deste trabalho prende-se com a dificuldade em realizar um estudo comparativo entre métodos de desenvolvimento, pelos motivos apresentados na secção 5.1, e pela dificuldade que existe em seleccionar um conjunto de características ou atributos que sejam realmente decisivos para o sucesso ou insucesso de um projecto de software.

No estudo apresentado foi utilizado um conjunto de atributos na análise comparativa que se consideram relevantes no que se refere ao sucesso de um projecto de software. No entanto muito ainda pode ser realizado nesta área. De seguida ficam alguns pontos em aberto, cuja investigação se pode revelar útil e interessante:

- Alargar o conjunto de MDAs utilizados no estudo, de forma a incluir MDAs como o *Adaptive software development* (ASD), *Agile modeling* (AM), *Dynamic systems development method* (DSDM) e *Feature-driven development* (FDD);
- Analisar em detalhe a relação entre o cliente e a equipa do projecto;
- Analisar as diferenças existentes entre os MDAs no que se refere à organização,

comunicação e colaboração dentro da equipa de um projecto de software;

- Analisar a existência de ferramentas de suporte para a aplicação dos diferentes MDAs num projecto de software;
- Alargar o conjunto de ACs analisadas.

Anexo A: Manifesto Ágil

Valores:

Indivíduos e interacções são mais importantes que processos e ferramentas;

Software executável é mais importante que documentação completa e detalhada;

Colaboração do cliente é mais importante do que negociação de contratos;

Respostas rápidas a alterações é mais importante do que seguir o plano inicial.

Princípios:

1. A maior prioridade é a satisfação do cliente através da disponibilização atempada e contínua de software com valor;

2. Devem-se aceitar as alterações de requisitos, mesmo quando isso acontece em fases tardias do desenvolvimento. Os MDAs utilizam as alterações para fornecer vantagem competitiva ao cliente;

3. Deve-se disponibilizar software operacional com frequência, em intervalos que podem variar entre as duas semanas e os dois meses, embora com preferência pelos prazos mais curtos;

4. Os *stakeholders* e os programadores têm que trabalhar diariamente em conjunto ao longo do projecto;

5. Os projectos devem ser executados por indivíduos motivados. A partir daqui, há que fornecer-lhes o ambiente e o suporte que precisam, além de confiar neles para a realização do trabalho;

6. O método mais eficiente e eficaz de transmitir informação a uma equipa de desenvolvimento e dentro da mesma é a conversação frente-a-frente;

7. O software operacional é a principal métrica de progresso;

8. Os MDAs promovem o desenvolvimento sustentado. Os patrocinadores, os especialistas em desenvolvimento e os utilizadores devem ser capazes de manter indefinidamente um ritmo constante;

9. A atenção contínua à excelência técnica permite aumentar a agilidade;

10. A simplicidade - enquanto arte de minimização da quantidade de trabalho por fazer - é essencial;

11. As melhores arquitecturas, os melhores requisitos e os melhores desenhos de arquitecturas emergem de equipas que se auto-organizam;

12. A intervalos de tempo regulares, a equipa deve reflectir sobre formas de se tornar mais eficaz, procedendo seguidamente a ajustes no seu comportamento com esse fim em mente.

Bibliografia

[ABDM04] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2004.

[ACT01] Dennis M. Ahern, Aaron Clouse, and Richard Turner. *CMMI distilled: a practical introduction to integrated process improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Amb05] S. Ambler. Remaining Agile. 2005. `http://www.agilemodeling.com/essays/remainingAgile.htm`.

[ASRW02] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods - Review and analysis. Technical Report 478, VTT PUBLICATIONS, 2002.

[AWSR03] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New directions on agile methods: a comparative analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, Washington, DC, USA, 2003. IEEE Computer Society.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[Bac94] James Bach. The immaturity of the CMM. *AP*, 7(9):13–18, September 1994.

[Bar06] Liz Barnett. Agile Survey Results: Solid Experience And Real Results. Agile Journal, 2006.

[Bec99] Kent Beck. Embracing Change with Extreme Programming. *Computer*, 32(10):70–77, 1999.

[Bei03] Eduardo J. C. Beira. *Mercados de tecnologias da informação: do Minho à aldeia global*. AIMinho - Associação Industrial do Minho, Braga, Portugal, Novembro 2003.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988.

[Boe02] Barry Boehm. Get Ready for Agile Methods, with Care. *Computer*, 35(1):64–69, 2002.

[BT75] Victor R. Basili and Albert J. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Trans. Software Eng.*, 1(4):390–396, 1975.

[BT03] Barry Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[CH01] Alistair Cockburn and Jim Highsmith. Agile Software Development: The People Factor. *Computer*, 34(11):131–133, 2001.

[Coc02] Alistair Cockburn. *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Con06] Trail Ridge Consulting. Agile Project Management (APM) - Tooling Survey Results. Technical report, Dezembro 2006.

[DB07] Pete Deemer and Gabrielle Benefield. The scrum primer. an introduction to agile project management with scrum. 2007.

[DEM05] Fadi P. Deek, Osama M. Elhabiri, and James A. M. McHugh. *Strategic Software Engineering: An Interdisciplinary Approach*. Auerbach Publications, Boston, MA, USA, 2005.

[Dua02] Francisco José Monteiro Duarte. Engenharia de Software Orientada aos Processos. Master's thesis, Universidade do Minho, Braga, Portugal, Julho 2002.

[Dun01] Richard Duncan. The Quality of Requirements in Extreme Programming. *The Journal of Defense Software Engineering*, Junho 2001.

[EFF+99] Wolfgang Emmerich, Anthony Finkelstein, Alfonso Fuggetta, Carlo Montangero, and Jean-Claude Derniame. Software Process - Standards, Assessments and Improvement. In *Software Process: Principles, Methodology, Technology*, pages 15–26, London, UK, 1999. Springer-Verlag.

[Ema97] Khaled El Emam. *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.

[FD96] A. Finkelstein and J. Dowell. A comedy of errors: the London Ambulance Service case study. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 2, Washington, DC, USA, 1996. IEEE Computer Society.

[FH01] Martin Fowler and Jim Highsmith. The Agile Manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[FPB95] Jr. Frederick P. Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Goe03] Clement James Goebel. Extreme Programming Practices Used to Facilitate Effective Project Management. 2003.

[Gre01] James Grenning. Launching Extreme Programming at a Process-Intensive Company. *IEEE Softw.*, 18(6):27–33, 2001.

[HCR+94] J. Herbsleb, A. Carleton, J. Rozum, J. Siegel, and D. Zubrow. "Benefits of CMM-Based Software Process Improvement: Initial Results, 1994.

[HF06] Geir Kjetil Hanssen and Tor Erlend Faegri. Agile customer engagement: a longitudinal qualitative case study. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 164–173, New York, NY, USA, 2006. ACM.

[HG96] James D. Herbsleb and Dennis R. Goldenson. A systematic survey of CMM experience and results. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 323–330, Washington, DC, USA, 1996. IEEE Computer Society.

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990.

[ISO02] ISO/IEC 12207:1995/Amd 2:2002 Information technology - Software life cycle processes. Technical report, International Organization for Standardization, 2002.

[Jal02] Pankaj Jalote. *Software project management in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Jur99] Jaak Jurison. Software project management: the manager's view. *Commun. AIS*, page 2, 1999.

[Koc05] Alan S. Koch. *Agile Software Development: Evaluating the Methods for Your Organization*. Artech House, Inc, London, 2005.

[LBB+02] Mikael Lindvall, Victor R. Basili, Barry W. Boehm, Patricia Costa, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero, Laurie A. Williams, and Marvin V. Zelkowitz. Empirical Findings in Agile Methods. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, pages 197–207, London, UK, 2002. Springer-Verlag.

[LT93] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[McC96] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Redmond, WA, USA, 1996.

[Mil01] Granville G. Miller. The Characteristics of Agile Software Processes. In *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, page 385, Washington, DC, USA, 2001. IEEE Computer Society.

[Mil03] Randy Miller. The Dynamics of Agile Software Processes, Part I: Characteristics, 2003. Disponível em `http://dn.codegear.com/article/29726` e consultado em 28/05/2008.

[MR05] Andrea Massaro and Joakim Rosendahl. Agile Programming - Agile Software Engineering. Maio 2005.

[NR69] Peter Naur and Brian Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.* 1969.

[Pau01] Mark C. Paulk. Extreme Programming from a CMM Perspective. *IEEE Softw.*, 18(6):19–26, 2001.

[PHS+08] M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still. The impact of agile practices on communication in software development. *Empirical Softw. Engg.*, 13(3):303–337, 2008.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Higher Education, 2001.

[Rei02] Donald J. Reifer. How Good are Agile Methods? *IEEE Softw.*, 19(4):16–18, 2002.

[Roy70] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON.* IEEE Press, August 1970. Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338.

[SA05] Bob Schatz and Ibrahim Abdelshafi. Primavera Gets Agile: A Successful Transition to Agile Development. *IEEE Softw.*, 22(3):36–42, 2005.

[Sch95] Ken Schwaber. Scrum Development Process. In *OOPSLA'95 Business Object Design and Implementation Workshop*, London, 1995. Springer.

[Sch04] Ken Schwaber. *Agile Project Management With Scrum.* Microsoft Press, Redmond, WA, USA, 2004.

[Ser07] Serena. An Introduction to Agile Software Development. Technical report, Junho 2007.

[SO92] Xiping Song and Leon J. Osterweil. Toward objective, systematic design-method comparisons. *IEEE Softw.*, 9(3):43–53, 1992.

[Sol83] Henk G. Sol. A feature analysis of information systems design methodologies: Methodological considerations. *Information Systems Design Methodologies: A Feature Analysis*, pages 1–8, 1983.

[SS97] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[sta02] The Chaos Report. Technical report, The Standish Group, 2002.

[sta03] The Chaos Report. Technical report, The Standish Group, 2003.

[Sut06] Jeff Sutherland. A Brief Introduction to Scrum. 2006.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 1986.

[VN07] Inc. VersionOne and Agile Project Leadership Network. 2nd Annual Survey - The State of Agile Development. Technical report, Junho 2007.

[Wes06] Jason Westland. *The Project Management Life Cycle: A Complete Step-By-Step Methodology for Initiating, Planning, Executing & Closing a Project Successfully.* Kogan Page, Limited, 2006.

[Wie05] Karl E. Wiegers. *More About Software Requirements: Thorny Issues and Practical Advice.* Microsoft Press, Redmond, WA, USA, 2005.