Instituto Superior de Engenharia do Porto

Desenvolvimento de *software* com integração contínua

António Sérgio Matos Coelho

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquiteturas, Sistemas e Redes

Orientador: Doutor Nuno Alexandre Magalhães Pereira

Júri:

Presidente:

Professor Doutor Paulo Alexandre Fangueiro Oliveira Maio

Vogais:

Professor Doutor Nuno Alexandre Magalhães Pereira

Professor Doutor Nuno Miguel Gomes Bettencourt

Porto, Outubro 2015

Resumo

A integração contínua é uma prática no desenvolvimento de *software* que já existe há algum tempo mas ainda não é muito conhecida nem usada. Esta prática no desenvolvimento de *software* surgiu com a programação extrema e tem evoluído ao longo dos últimos anos, adaptando-se às novas tecnologias.

O estudo aqui apresentado pretende essencialmente mostrar a real importância e o valor acrescido que a integração contínua pode trazer a um projeto de desenvolvimento de *software*.

O trabalho aqui exposto surge no âmbito de um projeto interno, realizado pelo autor na empresa Konkconsulting, cuja finalidade se prende com o conhecimento mais aprofundado da integração contínua e com o levantamento das necessidades e criação de soluções, de modo a conseguir a sua utilização nos produtos que irão ser desenvolvidos pela empresa.

Nesta dissertação, é proposto um conjunto de ferramentas para responder às necessidades imediatas da empresa na implementação de integração contínua num dos seus projetos. Estas ferramentas devem ser de simples e de fácil utilização, de modo a ajudar os programadores durante os desenvolvimentos e responder às necessidades da utilização da integração contínua em um projeto a ser desenvolvido pela empresa, mas ao mesmo tempo, podendo ser facilmente incorporadas em futuros projetos.

Palavras-chave: Integração contínua, Desenvolvimento de *software*.

Abstract

Continuous integration is a software development practice that has existed for some time, but it is not widely known or used. This practice in the software development came with the extreme programming and has evolved over the years, adapting to new technologies.

The study presented here essentially aims to show the real importance and the value that continuous integration can bring to a software development project.

The work exposed here comes as part of an internal project, conducted by the author in Konkconsulting, whose purpose relates to the deeper knowledge of continuous integration and the needs assessment and creating solutions in order to achieve their use in products that will be developed by the company.

In this thesis, is proposed a set of tools to meet the company's close needs in implementing a continuous integration solution in their projects. These tools must be simple and easy to use in order to help developers during the developments and the needs of the use of continuous integration on a project being developed by the company, but at the same time can be easily incorporated into future projects.

Keywords: Continuous integration, Software Development.

Agradecimentos

Em primeiro lugar, quero agradecer aos meus pais e irmã que sempre me apoiaram em tudo o que foi necessário.

Ao Instituto Superior de Engenharia do Porto e mais concretamente ao Departamento de Engenharia Informática que contribuiu para o meu crescimento pessoal e profissional.

Ao meu orientador Nuno Pereira, pela disponibilidade demonstrada nos últimos anos e pela orientação que me prestou ao longo da elaboração do presente documento.

Por último, quero agradecer à Konkconsulting, aos meus colegas de trabalho e amigos, sem o apoio deles não seria possível a realização desta dissertação.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

CI	*Continuous integration*
CSV	*Comma-separated values*
DSDM	Desenvolvimento de sistemas dinâmicos
FDD	Desenvolvimento guiado por funcionalidades
GIT	*Global Information Tracker*
IDE	*Integrated Development Environment*
IIS	*Internet Information Services*
SVN	*Apache Subversion*
TFS	*Team Foundation Server*
XML	*eXtensible Markup Language*
XP	*Extreme programming*

1 Introdução

Neste primeiro capítulo, será feita uma breve apresentação do tema da dissertação, explicando como surgiu. Também será apresentada a motivação e os objetivos que levaram à realização deste trabalho. Por fim, será explicada a estrutura em que se encontra organizado este documento.

1.1 Enquadramento

Com a evolução da tecnologia, é necessário encontrar soluções para melhorar a qualidade e reduzir o número de erros no desenvolvimento de *software*. Os projetos tendem a ser maiores e mais complexos, o que aumenta o número de erros encontrados. Por exemplo, nos projetos que foram desenvolvidos ao longo dos últimos anos pela Konkconsulting[1], nota-se claramente um aumento da complexidade dos mesmos. Nesse sentido, a Konkconsulting decidiu apostar em novas tecnologias e práticas com o intuito de as usar nos desenvolvimentos dos seus novos produtos.

Como o desenvolvimento de um novo produto é demorado, de modo a reduzir riscos de atraso, o plano do projeto do novo produto foi orientado às funcionalidades. Foram definidas entregas, em média de duas em duas semanas. O objetivo destas entregas é ter um ambiente com novas funcionalidades e correções de erros introduzidas no último ciclo de desenvolvimentos, onde o gestor de projeto e o gestor de qualidade possam testar a aplicação.

De modo a obter um ambiente onde os gestores do projeto possam testar a aplicação, foi necessário alterar as metodologias de trabalho durante os desenvolvimentos. Foi definido pela empresa, em conjunto com os seus colaboradores, que iríamos usar a prática integração contínua na construção deste novo produto. Ao longo da tese, este novo produto

[1] http://www.konkconsulting.com/

desenvolvido pela empresa será usado como referência para provar ou exemplificar conceitos teóricos.

1.2 Motivação

No início da minha carreira profissional como programador informático, tive sempre a ambição e interesse de estar a par das novas tecnologias e metodologias que eram usadas no desenvolvimento de *software*. Nos primeiros projetos "a sério" da minha carreira como profissional, não só coloquei em prática conceitos que tinha aprendido durante a faculdade como também aprendi novos métodos para criar *software*. À medida que os anos foram passando, os projetos em que estive envolvido eram cada vez mais complexos, as minhas responsabilidades também aumentaram e a rápida resolução de problemas e criação de novas funcionalidades nas diferentes aplicações passou a ser bastante comum.

Para mim, é uma enorme satisfação criar um *software* novo. Ver algo crescer e que irá servir para ajudar outras pessoas é uma sensação muito boa, difícil de explicar. Com o lançamento de um novo produto pela empresa, surgiu a oportunidade de introduzir e apreender novas práticas nos desenvolvimentos de *software*. Uma dessas práticas (que foi introduzida) foi a integração contínua. No entanto, para introduzir a integração contínua nos desenvolvimentos, era necessário construir novas ferramentas com o objetivo de ajudar os programadores.

O trabalho aqui apresentado visa proporcionar as condições necessárias para uma equipa sem experiência em desenvolver *software* com integração contínua, consiga colocar em prática a integração contínua durante a criação de um novo produto. As ferramentas aqui criadas têm como objetivo responder a necessidades específicas que foram identificadas para introduzir esta nova prática.

Considero o tema desta tese bastante cativador e extremamente enriquecedor sobretudo para equipas que desenvolvam *software* e tenham a necessidade de estar constantemente a entregar versões novas da aplicação com novas funcionalidades. Com este projeto, pretendo essencialmente aplicar um conceito teórico em prática e demonstrar que a integração contínua pode ser uma mais-valia e trazer bastantes benefícios ao desenvolvimento de *software*.

1.3 Objetivos

Esta tese tem os seguintes objetivos:

- Apresentar a origem da integração contínua.
- Compreender a integração contínua.

    - Valor acrescentado pela integração contínua.
    - Benefícios e desvantagens.
    - Dificuldades em introduzir integração contínua nos projetos.
- Observar o ponto de visto do programador sobre integração contínua.
    - De que modo o pode influenciar.
    - Quais os benefícios.
- Mostrar práticas para conseguir obter a integração contínua eficazmente.
- Expor problemas e soluções encontradas durante a implementação da integração contínua nos desenvolvimentos de um produto real.

1.4 Organização do relatório

O presente documento encontra-se dividido em seis capítulos.

No primeiro capítulo, é efetuado um enquadramento do tema, enunciado o problema que levou à realização deste trabalho e são definidos os objetivos da tese.

No segundo capítulo, é apresentada a metodologia desenvolvimento ágil de *software*. É feita uma comparação com outras metodologias e são apresentados princípios e alguns métodos ágeis. Neste capítulo, também é feita a descrição da integração contínua, são apresentadas características e qual o valor acrescentado em usar integração contínua no desenvolvimento de *software*.

No terceiro capítulo, são apresentadas práticas para usar na integração contínua. Estas práticas têm como objetivo implementar eficazmente a integração contínua em diferentes projetos.

No quarto capítulo, é descrito o caso de estudo. É feita uma contextualização dos problemas, são expostas as exigências e são apresentadas práticas para efetuar o controlo de versões de bases de dados. Também é feito um enquadramento de todas as ferramentas desenvolvidas.

No quinto capítulo, é efetuada uma descrição detalhada de todas as ferramentas desenvolvidas.

No sexto e último capítulo, são apresentadas as conclusões onde é feita uma análise crítica ao trabalho desenvolvido e é referido o potencial trabalho futuro.

2 Estado da arte

Esta secção é destinada a uma análise do estado da arte do tema em estudo. Serão mostradas e explicadas diferentes práticas para desenvolver *software* sendo feita uma comparação entre si. Também será introduzido a integração contínua onde serão abordadas as suas características, qual o seu valor e os seus problemas. Por fim, serão demostradas ferramentas que ajudam à utilização da integração contínua em projetos de desenvolvimento de *software*.

2.1 Desenvolvimento de *software*

O desenvolvimento de *software* é um processo complicado que envolve muitas etapas e requer um planeamento cuidadoso de modo a cumprir os objetivos. Por vezes, um programador deve reagir rapidamente e de forma agressiva para responder às exigências do mercado em constante mudança. Manter a qualidade do *software* é um desafio e requer muitos períodos de testes para garantir produtos de qualidade.

Cada etapa no desenvolvimento de *software* requer muita papelada e documentação, além do processo de desenvolvimento e planeamento. Isto é o contraste com o que o comum das pessoas pensa do desenvolvimento de *software*, que acredita que desenvolver *software* é apenas escrever código. Em geral, o processo de desenvolvimento de *software* tem que passar pelas seguintes etapas:

- Levantamento de requisitos.
- Escrever as especificações funcionais.
- Arquitetura e desenho da aplicação.
- Implementação e codificação.

- Testes.
- Lançamento do produto.
- Documentação.
- Suporte e desenvolvimento de novas funcionalidades.

Estas etapas no desenvolvimento de *software* são ilustradas na Figura 1.

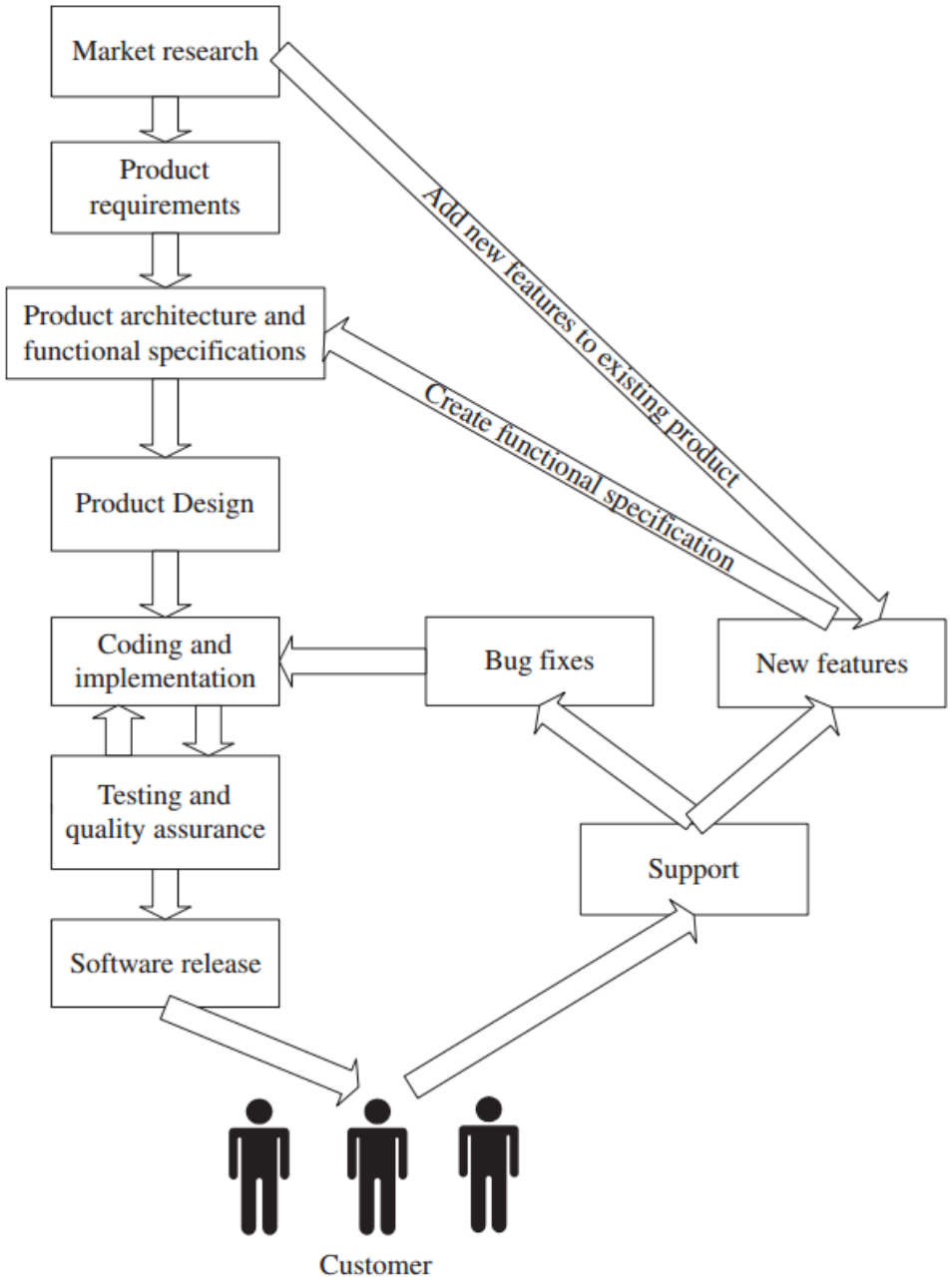

Figura 1 – Processo típico de desenvolvimento de *software* [Rafeeq Rehman e Christopher Paul, 2007]

2.2 Desenvolvimento ágil de *software*

2.2.1 Introdução

A tecnologia evolui muito rapidamente, os requisitos "mudam a um ritmo que esmagam os métodos tradicionais" [Jim Highsmith, 2000] e os clientes não estão disponíveis para dizer as suas necessidades enquanto ao mesmo tempo esperam mais do seu *software*. Como consequência, vários consultores desenvolveram de forma independente métodos e práticas para responder às mudanças que eles estavam a sentir. Cada consultor tinha a sua própria filosofia em como desenvolver o *software*. No entanto, todos eles defenderam uma colaboração conjunta entre a equipa de negócios e a equipa de *software*, ao contrário do desenvolvimento a solo pelas equipas de *software*.

Estes consultores preferiam comunicação cara-a-cara em vez de comunicação por documentos escritos, entregas frequentes de porções de *software* em vez de entregar apenas o produto no final e aceitar as mudanças nos requisitos em vez de ter requisitos fixos. Estes princípios [Martin Fowler, 2002] são a base da filosofia de desenvolvimento ágil de *software*.

O nome "ágil" surgiu em 2001, quando 17 programadores realizaram uma reunião para falar sobre as tendências futuras no desenvolvimento de *software*. O resultado dessa reunião foi a formação da "aliança ágil" e do manifesto para o desenvolvimento ágil de *software* [Kent Beck, 2001]. Este manifesto para o desenvolvimento ágil de *software* assenta em 4 valores:

1. Indivíduos e interações mais do que processos e ferramentas.
2. *Software* funcional mais do que documentação abrangente.
3. Colaboração com o cliente mais do que negociação contratual.
4. Responder à mudança mais do que seguir um plano.

O que significa ser ágil? Jim Highsmith enuncia que ser ágil significa ser capaz de entregar rapidamente, mudar rapidamente e mudar frequentemente [Jim Highsmith, 2000]. Nos métodos ágeis, as pessoas desempenham um papel importante no sucesso do projeto e muitas reuniões de curta duração são realizados para a partilha de conhecimento e para ajustar o projeto, se for necessário. Algumas pessoas argumentam que o *software* que funciona sem documentação é melhor do que o *software* que não funciona e tem uma enorme quantidade de documentação [Juha Koskela, 2003]. Não existe uma definição universal aceite de agilidade. Agilidade é dinâmica, especifica do contexto, disponível à mudança e é orientada para o crescimento [Steven Goldman, 1995]. O principal conceito da agilidade é a resposta rápida à mudança [Alistair Cockburn e Jim Highsmith, 2001].

2.2.2 Princípios

O manifesto para o desenvolvimento ágil de *software* é baseado em 12 princípios [Kent Beck entre outros, 2001]:

1. Desde as primeiras etapas do projeto, garantir a satisfação do cliente através da entrega rápida e contínua de *software* com valor.

2. Aceitar alterações de requisitos, mesmo numa fase tardia do ciclo de desenvolvimento. Os processos ágeis potenciam a mudança em benefício da vantagem competitiva do cliente.

3. Fornecer frequentemente *software* funcional. Os períodos de entrega devem ser de poucas semanas a poucos meses, dando preferência a períodos mais curtos.

4. O cliente e a equipa de desenvolvimento devem trabalhar juntos, diariamente, durante o decorrer do projeto.

5. Desenvolver projetos com base em pessoas motivadas, dando-lhes o ambiente e o apoio de que necessitam, confiando que irão cumprir os objetivos.

6. O método mais eficiente e eficaz de passar informação para dentro de uma equipa de desenvolvimento é através da conversa pessoal e direta.

7. A principal medida de progresso é a entrega de *software* funcional.

8. Os processos ágeis promovem o desenvolvimento sustentável. Os gestores, a equipa e os utilizadores deverão ser capazes de manter, indefinidamente, um ritmo constante.

9. A atenção permanente à excelência técnica e um bom desenho da solução aumentam a agilidade.

10. Simplicidade – a arte de maximizar a quantidade de trabalho que não é feito – é essencial.

11. As melhores arquiteturas, requisitos e desenhos surgem de equipas auto-organizadas.

12. A equipa reflete regularmente sobre o modo de se tornar mais eficaz, fazendo as adaptações e os ajustes necessários.

2.2.3 Comparação com outros métodos

2.2.3.1 Modelo em cascata

Os métodos ágeis são muitas vezes caracterizados pela falta de planeamento ou disciplina, por isso o desenvolvimento ágil tem pouco em comum com o modelo em cascata. O modelo em cascata é uma abordagem linear no desenvolvimento de *software*. É muito simples de perceber e usar. No modelo em cascata, cada fase deve ser completada antes que a próxima fase possa começar. Este tipo de modelo é geralmente usado em projetos pequenos onde os requisitos não vão mudar. No fim de cada fase, é feita uma revisão para determinar se o projeto está no caminho correto e se é para continuar ou cancelar o projeto. Neste modelo, os testes começam apenas depois dos desenvolvimentos estarem concluídos e as fases não se sobrepõem.

No modelo em cascata original de Winston Royce [Winston Royce 1970], ilustrado na Figura 2, as seguintes fases são seguidas em perfeita ordem:

1. Requisitos do sistema
2. Requisitos do *software*
3. Análise
4. Desenho do produto
5. Codificação
6. Testes
7. Manutenção

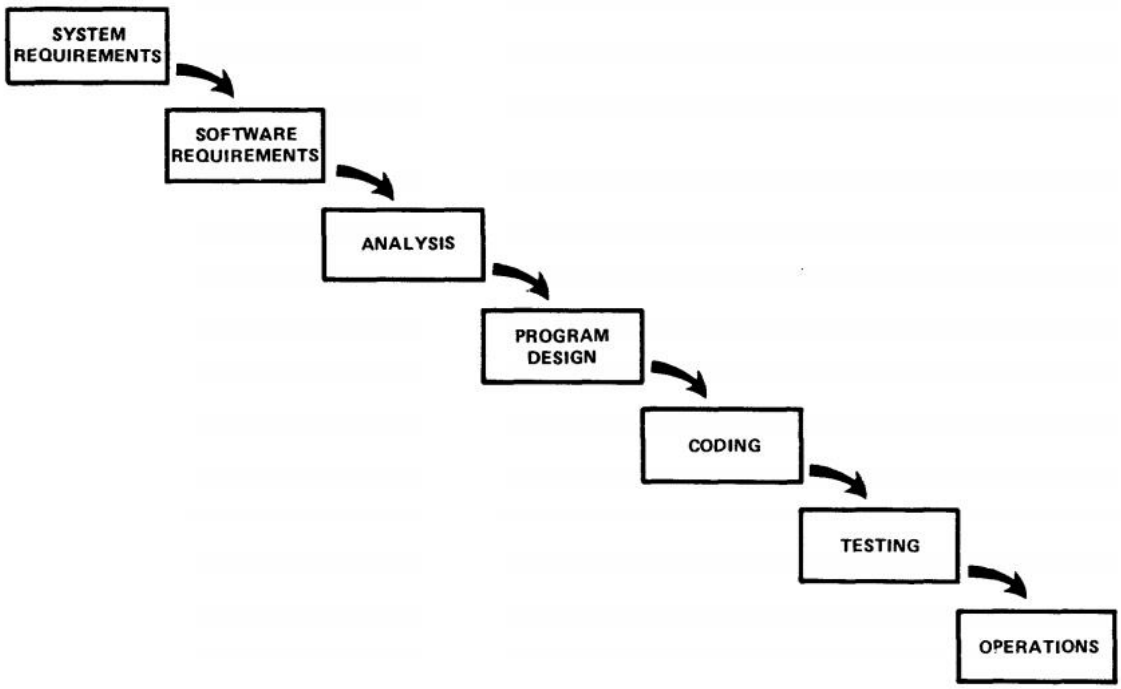

Figura 2 – Fases do modelo em cascata original [Winston Royce 1970]

O maior problema deste modelo é a sua falta de flexibilidade. Os clientes não sabem ao certo todos os seus requisitos antes de ver o *software* a funcionar, o que pode levar a mudar os requisitos. Esta alteração nos requisitos leva a redesenhar a aplicação, voltar a desenvolver e a testar a mesma, o que leva a um aumento no custo do *software*.

O modelo em cascata é usado por algumas equipas ágeis mas em pequena escala, repetindo o ciclo de cascata inteiro em cada iteração. Na programação extrema, as equipas trabalham com estas atividades em simultâneo.

2.2.3.2 Desenvolvimento iterativo e incremental

O desenvolvimento iterativo e incremental é um dos modelos clássicos no processo de desenvolvimento de *software* criado para responder às fraquezas do modelo em cascata.

Este modelo é uma estratégia de planeamento em que várias partes do sistema são desenvolvidas em paralelo e integradas quando completas. Uma alternativa ao desenvolvimento incremental é desenvolver todo o sistema com uma única integração.

A maioria dos métodos ágeis partilha o destaque no desenvolvimento iterativo e incremental para a construção de *software* em curtos períodos de tempo. Métodos ágeis diferem dos métodos iterativos porque seus períodos de tempo são mais curtos, medidos em semanas, ao contrário de meses e a sua realização é efetuada de uma maneira altamente colaborativa.

2.2.4 Métodos ágeis

Enquanto as práticas ágeis podem diferir um pouco de acordo com a metodologia ágil específica, existem práticas ágeis fundamentais que são baseadas nos quatro valores ágeis e nos doze princípios e são comuns a todas as metodologias ágeis. Neste capítulo, são abordadas algumas metodologias de desenvolvimento ágil.

2.2.4.1 Metodologia de desenvolvimento de sistemas dinâmicos

A metodologia de desenvolvimento de sistemas dinâmicos (DSDM) é provavelmente o método de desenvolvimento ágil original. O DSDM existia antes do termo "ágil" ser inventado, mas é baseado nos princípios que hoje conhecemos como ágeis. Esta metodologia é muito conhecida dentro do Reino Unido. Desde a sua origem em 1994, o DSDM tem gradualmente tornando-se a ferramenta número 1 para o desenvolvimento rápido de aplicações no Reino Unido [Jennifer Stapleton, 1997a].

A ideia fundamental por trás do DSDM é preferível, em vez de corrigir a quantidade de funcionalidades num produto e depois ajustar tempo e recursos para atingir as funcionalidades, corrigir tempo e recursos e depois ajustar a quantidade de funcionalidades. Esta ideia é ilustrada na Figura 3.

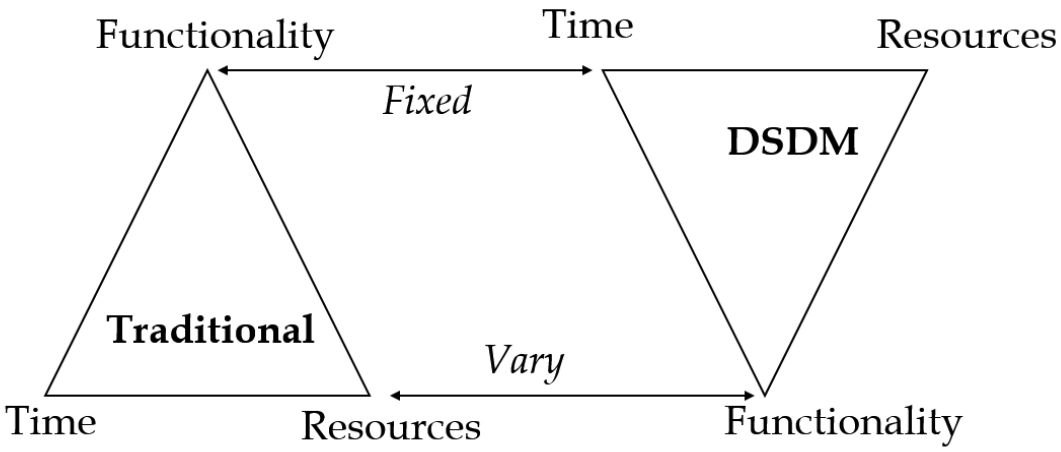

Figura 3 – Diferenças entre o desenvolvimento tradicional e o DSDM [Northumbria University, 2005]

Nove práticas definem esta metodologia e a base para todas as atividades no DSDM [Jennifer Stapleton, 1997a]. Estas práticas, chamadas de princípios no DSDM são:

1. Utilizador ativo – alguns utilizadores com conhecimento têm que estar presentes durante o desenvolvimento do sistema para garantir *feedback* atempado e correto.

2. A equipa deve poder tomar decisões – Um processo longo de tomadas de decisões não pode ser tolerado num rápido desenvolvimento de *software*. Os utilizadores envolvidos no desenvolvimento devem ter o conhecimento para dizer que rumo os desenvolvimentos devem tomar.

3. O foco deve ser a entrega frequente do produto – decisões erradas podem ser corrigidas se o ciclo de entrega for curto e os utilizadores fornecerem *feedback* correto.

4. Entrega de um sistema que se aproxime das necessidades atuais de negócio – Construir o produto certo antes de construí-lo corretamente. Antes das necessidades do negócio não estejam satisfeitas, a perfeição técnica é menos importante.

5. O desenvolvimento iterativo e incremental é necessário para convergir com precisão às soluções do negócio – os requisitos raramente não são alterados deste o início até ao fim do projeto. Ao construir um sistema iterativamente, os erros podem ser encontrados e corrigidos mais cedo.

6. Todas as mudanças durante os desenvolvimentos podem ser revertidas – durante os desenvolvimentos, um caminho errado pode ser facilmente usado. Ao usar iterações pequenas, é fácil retomar o caminho correto.

7. Manter os requisitos iniciais em alto nível – aprofundar os requisitos iniciais nas discussões iniciais não ajuda, visto que a solução evolve à medida que o produto ganha forma.

8. Os testes estão integrados durante o ciclo de vida – com as restrições de tempo, os testes são ignorados e deixados para o fim dos desenvolvimentos. Portanto, todos os componentes do sistema devem ser testados pelos programadores e utilizadores à medida que são desenvolvidos.

9. Uma abordagem colaborativa e cooperativa entre as partes envolvidas (*stakeholders*) é essencial – a escolha do que é entregue e o que é deixado de fora é sempre um compromisso e requer um acordo comum entre as partes envolvidas. Numa escala mais pequena, as responsabilidades do sistema são partilhadas, portanto a colaboração entre o utilizador e programador deve funcionar sem problemas.

No DSDM, existem 4 grandes fases durante o desenvolvimento. Dentro de cada fase, existem pequenas atividades. Estas fases, ilustradas na Figura 4 são:

1. Análise

a. Análise de viabilidade – nesta etapa, é analisado o tipo de projeto, os problemas organizacionais e das pessoas e é tomada a decisão de se utilizar ou não o DSDM.

b. Análise de negócio – nesta fase, são analisadas as características essenciais do negócio e as tecnologias a ser utilizadas.

2. Iterações do modelo funcional

    a. Identificar o protótipo funcional – nesta iteração, o objetivo é determinar as funcionalidades que serão implementadas.

    b. Agenda – nesta etapa, é definido quando e como as funcionalidades enumeradas no ponto anterior serão implementadas.

    c. Criação do protótipo funcional – neste passo, é criado um protótipo funcional da aplicação.

    d. Revisão do protótipo funcional – a finalidade desta fase é corrigir o protótipo criado no passo anterior.

3. Iterações do desenho e construção

    a. Identificar o modelo do desenho – efetuar o levantamento dos requisitos funcionais e não-funcionais que deverão estar no sistema.

    b. Agenda – nesta etapa, é definido quando e como os requisitos enumerados no ponto anterior serão realizados.

    c. Criação do protótipo do desenho – nesta fase, o objetivo é criar um sistema protótipo que será usado pelos utilizadores finais no dia-a-dia para testar o sistema.

    d. Revisão do protótipo – a finalidade desta etapa é fazer correções ao sistema desenhado no ponto anterior.

4. Implementação

    a. Orientações e aprovação do utilizador – nesta etapa, os utilizadores finais aprovam o sistema testado anteriormente.

    b. Treinar os utilizadores – preparar futuros utilizadores finais no uso do sistema.

    c. Implementação – implementar o sistema testado e entregar aos utilizadores finais.

d. Revisão de negócio – rever o impacto que o sistema implementado causa sobre o negócio. Pode-se utilizar os objetivos iniciais com a análise atual como comparação.

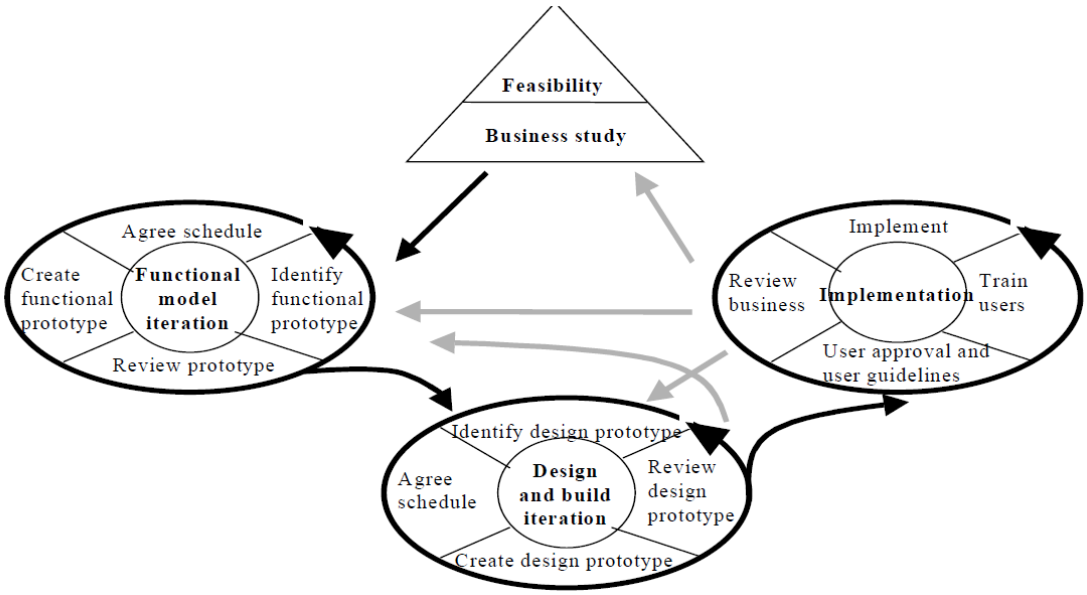

Figura 4 – Ciclo de vida de um projeto em DSDM [Jennifer Stapleton, 1997b]

2.2.4.2 Scrum

O *Scrum*, juntamente com a programação extrema, é um dos métodos ágeis mais utilizado. A abordagem *Scrum* foi desenvolvida para gerir o processo de desenvolvimento de sistemas. A primeira ocorrência do termo "*Scrum*" foi no artigo de Hirotaka Takeuchi e Ikujiro Nonaka em 1986 [Hirotaka Takeuchi e Ikujiro Nonaka, 1986], onde um processo de desenvolvimento de produto adaptável, rápido e que se organiza a si mesmo que é originário do Japão é apresentado [Ken Schwaber e Mike Beedle, 2002].

O *Scrum* não define quaisquer técnicas de desenvolvimento de *software* específicas para a fase de implementação. Esta metodologia concentra-se em como os membros da equipa devem funcionar de modo a produzir o sistema de forma flexível em um ambiente em constante mudança.

A principal ideia do *Scrum* é que o desenvolvimento de sistemas envolva várias variáveis ambientais e técnicas (por exemplo: requisitos, prazos, recursos e tecnologia) que são suscetíveis de alteração durante o processo. Isso faz com que o processo de desenvolvimento seja imprevisível e complexo, exigindo a flexibilidade do processo de desenvolvimento de sistemas para que seja capaz de responder às alterações. Como resultado do processo de desenvolvimento, é produzido um sistema que é útil quando é entregue [Ken Schwaber, 1995].

Os princípios chave do *Scrum* são [Ken Schwaber e Mike Beedle, 2002]:

- Pequenas equipas que maximizam a comunicação minimizam a sobrecarga e maximizam a partilha de conhecimento.
- Adaptabilidade às mudanças técnicas ou de mercado (utilizador/cliente) para garantir que o que é produzido é o melhor produto possível.
- Construções frequentes que podem ser inspecionadas, ajustadas, testadas e documentadas.
- Divisão de trabalho em tarefas simples e de baixo acoplamento.
- Constante documentação e testes do produto à medida que é construído.
- Possibilidade de declarar um produto como acabado sempre que necessário (porque a competição já acabou, a empresa precisa de dinheiro, o cliente precisa das funcionalidades ou porque foi quando foi prometido).

No *Scrum* são definidos 3 papéis principais num projeto:

1. *Scrum master* – garante que a equipa é funcional e produtiva. É o responsável pelo processo *Scrum*. Geralmente, é o gestor de projeto.
2. Dono do produto – é o responsável pelo projeto e é responsável por garantir que a equipa traga valor ao negócio. É quem define e prioriza os requisitos do sistema.
3. Equipa – a equipa de desenvolvimento é responsável pela entrega do produto. A equipa é geralmente composta de 5-9 pessoas com aptidões multifuncionais para realizar o trabalho.

As etapas de desenvolvimento no *Scrum* são apelidadas de *sprints*. Cada *sprint* dura tipicamente entre uma semana a um mês. No início de cada *sprint,* é realizada uma reunião de planeamento onde são definidas as tarefas para essa *sprint*. Nesta reunião, o dono do produto informa a equipa das tarefas que quer concluídas durante a *sprint*.

Durante o ciclo de vida do projeto no *Scrum*, existe uma reunião diária. Esta reunião deve ter a duração inferior a 15 minutos e tem como objetivo sincronizar os vários elementos da equipa. Nesta reunião, devem participar o *Scrum master* e os membros da equipa de desenvolvimento. Os membros da equipa devem responder a 3 perguntas nas reuniões diárias:

1. O que é que eu fiz ontem?
2. O que é que eu vou fazer hoje?

3. Quais os obstáculos que impedem de progredir nas minhas tarefas (se existirem)?

As iterações entre as diferentes partes envolvidas no ciclo de desenvolvimento no *Scrum* estão ilustradas na Figura 5.

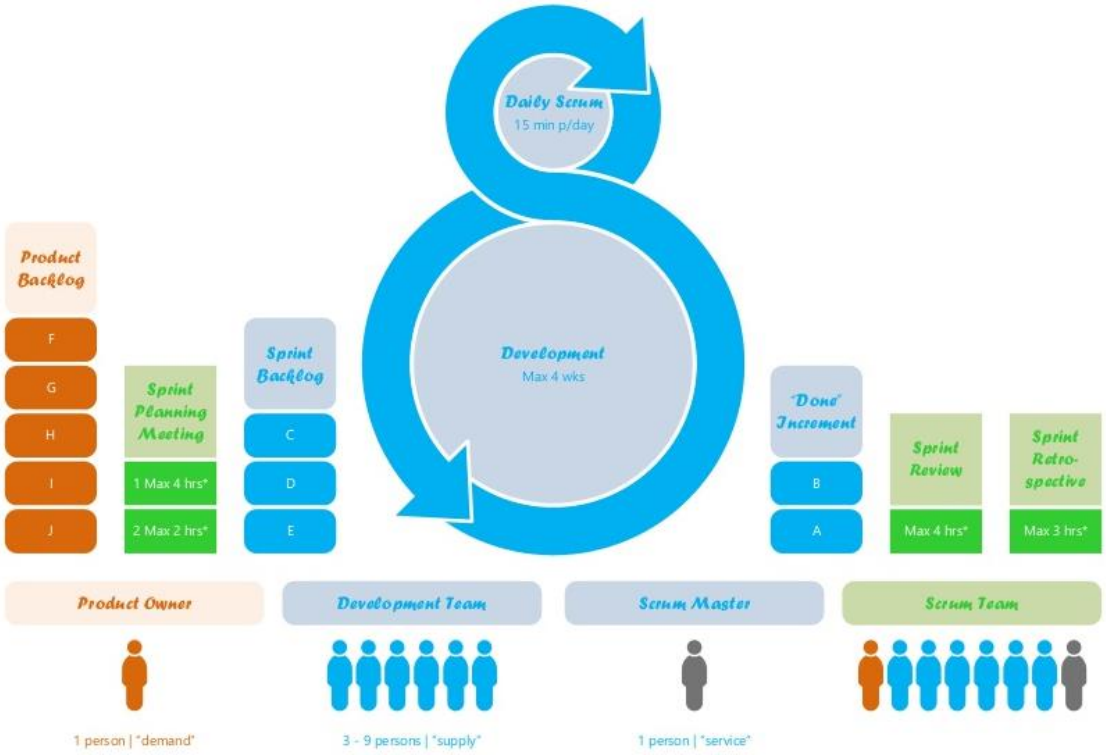

Figura 5 – Ciclo de vida de um projeto em *Scrum* [Mark Hoogveld, 2012]

2.2.4.3 Programação extrema

A programação extrema (XP) evoluiu a partir dos problemas causados pelos ciclos longos de desenvolvimento de modelos de desenvolvimento tradicionais [Kent Beck, 1999a]. Começou como uma simples oportunidade de realizar o trabalho [Jim Haungs, 2001] com práticas que tinham sido eficazes nos processos de desenvolvimento de *software* durante as décadas anteriores [Kent Beck, 1999b].

A programação extrema é uma disciplina de desenvolvimento de *software* baseada em 5 valores [Don Wells, 2009a]:

1. Simplicidade – fazer o que é preciso e pedido e nada mais. Isto irá maximizar o valor criado para o investimento feito. Tomar pequenos passos simples para o objetivo e atenuar as falhas à medida que elas acontecem. Criar algo que os programadores estejam orgulhosos de manter a longo prazo a custos razoáveis.

2. Comunicação – todos fazem parte da equipa e a comunicação é feita diariamente cara-a-cara. Trabalhar em conjunto em tudo, desde os requisitos até à codificação. Criar juntos a melhor solução para o problema.

3. *Feedback* – levar cada iteração a sério e entregar *software* a funcionar. Mostrar o *software* numa fase cedo dos desenvolvimentos e muitas vezes, ouvir atentamente e fazer as alterações necessárias. Falar do projeto e adaptar o processo a ele e não ao contrário.

4. Respeito – toda a gente dá e sente o respeito que eles merecem como um membro valioso da equipa. Toda a gente contribui com valor. Os programadores respeitam a perícia dos clientes e vice-versa.

5. Coragem – podemos dizer a verdade sobre o progresso e estimativas. Não são documentadas desculpas para o falhanço, porque é planeado para o sucesso. Não há medo de nada, porque ninguém trabalha sozinho. Os programadores adaptam-se às mudanças sempre que elas acontecem.

Esta disciplina funciona, juntando toda a equipa na presença de práticas simples, com *feedback* suficiente para que a equipa consiga ver em que ponto estão e ajustar as práticas à sua posição única. Como um tipo do desenvolvimento ágil de *software*, esta invoca lançamentos frequentes da aplicação com curtos períodos de desenvolvimento, que têm como objetivo melhorar a produtividade e permitir que as novas funcionalidades propostas pelo cliente possam ser adotadas. A Figura 6 ilustra as fases do ciclo de vida de um projeto com a programação extrema.

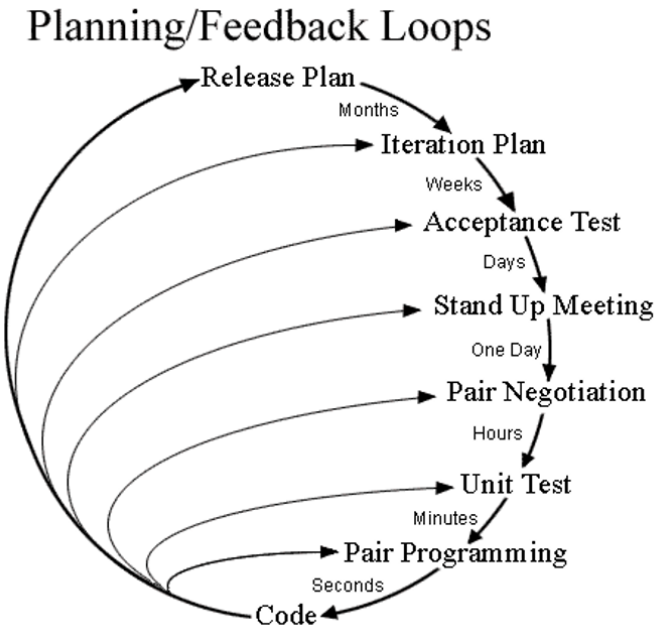

Figura 6 – Ciclo de vida de um projeto na programação extrema [Don Wells, 2009b]

A programação extrema é uma coleção de ideias e práticas criadas a partir de metodologias existentes [Kent Beck, 1999a]. As 12 boas práticas sugeridas por Kent Beck [Kent Beck, 1999b] são:

1. Planeamento – interação próxima entre o cliente e os programadores. Os programadores estimam o esforço necessário para implementar as funcionalidades e o cliente decide sobre o âmbito e quando é o lançamento das funcionalidades.

2. Entregas frequentes – um sistema simples é produzido rapidamente. Cada versão entregue deve ter o menor tamanho possível, contendo apenas os requisitos de maior valor para o negócio. Idealmente, devem ser entregues novas versões a cada mês, ou no máximo a cada dois meses, aumentando a possibilidade de *feedback* rápido do cliente. Isto evita surpresas, caso o *software* seja entregue após muito tempo e melhora as avaliações do cliente, aumentando a probabilidade do *software* final estar de acordo com os requisitos do cliente.

3. Metáfora – o sistema é definido sem a utilização de termos técnicos, pelo cliente e pelo programador.

4. Desenho simples – a solução implementada deve ser a solução mais simples que pode ser implementada no momento. Complexidade que não é necessária e código extra são removidos imediatamente.

5. Teste – Os testes unitários são implementados antes de o código e estão a correr continuamente.

6. Reformulação – a reformulação do sistema deve ser feita apenas quando é necessário, quando há código duplicado ou é possível simplificar e adicionar flexibilidade sem remover funcionalidades.

7. Programação aos pares – dois programadores sentados na mesma máquina a escrever código.

8. Propriedade coletiva – qualquer pessoa pode mudar qualquer parte do código a qualquer altura.

9. Integração contínua – um novo pedaço de código é introduzido no código base assim que possível. Todos os testes devem passar para que as alterações sejam aceites.

10. 40 Horas de trabalho semanal – não se deve fazer horas extra constantemente. Caso seja necessário fazer horas extras por mais que uma semana consecutiva, algo está mal no projeto. Tem que ser encarado como um problema e precisa de ser resolvido.

11. Cliente presente – o cliente tem que estar presente e disponível sempre que a equipa precisar dele para responder a questões e garantir que o desenvolvimento seja gerado como esperado.

12. Código padrão – regras de codificação existem e devem ser seguidas pelos programadores para que o código possa ser partilhado entre todos.

A Figura 7 ilustra as 12 práticas da programação extrema e as relações entre si.

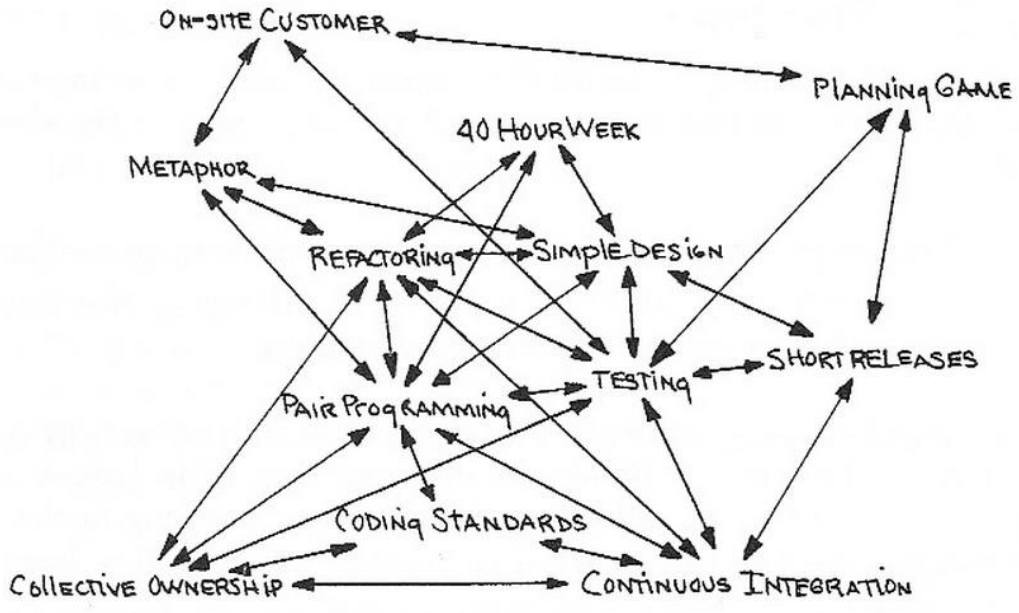

Figura 7 – As práticas suportam-se entre si [Kent Beck, 1999c]

"As práticas suportam-se entre si. As fraquezas de uma estão cobertas pelas forças de outras." [Kent Beck, 1999d].

2.2.4.4 Desenvolvimento guiado por funcionalidades

O desenvolvimento guiado por funcionalidades (FDD) foi inicialmente publicado em 1999, no capítulo 6 do livro "*Java Modeling in Color with UML*", de Peter Coad, Eric Lefebvre e Jeff De Luca. É uma abordagem ágil e adaptável para desenvolver sistemas. O FDD não cobre o processo todo de desenvolvimento de *software* mas centra-se nas fases de conceção e de construção [Steve Palmer e Mac Felsing, 2002]. No entanto, foi desenhado para trabalhar com outras atividades de um projeto de desenvolvimento de *software* [Steve Palmer e Mac Felsing, 2002] e não requer qualquer modelo específico para ser usado.

O FDD procura o desenvolvimento por funcionalidades, ou seja, por um requisito funcional do sistema. Apesar de haver algumas diferenças entre o FDD e a programação extrema, é possível utilizar as melhores práticas de cada metodologia. O FDD atua muito bem em conjunto com o *Scrum*, pois o *Scrum* atua no foco de gestão do projeto e o FDD atua no processo de desenvolvimento.

O FDD possui cinco processos básicos:

1. Desenvolvimento de um modelo geral – análise orientada por objetos.
2. Construir uma lista de funcionalidade – decomposição funcional.
3. Planear por funcionalidades – planeamento incremental.
4. Desenhar por funcionalidades – desenho orientado a objetos.
5. Construção por funcionalidades – programação e testes orientados a objetos.

A Figura 8 ilustra estes cinco processos básicos do FDD.

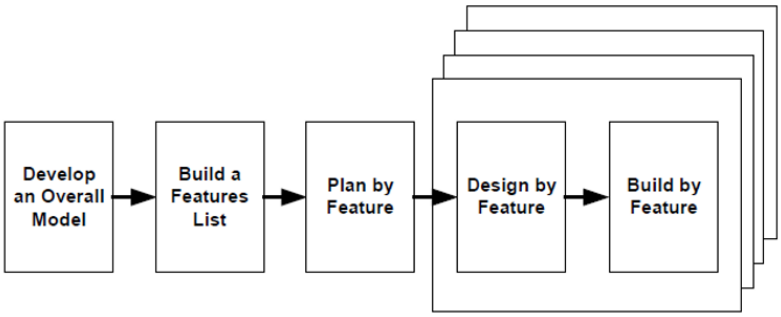

Figura 8 – Processos do FDD [Palmer e Felsing, 2002]

2.3 Introdução à integração contínua

A integração contínua é uma prática no desenvolvimento de *software*, onde os membros de uma equipa integram o seu trabalho frequentemente. Habitualmente, cada pessoa integra o seu trabalho uma vez por dia, o que leva a múltiplas integrações por dia. Cada integração é verificada por um sistema automático, que inclui testes de modo a detetar erros de integração o mais rapidamente possível. O objetivo destes testes é fornecer uma resposta rápida para quando um erro for introduzido no código, possa ser identificado e retificado de imediato.

O termo "integração contínua" teve origem com o processo de desenvolvimento de *software* programação extrema, mas os seus princípios podem ser aplicados a qualquer modelo de programação interativo.

A programação extrema é uma disciplina de desenvolvimento de *software* baseada nos valores de simplicidade, comunicação, *feedback*, coragem e respeito. A integração contínua assenta sobretudo nos princípios básicos do *feedback* rápido, mudanças incrementais e trabalho em equipa mas, tem evoluído desde a sua criação. Originalmente, a construção diária da aplicação era o *standard*. Hoje em dia, a regra é cada membro da equipa submeter o seu trabalho diariamente ou até mais frequentemente e a construção da aplicação ser efetuada a cada mudança significativa da aplicação.

Quando usada corretamente, a integração contínua fornece vários benefícios como o *feedback* constante do estado do desenvolvimento do *software* para todos os elementos da equipa (programadores e os vários gestores de projeto). Como a integração contínua deteta problemas numa fase inicial do desenvolvimento *software*, os problemas são geralmente pequenos, pouco complexos e fáceis de resolver.

Num projeto, pessoas com diferentes funções podem submeter código o que provoca um ciclo de integração contínua: programadores mudam o código fonte, administradores de base de dados alteram a definição de tabelas, gestores de projeto mudam configurações no projeto, entre outras. Os passos de um cenário de integração contínua, ilustrados na Figura 9, são habitualmente os seguintes:

1. Um cenário de integração contínua começa com o programador a submeter o código fonte no repositório. Entretanto, o servidor de integração contínua está constantemente a verificar o repositório por alterações (por exemplo a cada 2 minutos)

2. Depois de o programador submeter as alterações no código fonte, o servidor de integração contínua deteta que uma alteração ocorreu no código fonte. Assim que são detetadas as alterações, o servidor obtém uma copia do repositório, com as últimas alterações e executa o *script* que efetua a construção da aplicação.

3. O servidor de integração contínua fornece *feedback* sobre a construção da aplicação, enviando uma mensagem para um membro específico da equipa.

4. O servidor de integração contínua prossegue a verificar o repositório por alterações.

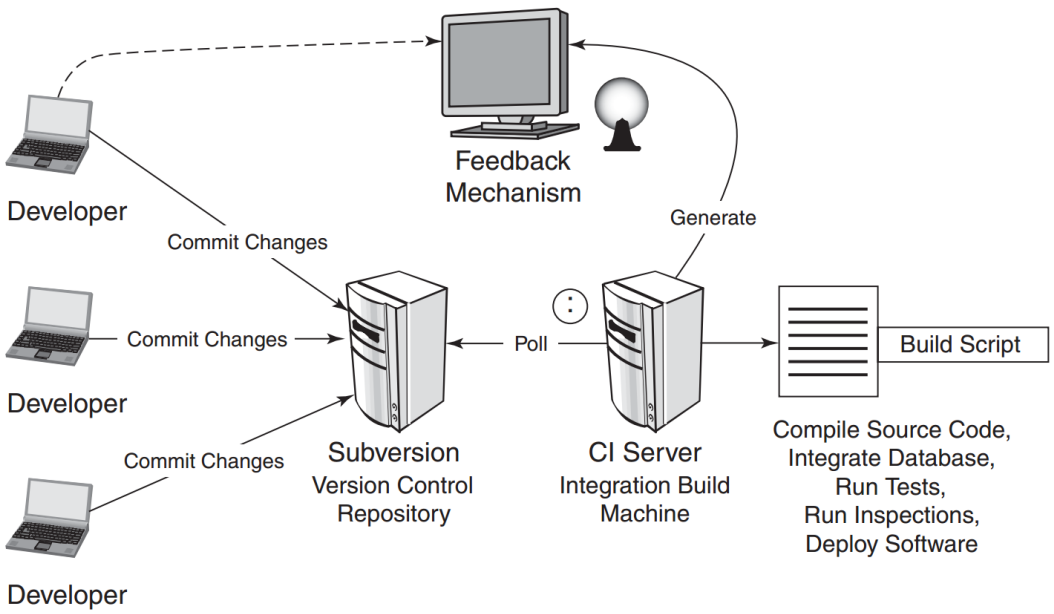

Figura 9 – Componentes de um sistema de integração contínua [Paul Duvall, 2007a]

Os componentes de um sistema de integração contínua que aparecem na Figura 9 são:

- Programador – a pessoa que provoca a integração contínua.
- Repositório de código – sistema que controla as versões do código fonte.
- Servidor de integração contínua – deteta alterações que ocorrem no código fonte e inicia a construção da aplicação.
- *Feedback* – sistema que é encarregado de informar o estado da construção da aplicação a um membro da equipa.

2.4 Características da integração contínua

Segundo Paul Duvall, existem apenas 4 características que são obrigatórias num sistema de integração contínua [Paul Duvall, 2007d]:

1. Ligação ao repositório de controlo de versões do código.
2. Um *script* para construir a aplicação.
3. Um sistema de *feedback* (por exemplo, através de correio eletrónico).
4. Um processo para integrar as alterações do código fonte (manualmente ou através de um servidor de integração contínua).

Esta base é a chave para um sistema de integração contínua. Assim que a construção da aplicação esteja a correr a cada alteração de código, é possível adicionar mais funcionalidades ao sistema de integração contínua.

Ao realizar automática e continuamente a integração da base de dados, realização de testes e construções da aplicação, é possível reduzir riscos comuns no projeto.

2.5 Qual é o valor da integração contínua?

Se começamos a usar alguma prática nova no desenvolvimento de *software*, é porque essa prática traz valor acrescido. O valor acrescido que a integração contínua traz, é a redução de riscos.

Ao integrar muitas vezes ao dia, é possível reduzir riscos no projeto. Como a integração contínua corre testes varias vezes ao dia, existe uma maior probabilidade de encontrar erros mais cedo. No limite, os erros podem ser encontrados logo após serem introduzidos, se os testes estiverem a correr após cada submissão de código.

Com a integração contínua, também são reduzidos os processos repetitivos que um programador iria executar. Ao reduzir os processos respetivos, é poupado tempo e dinheiro, visto que o programador pode estar a fazer outras tarefas em vez de construir a aplicação.

Como a integração contínua está constantemente a correr, isto torna possível lançar *software* a qualquer ponto no tempo. Do ponto de vista externo, isto é o benefício mais óbvio. Podemos falar em melhorar a qualidade de *software* e reduzir os riscos, mas para os cliente e utilizadores ter um *software* a funcionar é o principal objetivo. É possível fazer pequenas alterações e integra-las rapidamente. Se houver algum problema, os membros do projeto podem comunicar entre si e uma correção é feita imediatamente. Projetos que não abraçam esta filosofia podem esperar até à entrega para integrar e testar. Se houver algum problema, isto pode atrasar o lançamento da nova funcionalidade ou correção do problema.

No geral, a integração contínua fornece uma maior confiança na produção de *software*. Sem integrações frequentes, a equipa não sabe o impacto das suas alterações. Com cada construção da aplicação, a equipa sabe que há testes que verificam o estado dos desenvolvimentos e que as normas de desenvolvimento são seguidas e isto resulta num produto mais robusto e de melhor qualidade. No geral, a integração contínua pode reduzir o risco e melhorar a qualidade de *software* [Paul Duvall, 2007b].

2.6 Quem precisa de integração contínua?

Como pudemos ver na integração contínua, quando há uma construção da aplicação a falhar, a equipa deve parar, focar-se em resolver os problemas e submeter código que resolva o problema. Isso demora tempo. Portanto, quem precisa de integração contínua?

- O dono do produto que está interessado em lançar novas funcionalidades o mais rapidamente possível.
- O gestor de projeto que é responsável por respeitar as metas temporais do projeto.
- O programador que não gosta de corrigir erros dos outros e tem sempre a pressão para resolver algum erro importante da aplicação.

Quem gosta desta integração contínua? A resposta é simples: ninguém.

O cenário é sempre o mesmo. Ao fim de algum tempo, o resultado da construção da aplicação é ignorado. A construção da aplicação pode estar a falhar ou a funcionar, que é ignorada. E porque é ignorada? Porque não há tempo para corrigir erros na construção da aplicação.

Este foi o principal problema de a integração contínua não ter tido muito sucesso no desenvolvimento do produto pela empresa. Nos primeiros dois meses de desenvolvimentos do produto, foram desenvolvidos apenas serviços que viriam a ser usados na próxima fase de desenvolvimento, o *website*. Por isso, não havia nada visível onde os gestores do projeto

pudessem ver que a integração contínua estava a funcionar. Tínhamos testes unitários com o objetivo de testar os serviços mas não havia um *website* onde fosse possível uma pessoa testar a aplicação.

Quando se iniciou os desenvolvimentos no *website,* começaram os problemas. A construção da aplicação era demorada. O servidor de integração contínua estava fora da rede da empresa, por isso a transferência de ficheiros era lenta. O servidor onde era efetuada a construção da aplicação era o mesmo que era usado para os gestores de projeto testarem a aplicação, por isso não era possível fazer construções regulares para testar. Os primeiros testes unitários não estavam corretos ou já não eram necessários mas não havia tempo de corrigir ou apaga-los.

Por causa destas condições, apenas era testada a construção da aplicação quando era realmente necessário. O problema é que quase sempre que era feita uma construção da aplicação, havia um pequeno problema que era necessário corrigir e começar todo o processo do início, o que demorava tempo e os gestores de projeto não ficavam nada satisfeitos com os programadores.

2.7 O que impede as equipas de usar integração contínua?

Se a integração contínua tem tantos benefícios, então o que impede uma equipa de usar? Geralmente é um dos seguintes fatores:

- Aumento da sobrecarga de manutenção – isto é geralmente uma perceção errada, porque a necessidade de testar, inspecionar e construir a aplicação existe, quer seja usando integração contínua ou manualmente.

- Demasiadas mudanças – por vezes, as equipas pensam que mudar as aplicações antigas com a intenção de usar integração contínua é um processo que requer muitas mudanças. Neste caso, uma mudança incremental é o ideal. Primeiro, acrescentar testes, depois, a construção da aplicação com pouca frequência e assim que os programadores estejam mais à-vontade, aumentar a frequência.

- Demasiadas construções da aplicação falhadas – normalmente acontece quando não se testa regularmente a construção da aplicação. Os programadores devem reservar tempo nas suas tarefas com o objetivo de corrigir problemas na construção da aplicação.

- Necessidade de adquirir *software*/*hardware* – de modo a usar eficazmente a integração contínua, pode ser necessário adquirir um novo servidor. No entanto, o custo de um novo servidor, comparado ao custo de encontrar problemas mais tarde na fase de desenvolvimento do projeto e atrasar o projeto, pode não compensar.

- Programadores já deviam realizar estas tarefas de integração contínua – por vezes, os gestores de projeto pensam que a integração contínua é a duplicação de tarefas que os programadores já deviam realizar. Isto é verdade, mas um sistema que automaticamente realiza estas tarefas é mais eficaz. Ao realizar estas tarefas em um outro sistema, no ambiente de integração contínua, podem ser encontrados problemas que não são encontrados localmente nas máquinas onde os programadores desenvolvem a aplicação.

2.8 Tipos de projetos que podem beneficiar da integração contínua

Equipas com menos de 50 elementos a trabalhar em projetos pouco complexos são o grupo ideal para usar a integração contínua mas, os produtos estão cada vez mais a ficar mais inteligentes e tem havido um aumento na sua complexidade. [IBM, 2012]

A inclusão de *software* em produtos tradicionais é enorme. Hoje em dia, um frigorífico pode ter ligação à internet! Este aumento de complexidade nos produtos surge com a aceleração na comercialização de novos produtos. Todos os dias, há um novo produto no mercado. Este aumento de *software* embebido nos diversos produtos, combinado com prazos apartados, trouxe práticas ágeis e a integração contínua para os programadores.

É melhor implementar integração contínua na fase inicial de um projeto. Embora seja possível, é mais difícil implementar integração contínua num projeto que está numa fase mais adiantada dos desenvolvimentos, onde as pessoas estão com a pressão de terminar as suas tarefas e não querem alterar os seus hábitos de trabalho. Estes são menos suscetíveis de resistir à mudança.

Integração contínua não é apenas uma implementação técnica, também é uma implementação cultural e organizacional.

2.9 Ferramentas

Nesta secção, são apresentadas duas ferramentas que ajudam a automatizar o processo de integração contínua. Apesar de não ser necessário utilizar nenhuma ferramenta para alcançar integração contínua, uma ferramenta que automatize o processo é uma ajuda preciosa.

2.9.1 TFS/Release Management

2.9.1.1 História

O *Team Foundation Server* (TFS)[2] é um produto criado pela Microsoft que fornece gestão do código fonte através do *Team Foundation Version Control* ou *Git*, criação de relatórios, gestão de requisitos, gestão de projeto tanto para desenvolvimento ágil bem como em cascata, construções automáticas da aplicação, testes das aplicações e gestão de lançamento das aplicações. Cobre todo o ciclo de vida da aplicação [Microsoft, 2013].

O objetivo principal do TFS não é servir de servidor de integração contínua. Tem evoluído ao longo dos anos adicionando novas funcionalidades a cada versão. Umas dessas funcionalidades é a construção automatizada de aplicações. Em 2013, a Microsoft comprou um produto chamado "*InRelease*" da *InCycle Software* [*The Next Web*, 2013]. Este produto foi totalmente incorporado na versão 2013 do TFS. Esta ferramenta completou a automatização da construção da aplicação e o processamento de testes, permitindo uma solução de contínuo lançamento de versões de *software*. Esta nova ferramenta foi batizada de *Release Management*[3] na versão 2013 do TFS.

2.9.1.2 Características

O *Release Management* permite:

- Personalizar a construção da aplicação – configurar variáveis de sistema, copiar ficheiros, copiar certificados, criar *websites* e configurar múltiplos sistemas.
- Configurar construções automáticas da aplicação após cada submissão de código.
- Configurar notificações – lançar uma notificação quando uma construção da aplicação é efetuada ou para aprovar a passagem do código entre ambientes.
- Funciona com qualquer aplicação – é possível fazer a construção de aplicação localmente ou na *cloud* em *Java*, *.NET*, *Windows* e *Linux*. Possui um vasto catálogo de scrips PowerShell.
- É possível personalizar e adicionar funcionalidades – é possível criar *workflows* próprios para personalizar tarefas.

O *Release Management* está incluído com o TFS 2013 ou 2015. Este *software* é instalado na mesma máquina que o TFS.

[2] https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx
[3] https://www.visualstudio.com/en-us/features/release-management-vs.aspx

2.9.2 Jenkins

2.9.2.1 História

O *Jenkins*[4] é uma ferramenta *open-source* número 1 para a integração contínua desenvolvida em *Java* [*Jenkins*, 2015]. O *Jenkins* foi originalmente desenvolvido como o projeto *Hudson*. O projeto *Hudson* começou em 2004 na *Sun Microsystems*. Foi inicialmente lançado em java em 2005 [Kohsuke Kawaguchi, 2007]. Em novembro de 2010, surgiu uma questão na comunidade *Hudson* no que diz respeito à infraestrutura utilizada, que cresceu para abranger questões sobre a gestão e controlo pela *Oracle*. As negociações entre os principais colaboradores do projeto e da *Oracle* aconteceram e o ponto-chave foi a marca registada "*Hudson*". Depois da disputa com a *Oracle* em 2010, foi mudado o nome de *Hudson* para *Jenkins* em 2011 [Andrew Bayer, 2011]. Nesse mesmo ano, a *Oracle* decidiu continuar o desenvolvimento do *Hudson* e considerou o *Jenkins* uma nova ferramenta [Susan Duncan, 2011].

2.9.2.2 Características

Basicamente, o *Jenkins* funciona como um servidor de integração contínua onde é possível programar tarefas para diversos projetos, inclusive sendo possível criar integrações entre essas tarefas.

As principais características do *Jenkins* são:

- Fácil instalação – é apenas preciso fazer o *download* de um executável e mais nada.
- Fácil configuração – pode ser configurado inteiramente a partir de uma interface *web*, sem necessidade de executar comandos.
- *Feedback* – o *Jenkins* fornece meios para mostrar ou enviar por correio eletrónico os resultados sobre as falhas no sistema.
- Testes – produzir relatórios de testes e apresentar graficamente os resultados.
- Balanceamento de carga – distribui a construção da aplicação por diversos servidores para aproveitar o máximo de cada servidor.
- Personalizável – ao *Jenkins* podem ser adicionadas funcionalidades através de *plugins* já criados pela comunidade ou criar novos. Estes *plugins* permitem ao *Jenkins* funcionar com outros projetos que não são desenvolvidos em java.

[4] https://jenkins-ci.org/

3 Melhores práticas para usar integração contínua

Neste capítulo, serão listadas melhores práticas sugeridas pelo Martin Fowler [Martin Fowler, 2006], de modo a alcançar e automatizar a integração contínua.

3.1 Manter um único repositório de código

Os projetos de desenvolvimento de *software* envolvem muitos ficheiros que são necessários de modo a construir um produto. Manter o controlo sobre estes ficheiros é dispendioso, particularmente quando estão muitas pessoas envolvidas num projeto.

A melhor forma de controlar o código fonte é usar um produto de controlo de versões que fornece uma auditoria completa e controlo de versões do código. Tudo o que é necessário com o objetivo de construir um produto, deve estar incluído no sistema controlo de versões, incluindo *scripts*, ficheiros com propriedades da aplicação, configurações da aplicação, dependência sobre aplicações externas, ficheiros para construir a base de dados e o próprio código fonte da aplicação. A regra básica é conseguir construir a aplicação a partir do controlo do código em uma máquina "limpa".

Tudo o que é necessário para construir a aplicação, deve estar incluído no sistema de controlo de versões. No entanto, também pode ser colocado outro material que os elementos da equipa utilizam. As configurações do ambiente de desenvolvimento integrado (IDE) são um bom exemplo para serem colocadas. Assim, os programadores podem partilhar as mesmas configurações e quando chegar um novo programador à equipa, este apenas tem que usar as configurações partilhadas.

Uma das características dos sistemas de controlo de versões é permitir criar múltiplos ramos onde seja possível manipular diferentes fases do desenvolvimento. Esta é uma característica

que, quando bem usada, é útil mas pode trazer problemas à equipa. Os problemas surgem, na maioria dos casos, quando é necessário juntar dois ramos que estão a ser desenvolvidos em paralelo. Num produto grande e complexo, juntar dois ramos de desenvolvimentos pode não ser uma tarefa fácil de realizar.

Em geral, deve-se evitar usar vários ramos durante a mesma fase no desenvolvimento de *software*. Claro que pode haver exceções: se estiverem a ser resolvidos problemas encontrados em produção e a serem adicionadas novas funcionalidades à aplicação em paralelo, é boa prática usar ramos diferentes.

3.2 Automatizar a construção da aplicação

Obter o código fonte e torna-lo num sistema funcional, pode ser um processo complexo que pode envolver compilar o código, mover ficheiros, construir a base de dados, entre outros. No entanto, como a maioria das tarefas individuais nesta parte do desenvolvimento de *software* podem ser automatizadas, o processo como um todo, também deve estar automatizado. Pedir às pessoas para correr comandos na consola ou seguir um conjunto de ecrãs é uma perca de tempo e pode causar erros.

Ambientes automatizados para construir aplicações são uma característica comum de sistemas. Um erro comum é não incluir tudo na construção automática. A construção da aplicação deve incluir a construção da base de dados e colocar em execução os diferentes processos (*website*, serviços, entre outros). A Figura 10 exemplifica um *script* para contruir uma aplicação.

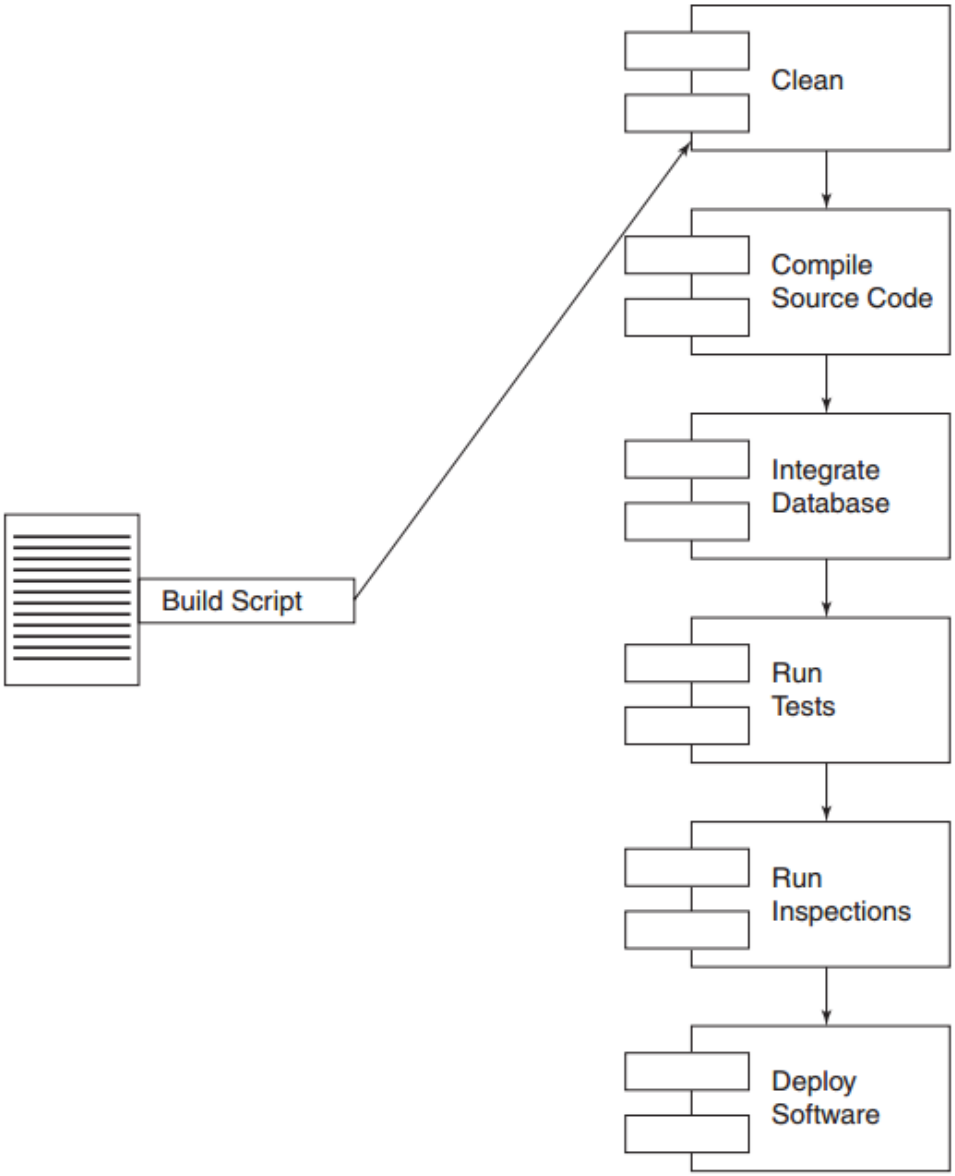

Figura 10 – Construção da aplicação [Paul Duvall, 2007c]

Uma grande construção da aplicação demora tempo e não é necessário efetuar todos os passos se, apenas, uma pequena alteração foi efetuada. Portanto, uma boa ferramenta de construção de aplicações deve analisar que alterações foram efetuadas. Uma forma comum de fazer isso é verificar as datas no sistema de controlo de versões e compilar apenas se houver alterações. As dependências depois podem ser complicadas. Se um objeto sofreu alterações, os restantes objetos que dependem deste, podem também ser preciso compilar novamente. Por isso, dependendo do que é preciso, podem ser necessárias diferentes soluções com o objetivo de construir a aplicação. O caso ideal é conseguir efetuar uma construção completa e parcial da aplicação, poupando assim bastante tempo quando pequenas alterações foram efetuadas na aplicação.

3.3 Incluir testes na construção da aplicação

Em geral, construir a aplicação requer a compilação do código, construir a base de dados, entre outras tarefas que são necessárias para colocar a aplicação a correr. Mas o programa estar a correr, não significa que está tudo correto.

Uma forma de detetar os erros rapidamente e eficazmente é incluir testes durante o processo de construção da aplicação. Os testes não são uma solução perfeita, mas podem apanhar vários erros. Com o crescimento da programação extrema e do desenvolvimento guiado por testes (TDD), tem havido uma grande propagação aos testes automáticos e como resultado, as pessoas têm visto o valor desta técnica. Independentemente de usar ou não a integração contínua em projetos, devemos ter sempre testes na aplicação.

Com o objetivo de realizar testes automáticos, são necessárias ferramentas que realizem testes em uma grande parte do código com a intenção de encontrar erros. Deve ser possível correr os testes a partir de uma simples linha de código na linha de comandos. De modo a que a construção da aplicação seja testável, é preciso que o sucesso da construção da aplicação esteja dependente do resultado dos testes. Se um teste falhar, a construção da aplicação também deve falhar.

Claro que não devemos contar que os testes cubram tudo, até porque os próprios testes podem estar errados. No entanto, testes defeituosos são melhores que testes que nunca chegam a ser escritos.

3.4 Submeter o código frequentemente

A integração permite que os programadores digam entre si, as alterações que produziram. A integração constante permite que os programadores estejam sempre a par do estado dos desenvolvimentos.

Um dos pré-requisitos para um programador submeter o seu trabalho é este não danificar a construção da aplicação. Isto significa que deve compilar corretamente e passar nos testes. Antes de submeter o código, o programador deve primeiro obter as últimas alterações e resolver eventuais conflitos que possam surgir. Após resolver os eventuais conflitos, deve compilar o código e correr os testes na sua máquina local. Apenas, após estes passos todos serem efetuados com sucesso, o programador deve submeter as suas alterações.

Ao submeter frequentemente o código, os programadores podem atempadamente encontrar conflitos com os restantes desenvolvimentos. A chave para resolver problemas rapidamente é encontrá-los o mais cedo possível. Com os programadores a submeter o código frequentemente, os conflitos podem ser encontrados pouco tempo depois de ser introduzidos, o que torna a sua resolução muito mais simples. Conflitos que ficam sem ser detetados durante muito tempo podem ser complicados de resolver. Submeter o código

frequentemente também encoraja os programadores a planear as suas tarefas e a separalas em pequenas tarefas.

3.5 Corrigir erros na construção da aplicação

Uma das vantagens de trabalhar com integração contínua, é saber que se está a trabalhar sobre uma versão estável da aplicação, uma vez que está continuamente a ser testada.

Não é mau quando a construção da aplicação falha, desde que não esteja a falhar muitas vezes. Se a construção da aplicação estiver a falhar constantemente, é sinal que os programadores não estão a ser cuidadosos o suficiente quando submetem o seu código. Quando a construção da aplicação falha, deve ser, assim que possível, corrigida. Na maioria dos casos, a forma mais rápida de corrigir uma construção falhada é reverter o código para a última versão estável.

Após um programador submeter o seu código e a construção da aplicação a seguir deixar de funcionar, isto significa que o problema está muito provavelmente nas alterações que ele efetuou. Neste caso, ele conseguirá corrigir o problema rapidamente, porque sabe o que alterou. Se a construção da aplicação não estiver a funcionar durante muito tempo, o problema pode estar em qualquer parte da aplicação, o que torna demorada a sua correção.

Por vezes, a construção da aplicação está a falhar porque os testes estão a falhar. O impulso é comentar os testes, visto que estes não fazem parte da aplicação de um posto de vista funcional mas, isso não deve ser feito. Se os testes existem, é porque eram necessários com o propósito de testar uma funcionalidade importante da aplicação. Se os testes estiverem a falhar, significa que o código está errado, o teste não está correto ou o teste já não é valido porque a funcionalidade da aplicação já foi removida.

Portanto, para além de corrigir o mais rapidamente possível a construção da aplicação, devemos também, não submeter código quando este não está a funcionar, tornando assim a correção da construção da aplicação mais fácil de descobrir e mais rápida de resolver.

3.6 Manter a construção da aplicação rápida

Um fator chave da integração contínua é fornecer um rápido *feedback*. Por isso, uma construção demorada da aplicação não ajuda. A meta da programação extrema para a duração da construção da aplicação é de 10 minutos. Na maioria dos projetos, este objetivo é possível de concretizar.

Vale a pena fazer algum esforço para conseguir este objetivo. Cada minuto que é reduzido na construção da aplicação, é tempo ganho por cada programador cada vez que ele submete o código.

Como a integração contínua exige submeter o código frequentemente, pode ser perdido muito tempo durante as várias construções da aplicação, por isso deve ser mantida a construção da aplicação o mais rápida possível.

3.7 Testar usando uma réplica de produção

O objetivo dos testes é encontrar sobre condições controladas, algum problema que o sistema possa vir a ter em produção. Se os testes forem efetuados em diferentes ambientes, é possível que tenham resultados diferentes. Como tal, é importante que o ambiente de testes seja uma cópia exata ou o mais aproximadamente possível ao ambiente de produção. Devemos usar o mesmo *software* para a base de dados, dentro das mesmas versões, a mesma versão do sistema operativo e as mesmas versões de *software* externo que a aplicação possa usar. Em casos particulares, devemos até ter o mesmo *hardware* que existe em produção, a fim de testar a aplicação.

Se estivermos a desenvolver uma aplicação móvel, é impossível conseguir testar em todos os modelos de telemóveis que existem no mercado. Neste caso, podemos encontrar os modelos mais usados no mercado de modo a que os nossos testes cubram o maior número de utilizadores possíveis.

Apesar dos limites, o objetivo é conseguir ter um ambiente de teste o mais próximo do ambiente de produção e quando não é possível, é necessário aceitar os riscos de efetuar testes num ambiente diferente que não o de produção.

4 Caso de estudo

Neste capítulo, será descrito detalhadamente o caso de estudo. Irão ser apresentados métodos para efetuar o controlo de versões em base de dados. Serão também descritos os requisitos e as tecnologias utilizadas nas diversas ferramentas propostas para resolver os problemas apresentados e será feita uma descrição de cada ferramenta a desenvolver.

4.1 Contextualização

Anteriormente, para efetuar o controlo de versões do código nos projetos desenvolvidas em *.NET* era usado o *Apache Subversion* (SVN). Hoje em dia, a Konkconsulting usa o *Team Foundation Server* (TFS) no controlo de versões.

Para as aplicações que requerem base de dados (*Oracle* e *SQL Server*), existem bases de dados criadas nos servidores da empresa. Durante os desenvolvimentos, são alteradas diretamente estas bases de dados e as alterações que são efetuadas (*scripts*) são guardadas em um ficheiro onde são posteriormente aplicados em produção. Este ficheiro é partilhado entre os programadores e está guardado no *SharePoint*, no *website* de cada projeto. No capítulo 4.2 são descritas outras formar de efetuar o controlo de versões em bases de dados.

Por norma, apenas existe uma base de dados para cada projeto. Apenas quando é efetuado suporte e desenvolvimento de novas funcionalidades ao mesmo tempo, dentro do mesmo projeto, é criada uma nova base de dados. Após os desenvolvimentos, é necessário unir as alterações das duas bases de dados e apagar a base de dados que foi criada.

Como algumas das bases de dados das aplicações são grandes, torna-se inviável criar uma base de dados local para cada aplicação na máquina dos programadores. Por isso, os programadores usam todos a mesma base de dados remota durante os desenvolvimentos.

Usando esta metodologia de trabalho, estamos constantemente a enfrentar as seguintes dificuldades:

- Alterações de um programador causam erros nos restantes programadores – por exemplo, quando um programador apaga ou muda o nome de uma coluna em uma tabela. O código está à espera de um nome específico de uma coluna e não encontra essa coluna o que irá gerar um erro, tornando a aplicação impossível de usar enquanto o programador que alterou a coluna não submeter o seu código com as alterações da nova coluna. Para resolver este erro, geralmente cada programador tem que comentar a leitura dessa coluna no seu código mas, nem sempre é possível, porque essa coluna pode estar a ser usada em alguma parte da aplicação e depois é possível encontrar comportamentos estranhos na aplicação.

- Quando é reportado um problema em produção, por vezes, não temos a mesma estrutura localmente de modo a conseguir reproduzir o erro – as versões em produção de base de dados, de sistema operativo, as aplicações externas, as permissões de utilizadores e ligações entre máquina são diferentes das usadas em desenvolvimento pelos programadores. Por vezes, é necessário estar ligado a produção e experimentar diferentes soluções para tentar encontrar o problema.

- Quando efetuamos a passagem dos *scripts* para a máquina do cliente, falta algum *script* – após correr os *scripts* na base de dados e atualizar o código fonte da aplicação, a aplicação começa a lançar erros a indicar que faltam colunas numa vista ou até mesmo em uma tabela. A solução nestes casos é correr o *script* à mão em falta.

Com a criação de novos produtos, nos próximos anos pela Konkconsulting, surgiu a oportunidade e necessidade de investigar novas metodologias de trabalho. O trabalho aqui apresentado é fruto de um projeto interno levado a cabo pelo autor na empresa, com o objetivo de melhorar e criar novas ferramentas de trabalho. O foco deste projeto interno foram os novos projetos a serem desenvolvidos em *.NET* pela empresa mas, os primeiros projetos desenvolvidos pela empresa serviram de lição para o que não deve acontecer e deve ser corrigido nos novos projetos.

4.2 Controlo de versões em base de dados

Quando se fala em integração contínua, falamos no código fonte da aplicação. Na maioria dos casos, é esquecida a parte mais crítica de uma aplicação, a base de dados.

Se for feito o *deploy* da versão 2.0 da aplicação e for usada a base de dados na versão 1.0, a aplicação não irá funcionar corretamente. Por isso, os *scripts* da gestão das alterações da base de dados devem estar sempre no sistema de controlo de versões do código fonte.

Quando falamos da gestão das alterações na base de dados, devemos ter em atenção às seguintes condições:

- Garantir que toda a base de dados esteja coberta (estrutura, código, permissões).

- O repositório de controlo de versões deve conter as últimas alterações.
- O *script* conhece o estado do sistema quando está a ser executado.
- O *script,* com as alterações, controla e funde conflitos.
- O *script,* com as alterações, apenas contém as alterações relevantes.
- O *script,* com as alterações, conhece as dependências da base de dados.

Na Konkconsulting, para gerir alterações na base de dados que surgem durante a fase de desenvolvimento e aplicar essas alterações em qualidade ou produção, é usado um ficheiro partilhado entre os programadores com todos os *scripts* produzidos por estes. No entanto, existem outras soluções, como utilizar a simples comparação e sincronização de base de dados ou utilizar um sistema que regista as alterações ocorridas na base de dados. De seguida, serão descritas em mais detalhe estas 3 abordagens.

4.2.1 Utilizar os *scripts* gerados durante os desenvolvimentos

O método mais básico para gerir alterações na base de dados, é guardar os comandos executados durante os desenvolvimentos em um único ficheiro ou conjunto de ficheiros e guardar no sistema de controlo de versões. Isto garante um repositório único que guarda todos os componentes da aplicação. Os programadores tem a mesma função quando estão a submeter alterações da base de dados, como quando estão a submeter alterações do código fonte da aplicação.

Vamos comparar esta solução com as condições enumeradas anteriormente:

- Garantir que toda a base de dados esteja coberta – como são os programadores que escrevem os *scripts*, estes podem garantir que os *scripts* cobrem totalmente a base de dados.
- O repositório de controlo de versões deve conter as últimas alterações – como são os programadores a efetuar o controlo de versão, estes podem alterar a base de dados e não incluir nos *scripts* as alterações efetuadas.
- O *script* conhece o estado do sistema quando está a ser executado – depende do programador e como o *script* está escrito. Se o *script* apenas contém os comandos executados pelos programadores em desenvolvimento ou contém alguma lógica. Por exemplo, o *script* pode adicionar uma coluna que já existe. Escrever *scripts* que estão a olhar para o estado do sistema, aumenta a complexidade do mesmo.
- O *script,* com as alterações, controla e funde conflitos – embora o repositório de controlo de versões dispõe da possibilidade de fundir conflitos, isto é irrelevante para

a base de dados porque o repositório de controlo de versões não é 100% igual à base de dados. A ordem que os *scripts* são executados na base de dados é muito importante.

- O *script,* com as alterações, apenas contém as alterações relevantes – os *scripts* são gerados durante a fase de desenvolvimento. Garantir que os *scripts* contêm apenas alterações relevantes e autorizadas (uma tarefa pode ser rejeitada), requer mudar os *scripts*, o que coloca mais risco e é uma perca de tempo.

- O *script,* com as alterações, conhece as dependências da base de dados – os programadores devem conhecer as dependências da base de dados durante o desenvolvimento dos *scripts*. Se apenas um ficheiro com o *script* está a ser usado, então as alterações são adicionadas ao ficheiro. Isto pode resultar em muitas alterações ao mesmo objeto (por exemplo, uma tabela). Se mais que um ficheiro para os *scripts* está a ser usado, então a ordem dos *scripts* é critica e é mantida manualmente.

O maior problema desta abordagem é depender muito dos programadores. Se algum programador não guardar o *script* executado durante a fase de desenvolvimento ou alterar a ordem de algum *script*, quando for efetuada a passagem para qualidade ou produção, podem ocorrer problemas graves.

4.2.2 Utilizar a comparação e sincronização de base de dados

Outra abordagem comum para fazer a gestão de alterações da base de dados é comparar base de dados de desenvolvimento com a que se pretende atualizar (qualidade ou produção) e gerar o *script* automaticamente.

Esta abordagem permite ganhar tempo aos programadores porque eles não precisam de gerir manualmente o *script*. Se é um *script* de criar ou atualizar, irá depender do estado da base de dados onde se irá executar os *scripts* e será gerado automaticamente.

Vamos comparar esta solução com as condições enumeradas anteriormente:

- Garantir que toda a base de dados esteja coberta – a maioria das ferramentas de comparação e sincronização de base de dados sabem lidar com diferenças entre os objetos mas, apenas algumas conseguem comparar dados.

- O repositório de controlo de versões deve conter as últimas alterações – ferramentas de comparação e sincronização de base de dados não utilizam controlo de versões. Quando é necessário, elas comparam a última versão da base de dados e geram os *scripts*.

- O *script* conhece o estado do sistema quando está a ser executado – a boa prática, é gerar os *scripts* momentos antes da execução. Assim, é praticamente garantido que o estado da base de dados é tido em conta na geração de *scripts*.

- O *script,* com as alterações, controla e funde conflitos – as ferramentas de comparação e sincronização de base de dados comparam duas bases de dados (origem e destino). Partindo da base de dados de origem, a ferramenta gera o *script* para fazer a atualização na base de dados de destino. Sem saber a origem da alteração, o *script* errado pode ser gerado. Por exemplo, se existe um índice que foi criado na base de dados destino de um diferente ramo ou de uma correção crítica. Se este índice não existe na base de dados de origem, o que deve ser feito? Remover o índice? Usar uma ferramenta de comparação e sincronização de base de dados, requer conhecimento profundo das alterações de cada alteração para garantir que são tratadas adequadamente.

- O *script,* com as alterações, apenas contém as alterações relevantes – a ferramenta de comparação e sincronização de base de dados compara a base de dados toda e mostra as diferenças. Elas não sabem a razão de origem destas alterações porque esta informação está guardada num sistema de controlo de versões que é externo à ferramenta. Pode ser bastante difícil determinar quais são as alterações relevantes a aplicar nos ambientes de qualidade ou produção.

- O *script,* com as alterações, conhece as dependências da base de dados – as ferramentas de comparação e sincronização de base de dados estão a par das dependências e geram os *scripts* relevantes, na ordem correta.

As ferramentas de comparação e sincronização de base de dados cumprem a maioria das condições. O maior problema destas ferramentas é não conseguir fazer a associação entre a alteração do código fonte da aplicação e a alteração da base de dados.

4.2.3 Utilizar um sistema que regista as alterações ocorridas na base de dados.

Um sistema que regista as alterações ocorridas na base de dados combina o uso de controlo de versões com a geração de *scripts* quando necessário.

Vamos comparar esta solução com as condições enumeradas anteriormente:

- Garantir que toda a base de dados esteja coberta – a maioria dos sistemas cobre toda a estrutura da base de dados.

- O repositório de controlo de versões deve conter as últimas alterações – ao forçar que as alterações na base de dados não sejam possíveis de efetuar usando a linha de comandos, mas sim dentro de um sistema que efetua o controlo dos *scripts* e ligação com o repositório de gestão de versões, é garantido que o repositório está atualizado.

- O *script* conhece o estado do sistema quando está a ser executado – a boa prática é gerar os *scripts* momentos antes da execução. Assim, é quase garantido que o estado da base de dados é tido em conta na geração de *scripts*.

- O *script,* com as alterações, controla e funde conflitos – ao ter a ligação entre a correção no código e a alteração na base de dados, a origem da alteração é conhecida e é possível saber o que fazer em cada situação.

- O *script,* com as alterações, apenas contém as alterações relevantes – com a ligação ao repositório, é possível conhecer a razão da alteração e saber se é relevante ou não.

- O *script,* com as alterações, conhece as dependências da base de dados – o sistema está a par das dependências e gere os *scripts* relevantes na ordem correta.

Esta solução resolve o maior problema da comparação entre base de dados: saber a origem das alterações, visto que tem ligação ao repositório das alterações do código fonte. O problema desta solução é depender de um produto externo que habitualmente não é barato e requer mudar os hábitos dos programadores. Cada produto tem a sua implementação e pode lidar de forma diferente com o mesmo problema.

4.3 Requisitos

O trabalho apresentado neste documento foi realizado em duas fases.

A primeira fase foi em 2013, quando a empresa começou a usar o TFS e deixou de usar o SVN. Nessa fase, foi definido o objetivo de manter a compatibilidade com as ferramentas de desenvolvimento existentes que usavam o SVN, nomeadamente o *XCode*[5].

A segunda fase foi em finais de 2014, quando surgiu a necessidade de construir um ambiente de integração contínua, onde os responsáveis do projeto possam testar os desenvolvimentos do novo produto a ser desenvolvido pela empresa e resolver as dificuldades acima referidas. Nesta fase, a empresa definiu as seguintes metas:

- Em qualquer versão de código, conseguir criar uma base de dados com essa mesma versão de código com o objetivo de testar ou resolver erros.

- Com a intenção de efetuar a gestão de dados mestres a serem usados nas aplicações, criar um *add-in* no *Excel*. Este *add-in* gera *scripts* que inserem dados na base de dados a partir dos dados presentes nas folhas de *Excel*.

- Construir um ambiente de integração contínua com atualizações automáticas onde é possível aos gestores de projeto testar a aplicação com os desenvolvimentos daquela

[5] https://developer.apple.com/xcode/

semana. Para a construção da aplicação, é necessário criar um *website* em *.NET* alojado no *Internet Information Services* (IIS)[6] e uma base de dados.

Uma condição comum a todas as ferramentas que se possam vir a usar é serem gratuitas. Nos seguintes subcapítulos, serão apresentadas e descritas as soluções encontradas para dar resposta às metas propostas pela empresa.

4.4 Arquitetura

O sistema proposto de modo a cumprir o objetivo de colocar a integração contínua a funcionar na empresa é na verdade um conjunto de ferramentas em que a sua utilização está relacionada. Apesar de a utilização das ferramentas estarem relacionadas, elas podem funcionar separadamente.

Como objetivo no desenvolvimento destas ferramentas, não há dependências funcionais entre as diversas ferramentas. Cada ferramenta foi desenvolvida separadamente, focando-se no objetivo que tem que cumprir. O diagrama completo de componentes e as suas interações está ilustrado na Figura 11.

[6] https://www.iis.net/

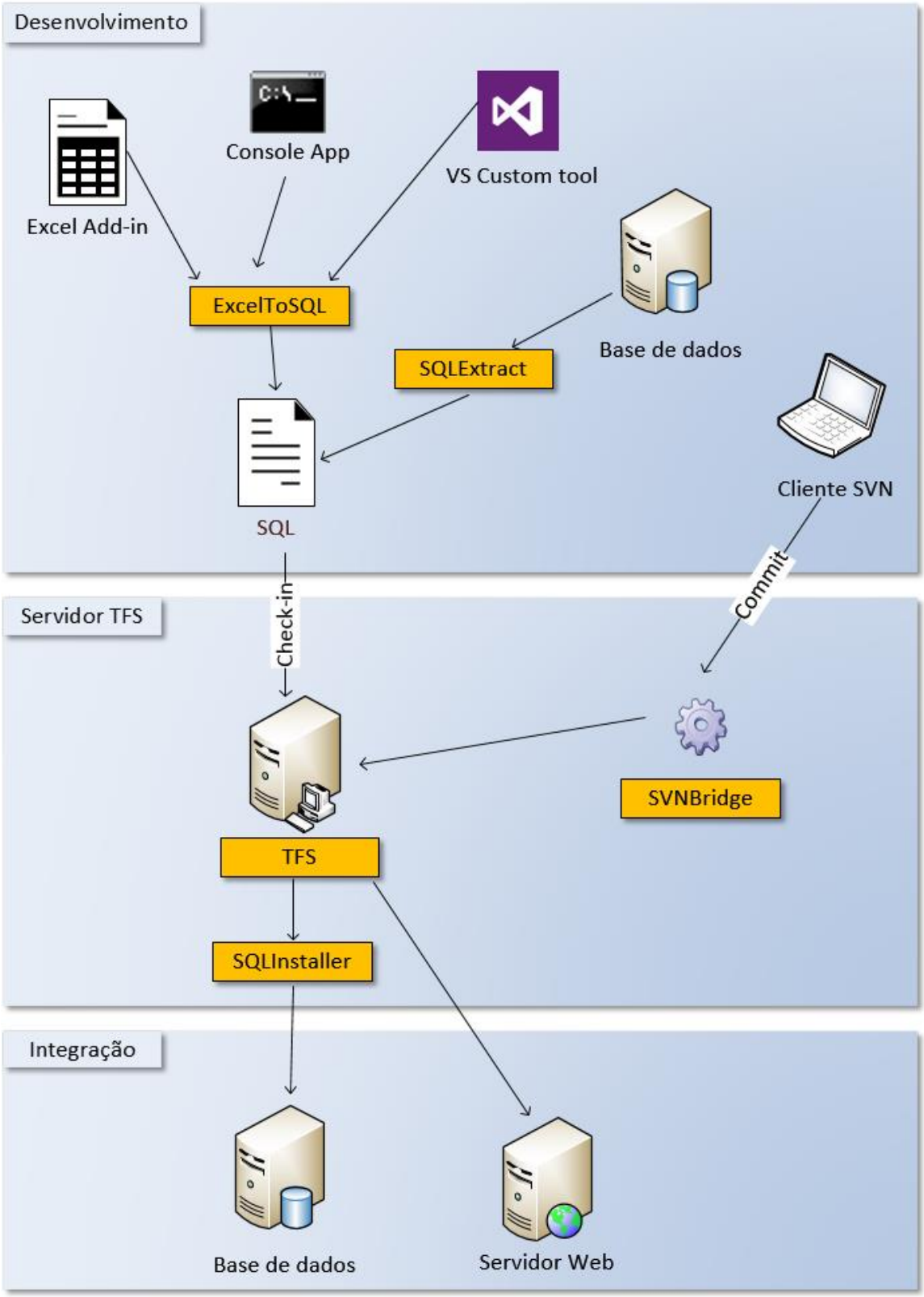

Figura 11 – Interações entre ferramentas

Como é possível observar na Figura 11, os diversos componentes terão papéis importantes no sistema e estão focados a resolver uma tarefa específica. Contudo, não é necessário utilizar todos os componentes. Por exemplo, o *SVNBridge* apenas será utilizado se o programa cliente de gestão de versões de código fonte for SVN e não TFS. O *SQLExtract* será apenas utilizado quando for necessário extrair os *scripts* iniciais de uma base de dados onde os desenvolvimentos já estão adiantados. Quando o projeto começar do zero, não será necessário utilizar esta ferramenta.

Os diversos componentes que fazem parte do sistema serão descritos nos capítulos seguintes.

4.4.1 SVNBridge

O *SVNBridge*[7] é usado para clientes SVN comunicarem com o TFS. O código fica guardado no TFS e para o cliente SVN, a comunicação é transparente.

Mantendo a compatibilidade com os clientes SVN, foi possível migrar os repositórios do código fonte SVN para o TFS. Atualmente é usado em um projeto para o *iOS*.

4.4.2 SQLInstaller

O *SQLInstaller*[8] é usado para construir a base de dados. Cada programador pode agora facilmente construir a sua base de dados local e usá-la durante os desenvolvimentos da aplicação.

Também é usado para o programador testar os seus *scripts*. Antes de submeter o código, cada programador pode correr o programa e executar os *scripts* de modo a testar os mesmos e saber se está tudo correto.

4.4.3 ExcelToSQL

O *ExcelToSQL* gera *scripts* para inserir dados em tabelas. Com esta ferramenta, agora é possível fazer uma gestão fácil dos dados mestres, numa folha de *Excel*, para posteriormente serem inseridos na base de dados.

A construção inicial da aplicação pode incluir facilmente a inserção destes dados que são gerados em *scripts* de inserção na base de dados. Também está a ser usado quando é preciso inserir dados de múltiplas fontes. Inicialmente, são agrupados e trabalhados esses dados no *Excel* sendo posteriormente inseridos na base de dados.

[7] https://svnbridge.codeplex.com/
[8] https://sqlinstaller.codeplex.com/

O *ExcelToSQL* terá um *add-in* para *Excel*, uma aplicação consola e uma *Custom Tool* no *Visual Studio*.

4.4.4 SQLExtract

O *SQLExtract* extrai os *scripts* da estrutura de uma base de dados de modo a reconstruír novamente. É usado em projetos mais antigos, onde facilmente se pode extrair os *scripts* e construir localmente, na máquina de cada programador, a base de dados.

Também é possível ligar à base de dados do produto em produção e obter uma cópia exata da estrutura da base de dados e usá-la para comparar ou resolver problemas.

4.4.5 TFS

O TFS é usado como repositório do código fonte e para construir a aplicação. Este usa o *SQLInstaller* para construir a base de dados e compila o código fonte que é necessário de modo a construir o *website*. Atualmente, o TFS está a ser usado pela Konkconsulting nos desenvolvimentos do seu novo produto como servidor de integração contínua.

Para além de conseguir introduzir a integração contínua no novo produto a ser desenvolvido pela empresa e responder às suas exigências, também foram resolvidos alguns dos problemas que a empresa tinha durante os desenvolvimentos, nomeadamente as alterações de um programador causar erros nos restantes programadores. Outro aspeto importante destas ferramentas é serem autónomas, podendo facilmente ser usadas separadamente.

4.5 Tecnologias e ferramentas utilizadas

Nos desenvolvimentos das ferramentas descritas neste capítulo, foram usadas as seguintes tecnologias e ferramentas:

- *Visual Studio 2010 SP1*
- *Team Foundation Server 2013*
- *.NET Framework 4*
- *Microsoft SQL Server 2012*
- *Oracle Database 11g Release 2*
- *Internet Information Services 7*
- *Windows 8.1*

4.6 Sumário

Não existe um método perfeito para efetuar o controlo de versões em bases de dados. Desde ter um ficheiro mantido pelos programadores a ter um produto que regista as alterações na base de dados, há sempre vantagens e desvantagens em cada abordagem escolhida. Por isso, é preciso em cada situação, analisar o que é necessário e escolher a melhor solução.

A solução escolhida foi continuar a usar a gestão manual de *scripts*, mas guarda-los de uma forma diferente. Anteriormente, eram usados 2 ficheiros para guardar os *scripts* que eram produzidos pelos programadores durante a fase de desenvolvimento. O nome dado aos 2 *scripts* era:

1. {Nome_do_projecto}_DDL – neste ficheiro, estavam os comandos que manipulavam a estrutura da base de dados (tabelas, vistas, funções, entre outros).

2. {Nome_do_projecto}_DML – neste ficheiro, estavam os comandos de manipulação de dados.

No entanto, muitas vezes os *scripts* estavam misturados num único ficheiro porque não havia controlo, nem atenção por partes dos programadores.

Usando o *SQLInstaller*, os *scripts* serão melhor organizados. Para além de estarem melhor organizados e estruturados, será possível executa-los usando o *SQLInstaller* para saber se existe algum erro, algo que anteriormente não era possível. Nessa altura, apenas eram detetados erros quando os *scripts* estavam a ser executados em qualidade ou produção, o que causava bastantes problemas para os corrigir posteriormente. Foi escolhido o *SQLInstaller* sobretudo por ser uma ferramenta aberta, sendo assim possível fazermos os ajustes necessários para o que necessitamos, de modo a o usar nos nossos projetos. Usando outros produtos que existem no mercado, estaríamos limitados às funcionalidades que esse mesmo produto possa oferecer.

Com o *add-in* para *Excel*, será possível incluir facilmente nos *scripts* inserções massivas de dados algo que, anteriormente era uma tarefa demorosa e trabalhosa. Normalmente, o cliente enviava um *Excel* com os dados que queria que fossem inseridos no lançamento da aplicação. Agora, usando o *add-in,* é fácil e rápido criar *scripts* para inserir esses dados.

No próximo capítulo, será descrito detalhadamente o processo de implementação das diversas ferramentas, com o objetivo de colocar a integração contínua a funcionar em um projeto que irá ser desenvolvido pela empresa.

5 Solução desenvolvida

Neste capítulo, serão descritas todas as soluções implementadas de modo a introduzir a integração contínua no projeto que irá ser construído pela empresa. Para cada ferramenta, será efetuada uma descrição do seu desenvolvimento e serão apresentadas as dificuldades e decisões tomadas que surgiram durante a sua criação.

5.1 SVN para TFS

Com o objetivo de centralizar o código fonte das aplicações e aumentar a produtividade dos programadores, foi decidido passar a usar apenas o TFS como repositório de código para as aplicações desenvolvidas pela empresa. Como os programadores já usavam o *Visual Studio* para programar, a escolha de TFS foi óbvia.

A maioria das aplicações que são desenvolvidas pela Konkconsulting estão a ser desenvolvidas usando o *Visual Studio*, portanto a utilização do TFS simplifica o processo de gestão do código fonte.

O programa usado na gestão do código fonte das aplicações nos repositórios SVN era o *TortoiseSVN*[9]. Usando este programa, era necessário aceder à localização no disco onde se encontra o código fonte e a partir do explorador do Windows, submeter o código usando a opção *SVN Commit* ou obter a última versão do código fonte usando a opção *SVN Update*. A Figura 12 mostra estas duas opções.

[9] http://tortoisesvn.net/

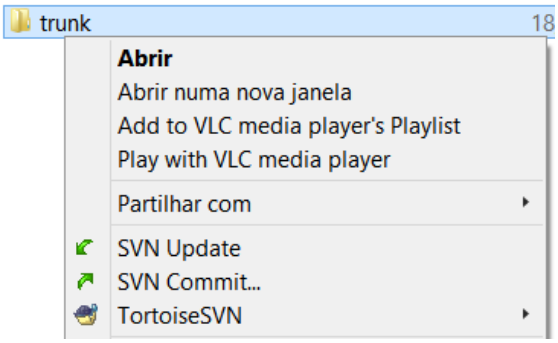

Figura 12 – Submeter e obter código usando o *TortoiseSVN*

Após selecionar a opção *SVN Commit*, aparece uma nova janela onde é possível introduzir uma pequena descrição do que foi alterado. A Figura 13 mostra essa janela.

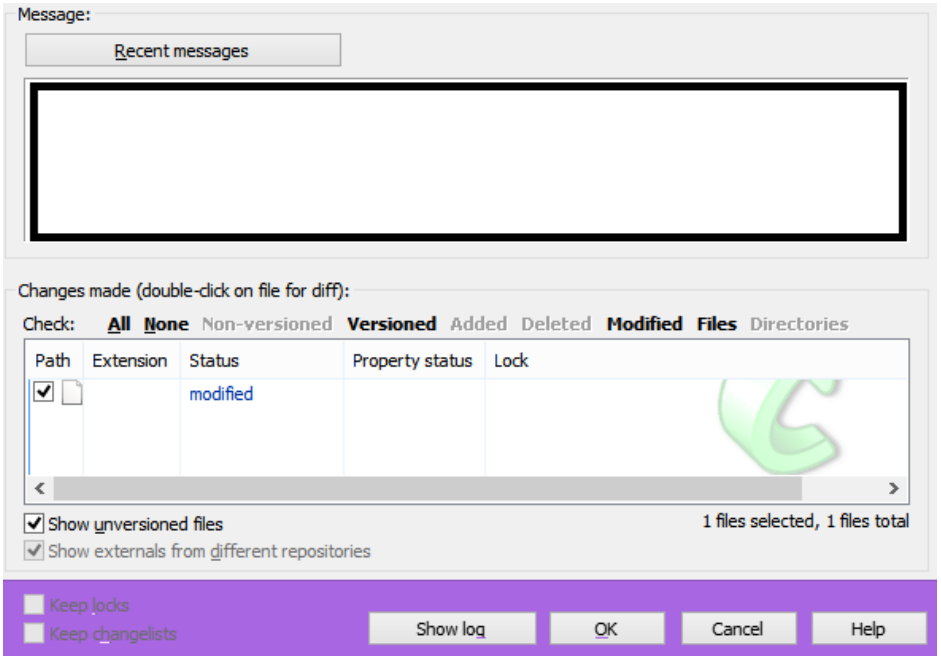

Figura 13 – Descrição das alterações no código fonte no *TortoiseSVN*

Usando o *Visual Studio* e o TFS como repositório do código fonte, é possível efetuar toda a gestão do código fonte dentro do IDE, o que é mais eficaz e menos demorado para os programadores. No *Visual Studio*, para obter a última versão do código fonte, é necessário abrir a janela *Team Explorer*. Dentro do *Team Explorer,* é necessário escolher o projeto e selecionar a opção *Get Latest Version*. A Figura 14 mostra estas opções.

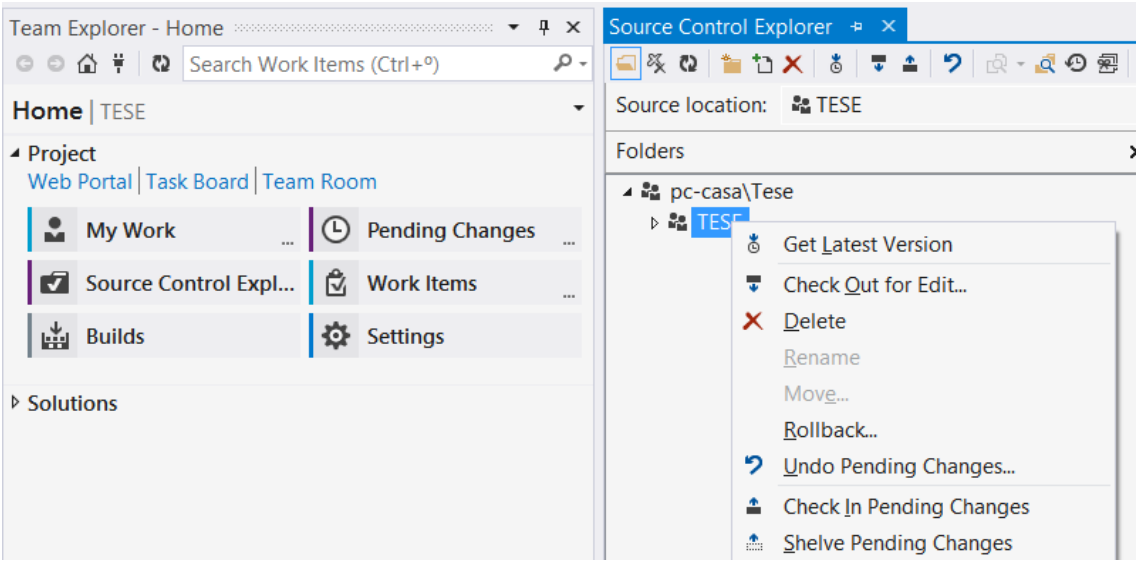

Figura 14 – Obter a última versão do código fonte usando o *Visual Studio*

Para submeter código no *Visual Studio,* é necessário aceder à janela *Team Explorer* e selecionar a opção *Pending Changes*. Dentro das *Pending Changes,* é possível escrever uma descrição das alterações efetuadas no código fonte. A Figura 15 exibe esta opção.

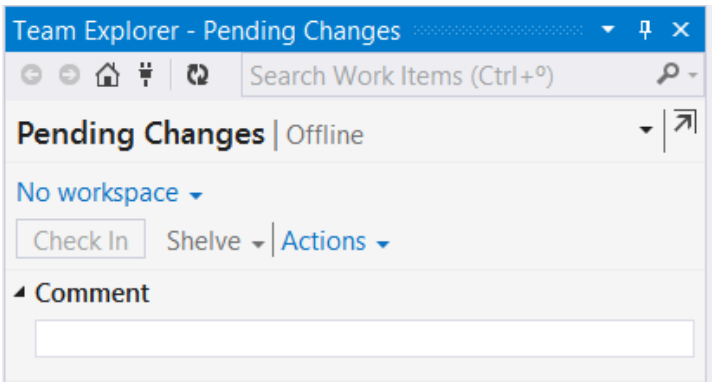

Figura 15 – Submeter código no *Visual Studio*

A mudança para o TFS do código fonte, para os programadores que usam o *Visual Studio,* não trouxe alterações nos seus hábitos de trabalho nem necessidade de mudar a infraestrutura que é usada para desenvolver aplicações. Nos restantes programadores, a empresa pretendia o mesmo.

Atualmente, apenas os programadores *iOS* não usam o *Visual Studio* como IDE, mas sim o *XCode*. Como era necessário encontrar uma solução no *XCode* e eventualmente clientes de SVN conseguirem comunicar com o TFS, foi escolhida a ferramenta *SVNBridge*. Em 2013, quando foi efetuada a mudança pela empresa para o TFS, o *XCode* não tinha suporte de repositórios GIT. O suporte deste tipo de repositórios foi adicionado ao *XCode* posteriormente.

5.1.1 SVNBridge

O *SVNBridge* é uma ferramenta que permite usar o *TortoiseSVN* ou qualquer outro cliente de SVN com o TFS. Esta ferramenta converte as chamadas efetuadas pelos clientes SVN para a interface do TFS. Esta ferramenta é gratuita e atua como ponte entre o cliente SVN e o TFS. O *SVNBridge* fica a comunicar com o TFS e o cliente SVN comunica com o *SVNBridge*. Isto permite a qualquer cliente SVN comunicar com o TFS. A Figura 16 ilustra estas comunicações.

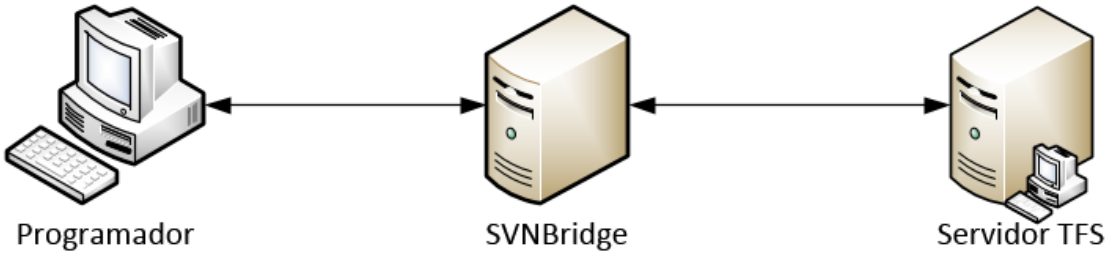

Figura 16 – Comunicações entre o programador, *SVNBridge* e servidor TFS

O *SVNBridge* tem uma versão cliente e servidor. A versão cliente corre como um executável no sistema e a versão servidor corre sobre o IIS.

5.1.2 Criação do servidor *SVNBridge*

Neste subcapítulo, serão enumerados os passos que foram necessários de modo a criar o servidor *SVNBridge*. Este servidor foi criado na mesma máquina onde está o servidor TFS instalado na empresa.

1. Efetuar o *download* do *SVNBridge*.
    a. O *SVNBridge* está disponível para *download* gratuitamente em svnbridge.codeplex.com.
2. Criar uma pasta onde ficaram os ficheiros do servidor *SVNBridge* e copiar para lá os ficheiros que foram obtidos no passo anterior. A Figura 17 mostra o conteúdo que deverá ter esta pasta.

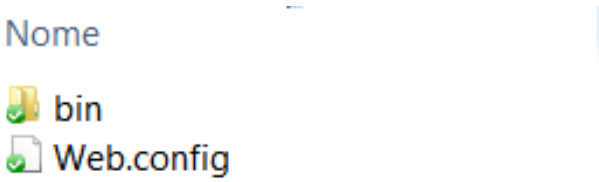

Figura 17 – Ficheiros do servidor *SVNBridge*

3. Abrir o ficheiro *web.config* e editar as configurações. Na Figura 18 é possível ver as configurações que são usadas atualmente pela empresa.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appSettings>
        <add key="TfsUrl" value="http://tfs_server:8080/tfs/projects" />
        <add key="LogPath" value="E:\SourceCode\svn_tfs" />
        <add key="DomainIncludesProjectName" value="False" />
        <add key="ReadAllUserDomain" value="konkconsulting" />
    </appSettings>
    <system.web>
        <httpRuntime maxRequestLength="500000" />
        <customErrors mode="Off" />
        <compilation debug="true" />
        <authentication mode="Windows" />
        <identity impersonate="true" />
    </system.web>
    <system.net>
        <defaultProxy enabled="true" />
    </system.net>
    <system.webServer>
        <validation validateIntegratedModeConfiguration="false" />
        <modules>
            <remove name="WebDAVModule" />
            <remove name="ServiceModel" />
        </modules>
        <handlers>
            <clear />
            <add name="SvnBridgeHandler" path="*" verb="*" responseBufferLimit="0"
            type="SvnBridgeServer.SvnBridgeHttpHandler" resourceType="Unspecified" preCondition="integratedMode" />
        </handlers>
        <security>
            <requestFiltering>
                <requestLimits maxAllowedContentLength="262144000" />
                <fileExtensions>
                    <clear />
                </fileExtensions>
                <hiddenSegments>
                    <clear />
                </hiddenSegments>
            </requestFiltering>
        </security>
    </system.webServer>
</configuration>
```

Figura 18 – *SVNBridge* web.config

4. Abrir o IIS e criar uma *application pool* que será usado no servidor do *SVNBridge.* A Figura 19 contém as configurações necessárias.

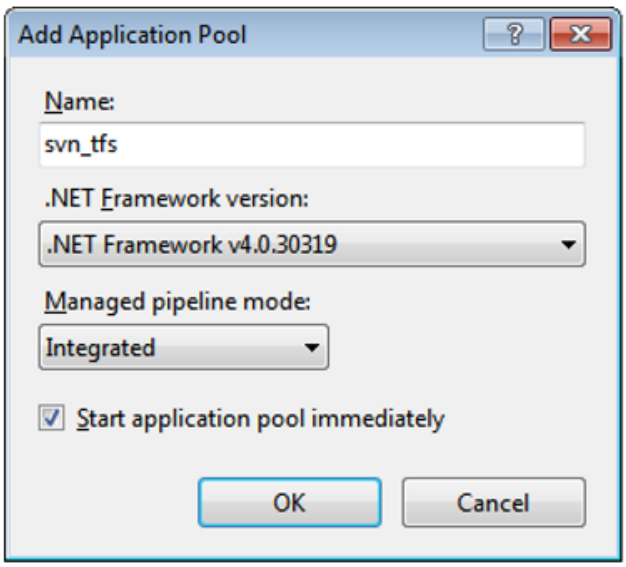

Figura 19 – Configurações da *application pool* do *SVNBridge*

5. No IIS, criar um novo *website* para o *SVNBridge*. A localização dos ficheiros deverá ser a pasta que foi criada no passo 2. O nome usado foi "svn_tfs". A Figura 20 mostra estas opções.

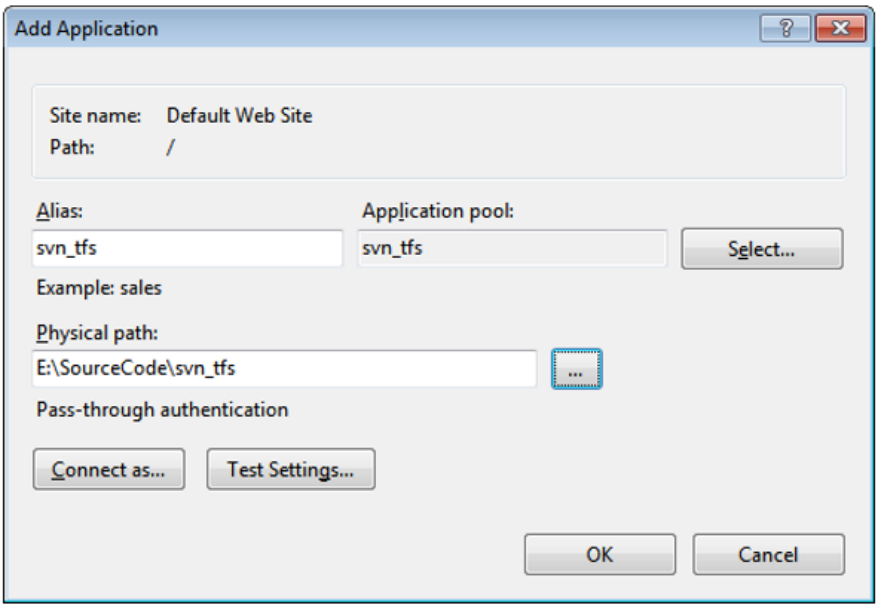

Figura 20 – Criação do *website* do *SVNBridge*

6. No IIS, alterar a autenticação para *ASP.NET Impersonation* e *Basic Authentication*. Isto é necessário para conseguir autenticar os clientes do servidor SVN com o TFS da empresa. A Figura 21 mostra as opções de autenticação disponíveis.

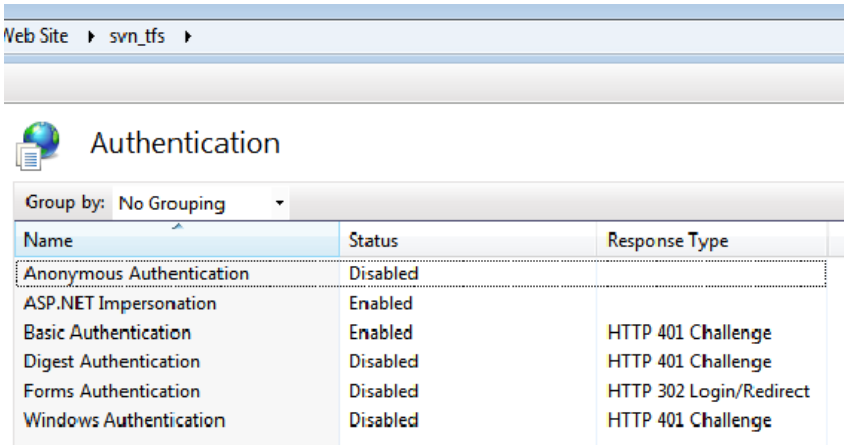

Figura 21 – Autenticação no *website* do *SVNBridge*

Para verificar se está tudo a funcionar corretamente, é necessário abrir o *browser* e aceder ao URL "localhost/SNV_TFS/". Se tudo estiver a funcionar corretamente, deverá aparecer uma pasta com os projetos disponíveis no TFS. Usando o *browser,* também é possível navegar no repositório. A Figura 22 mostra um exemplo de uma lista de projetos.

Figura 22 – Lista de projetos do TFS usando o *SVNBridge*

No *browser,* ao selecionar o projeto é possível ver o conteúdo do mesmo. A Figura 23 mostra o conteúdo de um projeto.

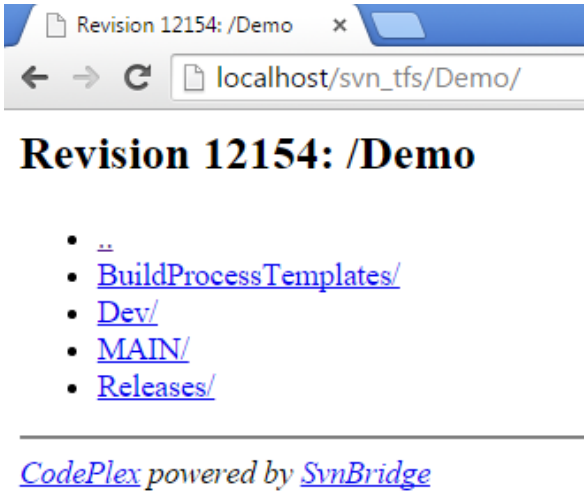

Figura 23 – Conteúdo de um projeto usando o *SVNBridge*

Para usar o *SVNBridge*, apenas é necessário utilizar o URL deste servidor seguido do nome do projeto (Exemplo: localhost/svn_tfs/demo) como URL do repositório. Os clientes SVN irão comunicar com o *SVNBridge* e este comunicará com o TFS.

5.2 Construção automática da base de dados

Com a introdução da integração contínua e com a necessidade de conseguir criar uma base de dados para uma versão de código, surgiu a necessidade de encontrar uma solução para conseguir construir a base de dados das aplicações durante a construção da aplicação. A solução encontrada foi a utilização do programa *SQLInstaller*.

O *SQLInstaller* é um simples programa que executa *scripts* de base de dados. Foi escolhido este programa devido essencialmente a 3 fatores:

- Suporte de *Oracle* e *SQL Server* que são as bases de dados usadas pela empresa.
- O código fonte está disponível gratuitamente no codeplex.com – caso seja necessário implementar alguma funcionalidade específica necessária para introduzir integração contínua nos seus projetos ou corrigir algum erro encontrado.
- Visto que é um simples e pequeno executável, pode ser facilmente integrado no TFS.

5.2.1 SQLInstaller

Quando falamos de código fonte, geralmente é usado uma classe por ficheiro e são usadas pastas que agrupam logicamente as várias classes. A ideia por traz do *SQLInstaller* é a mesma, mas em vez de falar em classes, falamos em objetos (tabelas, vistas, procedimentos, funções, entre outros).

Como o objetivo é tornar a criação da base de dados consistente, determinística e repetível, o programa utiliza uma estrutura de pastas e ficheiros que são executados por ordem. A Figura 24 ilustra a típica estrutura de pastas que é utilizada no *SQLInstaller*.

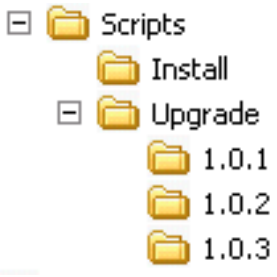

Figura 24 – Estruturas de pastas do *SQLInstaller*

A pasta *Install* contém todos os *scripts* para criar a base de dados a partir do zero. A pasta *Upgrade* contém os *scripts* necessários para efetuar a atualização da base de dados de versão em versão. Por defeito, uma nova base de dados instalada a partir apenas da pasta *Install* é a versão RTM. Em seguintes execuções do *SQLInstaller,* para atualizar a base de dados, este reconhece que existe a base de dados e executa os *scripts* na pasta *Upgrade* começando na versão atual e acabando na última pasta que se irá tornar a versão atual da aplicação (versão 1.0.3 na Figura 24).

É possível também começar numa determinada versão e fazer a gestão dos *scripts* a partir apenas da pasta Upgrade. Neste cenário, a versão mais baixa presente na pasta *Upgrade* será executada durante a criação da base de dados (1.0.1 na Figura 24) e todas as restantes pastas serão executadas em sequência. A maior vantagem desta abordagem é os *scripts* não ficarem duplicados nas pastas *Install* e *Upgrade*. A maior desvantagem é ser feita uma migração total, independentemente de ser uma instalação limpa ou uma atualização.

5.2.1.1 Extensões

O *SQLInstaller* usa uma convenção de nomes de extensões para diferenciar os diferentes tipos de objetos na base de dados. As 8 extensões por defeito são:

1. *PreInstall* – contém qualquer pré-requisito. (por exemplo: tipos de dados).
2. *Table* – contém as tabelas.
3. *UserDefinedFunction* – contém as funções.
4. *View* – contém as vistas.
5. *StoredProcedure* – contém os procedimentos.
6. Trigger – contém os triggers.

7. *PostInstall* – contém qualquer *script* que seja necessário correr depois da instalação (Por exemplo: dados mestre).

8. *ForeignKey* – contém as chaves estrangeiras das tabelas.

Estas 8 extensões que estão definidas por defeito podem ser personalizadas.

5.2.1.2 Formatação

Além das convenções de nomes nos ficheiros impostas pelo *SQLInstaller*, existe uma restrição especial para *scripts Oracle* e IBM DB2. Nos *scripts*, é necessário um caracter de terminação especial de modo a diferenciar entre o fim de um procedimento das declarações individuais. Os programadores devem incluir uma barra "/" para indicar o fim do bloco. O seguinte excerto de código contém um exemplo.

```
BEGIN
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (1, 'Beverages', 'Soft drinks, coffees, teas, beers, and ales');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (2, 'Condiments', 'Sweet and savory sauces, and seasonings');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (3, 'Confections', 'Desserts, candies, and sweet breads');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (4, 'Dairy Products', 'Cheeses');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (5, 'Grains/Cereals', 'Breads, crackers, pasta, and cereal');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (6, 'Meat/Poultry', 'Prepared meats');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (7, 'Produce', 'Dried fruit and bean curd');
INSERT INTO Categories (CategoryID, CategoryName, Description)
VALUES (8, 'Seafood', 'Seaweed and fish');
END;
/
```

Código 1 – Terminação de um *script* no *SQLInstaller*

5.2.1.3 Execução

Para executar o *SQLInstaller*, é apenas necessário correr o executável. Os parâmetros do programa podem ser passados diretamente na linha de comandos ou é passado um parâmetro com o caminho de um ficheiro que contém todas as configurações.

A Tabela 1 contém todos os parâmetros do *SQLInstaller*.

Tabela 1 – Parâmetros do *SQLInstaller*

Parâmetro	**Alternativa**	**Descrição**

/ConnectionString	*/conn*	Conexão da base de dados.
/Database	*/d*	Nome da base de dados para criar ou atualizar.
/InstallPath	*/ins*	Caminho relativo da pasta que contém os ficheiros de criação da base de dados.
/FileTypes	*/f*	Lista ordenada, separada por vírgulas, com o nome dos tipos de ficheiro usados durante a execução.
/NoPrompt	*/nop*	Pedir confirmação para efetuar a atualização.
/Options	*/o*	Lista separada por vírgulas com as opções disponíveis: • *Create* – criar a base de dados. • *Drop* – apagar a base de dados. • *Retry* – tentar a última atualização. • *Verbose* – mostrar todas as mensagens. • *ExitCode* – alterar o código de terminação do processo dos erros de SQL.
/Provider	*/p*	Tipo da base de dados: *SQL Server, Oracle, PostGres, DB2, FireBird, MySQL, SQLite, Teradata*
/ScriptExtension	*/ext*	Extensão dos ficheiros de *script.*
/UpgradePath	*/upg*	Caminho relativo da pasta que contém os *scripts* para a atualização da base de dados.
/WriteConfig	*/w*	Indica se é para escrever em ficheiro as configurações passadas na linha de comandos.

O código seguinte ilustra as mesmas configurações mas em formato *XML*.

```
<Parameters Options="Create" IsProtected="false" NoPrompt="false"
ScriptExtension="sql" InstallPath="Install" UpgradePath="Upgrade">
   <Database>DATABASE_NAME</Database>
   <Provider Name="SqlServer|Oracle" />
   <ConnectionString>Data Source=dbsrv;Integrated
Security=SSPI</ConnectionString>
   <FileTypes>
      <FileType Name="Table" Description="TABLES" IsDisabled="false"
IsGlobal="false" HaltOnError="false"/>
   </FileTypes>
</Parameters>
```

Código 2 – Configurações do *SQLInstaller* em ficheiro

O processo de instalação do *SQLInstaller* espera que seja criada a base de dados a partir do zero. Durante os desenvolvimentos, geralmente é adicionada a opção *Drop* para apagar a base

de dados sempre que é executado o programa. Em produção, obviamente não queremos apagar a base de dados. Nesse caso, o programa irá detetar a versão da base de dados e não executar qualquer *script*.

Com o objetivo de identificar o número da versão de base de dados e atualizar esse mesmo número, o *SQLInstaller* executa *scripts* na base de dados. Estes *scripts* estão presentes no ficheiro *ProviderFactory.xml* e podem ser alterados, se for necessário. O Código 3 contém a configuração por defeito presente no ficheiro da base de dados *Oracle*.

```
<Provider Name="Oracle" InvariantName="System.Data.OracleClient">
<Scripts>
<Script Type="Exists">
 SELECT COUNT(*) FROM all_users WHERE username = UPPER('{0}')
</Script>

<Script Type="Drop">
 DROP USER {0} CASCADE
</Script>

<Script Type="Create">
 BEGIN
 EXECUTE IMMEDIATE 'CREATE USER {0} IDENTIFIED EXTERNALLY';
 EXECUTE IMMEDIATE 'GRANT UNLIMITED TABLESPACE TO {0}';
 END;/
</Script>

<Script Type="GetVersion">
 SELECT VERSION_INFO || ';' || UPGRADEBY FROM {0}.DB_VERSION
</Script>

<Script Type="SetVersion">
 CREATE OR REPLACE VIEW {0}.DB_VERSION AS SELECT '{1}' AS VERSION_INFO,
'{2}' AS UPGRADEBY FROM DUAL
</Script>

</Scripts>
</Provider>
```

Código 3 – Configuração por defeito do *SQLInstaller* em *Oracle*

O Código 4 contém a configuração por defeito presente no ficheiro ProviderFactory.xml para a base de dados *SQL Server*.

```
<Provider Name="SqlServer" InvariantName="System.Data.SqlClient">
<Scripts>
 <Script Type="Exists">SELECT COUNT(*) FROM sysdatabases WHERE '[' + name +
']' = QUOTENAME('{0}')</Script>
 <Script Type="Drop">ALTER DATABASE [{0}] SET SINGLE_USER WITH ROLLBACK
IMMEDIATE; DROP DATABASE [{0}];</Script>
 <Script Type="Create">
        USE master
        GO
        DECLARE @data_path nvarchar(512);
        SET @data_path = (SELECT SUBSTRING(physical_name, 1,
CHARINDEX(N'master.mdf', LOWER(physical_name)) - 1)
        FROM master.sys.master_files
```

```
        WHERE database_id = 1 AND file_id = 1);
        -- Create database
        EXECUTE( 'CREATE DATABASE [{0}] ON  PRIMARY
        ( NAME = [{0}], FILENAME = ''' + @data_path + '{0}.mdf'' )
        LOG ON
        ( NAME = [{0}_log], FILENAME = ''' + @data_path + '{0}_log.ldf'' )'
        )
        GO
 </Script>
 <Script Type="GetVersion">
        SELECT CAST(ISNULL(V.value,'') AS VARCHAR(255)) + ';' +
CAST(ISNULL(U.value,'') AS VARCHAR(1024))
        FROM fn_listextendedproperty('Version', default, default, default,
default, default, default) V,
        fn_listextendedproperty('UpdatedBy', default, default, default,
default, default, default) U
 </Script>
 <Script Type="SetVersion">
        IF NOT EXISTS (SELECT value FROM fn_listextendedproperty('Version',
default, default, default, default, default, default))
        EXEC sp_addextendedproperty 'Version', '{1}'
        ELSE
        EXEC sp_updateextendedproperty 'Version', '{1}'
        IF NOT EXISTS (SELECT value FROM
fn_listextendedproperty('UpdatedBy', default, default, default, default,
default, default))
        EXEC sp_addextendedproperty 'UpdatedBy', '{2}'
        ELSE
        EXEC sp_updateextendedproperty 'UpdatedBy', '{2}'
 </Script>
</Scripts>
</Provider>
```

Código 4 – Configuração por defeito do *SQLInstaller* em *SQL Server*

Em cada tipo de base de dados existem 5 diferentes tipos de configurações:

- *Exists* – *script* que será executado com o intuito de saber se a base de dados existe.
- *Drop* – *script* que apaga a base de dados.
- *Create* – *script* que cria a base de dados.
- *GetVersion* – *script* usado para saber a versão atual da base de dados.
- *SetVersion* – *script* usado para atualizar a versão da base de dados.

Nos Código 3 e Código 4, é possível observar que existem 3 campos que serão substituídos pelo *SQLInstaller* quando estiver a ser executado. Esses campos são:

- {0} – nome da base de dados.
- {1} – versão a atualizar.

- {2} – utilizador a correr a programa.

5.2.2 Modificações ao *SQLInstaller*

De modo a ajustar a utilização do *SQLInstaller* aos projeto acuais e futuros da empresa, foi necessário fazer pequenas alterações ao programa. Os subcapítulos seguintes descrevem essas alterações.

5.2.2.1 Utilização excessiva de memória

Quando começamos a utilizar o *SQLInstaller* para realizar alguns testes, reparamos que o processo utilizava memória RAM em demasiado. Após uma análise cuidada do código fonte, foi descoberta a origem do problema.

Na versão original, Código 5, o *SQLInstaller* carrega o ficheiro todo para memória. Depois, tenta encontrar o caracter "/" para separar os *scripts* e guarda cada bloco de *script* em uma lista. Posteriormente, os blocos de *scripts* da lista são executados.

```
List<string> scripts = new List<string>();
List<string> lines = new List<string>();

// Iterate through script line-by-line looking for special terminator at
EOL (/). If
// this is found, then add all prior lines as a single command.
foreach (string line in script.Split(new char[] { Constants.CarriageReturn,
Constants.NewLine }, StringSplitOptions.RemoveEmptyEntries))
{
 if (line.Trim().EndsWith(Constants.ForwardSlash.ToString(),
StringComparison.OrdinalIgnoreCase))
 {
        lines.Add(line.TrimEnd(null).TrimEnd(new char[]
{ Constants.ForwardSlash }));
        scripts.Add(string.Join(Constants.NewLine.ToString(),
lines.ToArray()));
        lines.Clear();
 }
 else
 {
        lines.Add(line);
 }
}

// Any remaining lines will be joined together then re-split on semi-colon
if (lines.Count > 0)
{
 scripts.AddRange(string.Join(Constants.NewLine.ToString(),
lines.ToArray()).Split(new char[] { Constants.SplitChar },
StringSplitOptions.RemoveEmptyEntries));
}

foreach (string sqlLine in scripts)
{
 if (sqlLine.Trim().Length > 0)
```

```
{
        cmd.CommandText = sqlLine;
        cmd.ExecuteNonQuery();
}
```

Código 5 – Código original do processamento de um ficheiro pelo *SQLInstaller*

Para corrigir o problema da utilização em demasiado da memória RAM, foi usada a classe *.NET StringBuilder* e também foi alterada a logica para executar o *script* à medida que é processado sem utilizar uma lista para guardar temporariamente os *scripts*. O Código 6 contém o código com a alteração efetuada.

```
public override void Execute(string script, bool changeDatabase){
 using (DbConnection connection =
this.DbProviderFactory.CreateConnection()){
        DbCommand cmd = SetupConnection(connection, changeDatabase);
        StringBuilder scriptToExecute = new StringBuilder();

        foreach (string line in script.Split(new char[]
{ Constants.CarriageReturn, Constants.NewLine },
StringSplitOptions.RemoveEmptyEntries)){
                ProcessLine(line, cmd, scriptToExecute,
Constants.ForwardSlash.ToString());
        }

        ProcessRemainScript(cmd, scriptToExecute.ToString());
 }
}

private void ProcessRemainScript(DbCommand cmd, string script){
 List<string> remainScripts = new List<string>();
 if (!string.IsNullOrWhiteSpace(script)){
        foreach (var item in script.Split(new char[]
{ Constants.SplitChar }, StringSplitOptions.RemoveEmptyEntries)){
                if (!string.IsNullOrWhiteSpace(item) && item !=
System.Environment.NewLine){
                        remainScripts.Add(item);
                }
        }
 }

 foreach (string scriptToExecute in remainScripts){
        if (script.Length > 0){
                cmd.CommandText = scriptToExecute;
                cmd.ExecuteNonQuery();
        }
 }
```

Código 6 – Novo código de processamento de um ficheiro pelo *SQLInstaller*

5.2.2.2 Personalização das extensões do SQLInstaller

As extensões do *SQLInstaller* também foram modificadas. De modo a utilizar o mesmo ficheiro com extensões em todos os projetos e em bases de dados *Oracle* e *SQL Server*, foram definidas 16 extensões:

1. *PreInstall* – contém os pré-requisitos.
2. *DBLink* – contém as ligações entre base de dados.
3. *Synonym* – contém os sinónimos.
4. *Sequence* – contém as sequências.
5. *Type* – contém os tipos.
6. *Table* – contém as tabelas.
7. *Function* – contém as funções.
8. *Procedure* – contém os procedimentos.
9. *View* – contém as vistas.
10. *PackageSpec* – contém a definição de pacotes.
11. *PackageBody* – contém o corpo dos pacotes.
12. *Trigger* – contém os *triggers*.
13. *Job* – contém os processos que correm na base de dados.
14. *PostInstall* – contém os dados mestres.
15. *Constraint* – contém os índices, chaves primárias e chaves estrangeiras.
16. *CleanUp* – contém os *scripts* que efetuam a limpeza da base de dados.

A Figura 25 exibe o conteúdo do ficheiro de configuração com as 16 extensões definidas.

```
<Parameters Options="Verbose" ScriptExtension="sql" IsProtected="true"
    InstallPath="Install" UpgradePath="Upgrade" NoPrompt="true" >
    <Database>TESTE_LOCAL</Database>
    <Provider Name="SqlServer" />
    <ConnectionString>Integrated Security=SSPI;Initial Catalog=TESTE_LOCAL;Data Source=.;</ConnectionString>
    <FileTypes>
        <FileType Name="PreInstall"  HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="DBLink"      HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Synonym"     HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Sequence"    HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Type"        HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Table"       HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Function"    HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Procedure"   HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="View"        HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="PackageSpec" HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="PackageBody" HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Trigger"     HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Job"         HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="PostInstall" HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="Constraint"  HaltOnError="true" IsGlobal="true" IsDisable="false"/>
        <FileType Name="CleanUp"     HaltOnError="true" IsGlobal="true" IsDisable="false"/>
    </FileTypes>
</Parameters>
```

Figura 25 – Conteúdo do ficheiro de configuração

5.2.2.3 Apagar e criar a base de dados

O ficheiro *ProviderFactory.xml* também foi modificado. No ficheiro original era apagado o utilizador e criado um novo. Por segurança, os programadores não tem permissões para criar nem apagar utilizadores na base de dados. A solução encontrada foi remover todos os objetos da base de dados. A implementação do novo *script* para remover os objetos foi efetuada em *Oracle* (ver Código 7) e *SQL Server* (ver Código 8).

```
DECLARE
PROCEDURE DROP_OBJECT(P_OBJECT_NAME IN VARCHAR2,P_OBJECT_TYPE IN VARCHAR2)
AS
BEGIN
 BEGIN
        IF P_OBJECT_TYPE = 'TABLE' THEN
               EXECUTE IMMEDIATE 'DROP ' || P_OBJECT_TYPE || ' "' ||
P_OBJECT_NAME || '" CASCADE CONSTRAINTS PURGE';
        ELSIF P_OBJECT_TYPE = 'TYPE' THEN
               EXECUTE IMMEDIATE 'DROP ' || P_OBJECT_TYPE || ' "' ||
P_OBJECT_NAME || '" FORCE';
        ELSIF P_OBJECT_TYPE = 'JOB' THEN
               EXECUTE IMMEDIATE ('BEGIN
DBMS_SCHEDULER.DROP_JOB('''||P_OBJECT_NAME||'''); END;');
        ELSE
               EXECUTE IMMEDIATE 'DROP ' || P_OBJECT_TYPE || ' "' ||
P_OBJECT_NAME || '"';
        END IF;
 EXCEPTION
 WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('FAILED: DROP ' || P_OBJECT_TYPE || ' "' ||
P_OBJECT_NAME || '"');
 END;
END;
```

```
PROCEDURE DROP_DATABASE AS
 CURSOR OBJECTS_DROP IS
 SELECT OBJECT_NAME, OBJECT_TYPE
 FROM   USER_OBJECTS
 WHERE  OBJECT_TYPE IN ('TABLE', 'VIEW', 'PACKAGE','PROCEDURE', 'FUNCTION',
'SEQUENCE','DATABASE LINK','SYNONYM','JOB','TYPE');
BEGIN
 FOR CUR_REC IN OBJECTS_DROPLOOP
       DROP_OBJECT(CUR_REC.OBJECT_NAME,CUR_REC.OBJECT_TYPE);
 END LOOP;
END;

BEGIN
 DROP_DATABASE;
 DROP_DATABASE;
 EXECUTE IMMEDIATE 'PURGE RECYCLEBIN';
END;
/
```

Código 7 – Novo *script* para remover objetos em *Oracle*

```
DECLARE @NAME VARCHAR(255)
DECLARE @TYPE VARCHAR(10)
DECLARE @PREFIX VARCHAR(255)
DECLARE @DROP_ORDER INT
DECLARE @SQL NVARCHAR(MAX) = N''

--DELETE ALL CONSTRAITS, FKs, Checks and Unique keys
SELECT
@SQL += N'ALTER TABLE ' +
QUOTENAME(OBJECT_SCHEMA_NAME(O.PARENT_OBJECT_ID))+ '.' +
QUOTENAME(OBJECT_NAME(O.PARENT_OBJECT_ID)) +
' DROP CONSTRAINT ' + QUOTENAME(O.NAME) + ';'
FROM SYS.OBJECTS O
INNER JOIN SYS.SCHEMAS S ON S.SCHEMA_ID = O.SCHEMA_ID
WHERE
 O.TYPE IN ('C', 'UQ', 'F')
 AND S.NAME IN ('dbo')

EXEC SP_EXECUTESQL @SQL

--DELETE ALL OBJECTS
DECLARE curs CURSOR FOR
SELECT
 O.[NAME],
 O.TYPE,
 CASE
       WHEN O.TYPE IN ('P') THEN 1
       WHEN O.TYPE IN ('FN','IF','TF') THEN 2
       WHEN O.TYPE IN ('TR') THEN 3
       WHEN O.TYPE IN ('V') THEN 4
       WHEN O.TYPE IN ('SN') THEN 5
       WHEN O.TYPE IN ('U') THEN 6
 ELSE 0
END [DROP_ORDER]
FROM SYS.OBJECTS O
INNER JOIN SYS.SCHEMAS S ON S.SCHEMA_ID = O.SCHEMA_ID
WHERE O.TYPE IN ('U', 'P', 'FN', 'IF', 'TF', 'V', 'TR','SN')
```

```
 AND S.NAME IN ('dbo')
 ORDER BY [DROP_ORDER]

OPEN CURS
 FETCH NEXT FROM CURS INTO @NAME, @TYPE,@DROP_ORDER
 WHILE @@FETCH_STATUS = 0
 BEGIN
       SET @PREFIX =
       CASE @TYPE
             WHEN 'U' THEN 'DROP TABLE'
             WHEN 'P' THEN 'DROP PROCEDURE'
             WHEN 'FN' THEN 'DROP FUNCTION'
             WHEN 'IF' THEN 'DROP FUNCTION'
             WHEN 'TF' THEN 'DROP FUNCTION'
             WHEN 'V' THEN 'DROP VIEW'
             WHEN 'TR' THEN 'DROP TRIGGER'
             WHEN 'SN' THEN 'DROP SYNONYM'
 END

 SET @SQL = @PREFIX + ' ' + @NAME
 PRINT @SQL
 EXEC(@SQL)
 FETCH NEXT FROM CURS INTO @NAME, @TYPE,@DROP_ORDER
 END
CLOSE CURS
DEALLOCATE CURS

--DROP ALL USER TYPES
DECLARE CUR_TYPES CURSOR FOR
SELECT T.NAME
FROM SYS.TYPES T
INNER JOIN SYS.SCHEMAS S  ON S.SCHEMA_ID = T.SCHEMA_ID
WHERE S.NAME = 'dbo'

OPEN CUR_TYPES
 FETCH NEXT FROM CUR_TYPES INTO @NAME
 WHILE @@FETCH_STATUS = 0
 BEGIN
       SET @SQL = 'DROP TYPE ' + @NAME
       EXEC(@SQL)
       FETCH NEXT FROM CUR_TYPES INTO @NAME
 END
CLOSE CUR_TYPES
DEALLOCATE CUR_TYPES
```

Código 8 – Novo *script* para remover objeto em *SQL Server*

Com estas modificações, foi possível utilizar o *SQLInstaller* nos projetos atuais e nos novos produtos a serem desenvolvidos pela empresa, sem necessidade de atribuir permissões adicionais aos programadores.

5.3 Extrair *scripts* para reconstruir a base de dados

A utilização do *SQLInstaller* requer que os *scripts* estejam em diferentes pastas num determinado formato. No entanto, nos novos projetos, apenas é necessário criar a estrutura

de pastas e os programadores vão adicionando novos *scripts* durante a construção da aplicação. Para os projetos mais antigos, que já tem a base de dados criada, é complicado e demoroso extrair os *scripts* em um determinado formato a fim de reconstruir a base de dados a partir do *SQLInstaller*.

De modo a extrair os *scripts* num determinado formato para reconstruir a base de dados a partir do *SQLInstaller,* foi desenvolvido um novo programa. O nome dado ao programa foi *SQLExtract*.

5.3.1 Configuração

Servindo de inspiração do *SQLInstaller*, este programa também usa um ficheiro semelhante de configurações para extrair os *scripts*. A Figura 26 mostra o conteúdo e formato do ficheiro de configuração.

```
<Parameters Options="GenerateDDLScripts ExtractDDLScripts ExtractDependencies" ScriptExtension="sql" ExtractPath="TP\Current\Install">
    <Provider>SqlServer</Provider>
    <Database>LOCAL_TESTE</Database>
    <ConnectionString>Integrated Security=SSPI;Initial Catalog=LOCAL_TESTE;Data Source=.;</ConnectionString>
    <Folders>
        <Folder Name="01_PreInstall"    />
        <Folder Name="02_DBLink"        SQLInstallerExtension="DBLink"      ObjectType="DB_LINK"        SingleFile="true"   FileName="ALL_DBLINKS"/>
        <Folder Name="03_Synonym"       SQLInstallerExtension="Synonym"     ObjectType="SYNONYM"        SingleFile="true"   FileName="ALL_SYNONYMS"/>
        <Folder Name="04_Sequence"      SQLInstallerExtension="Sequence"    ObjectType="SEQUENCE"       SingleFile="true"   FileName="ALL_SEQUENCES"/>
        <Folder Name="05_Type"          SQLInstallerExtension="Type"        ObjectType="TYPE"           SingleFile="true"   FileName="ALL_TYPES"/>
        <Folder Name="06_Table"         SQLInstallerExtension="Table"       ObjectType="TABLE"          SingleFile="false"/>
        <Folder Name="07_Function"      SQLInstallerExtension="Function"    ObjectType="FUNCTION"       SingleFile="false"/>
        <Folder Name="08_Procedure"     SQLInstallerExtension="Procedure"   ObjectType="PROCEDURE"      SingleFile="false"/>
        <Folder Name="09_View"          SQLInstallerExtension="View"        ObjectType="VIEW"           SingleFile="false"/>
        <Folder Name="10_PackageSpec"   SQLInstallerExtension="PackageSpec" ObjectType="PACKAGE_SPEC"   SingleFile="false"/>
        <Folder Name="11_PackageBody"   SQLInstallerExtension="PackageBody" ObjectType="PACKAGE_BODY"   SingleFile="false"/>
        <Folder Name="12_Trigger"       SQLInstallerExtension="Trigger"     ObjectType="TRIGGER"        SingleFile="false"/>
        <Folder Name="13_Jobs"          SQLInstallerExtension="Job"         ObjectType="JOB"            SingleFile="false"/>
        <Folder Name="14_PostInstall"   />
        <Folder Name="15_Constraint"    SQLInstallerExtension="Constraint"  ObjectType="INDEX"          SingleFile="true"   FileName="01_INDEXES"/>
        <Folder Name="15_Constraint"    SQLInstallerExtension="Constraint"  ObjectType="CONSTRAINT"     SingleFile="true"   FileName="02_CONSTRAINTS"/>
        <Folder Name="15_Constraint"    SQLInstallerExtension="Constraint"  ObjectType="REF_CONSTRAINT" SingleFile="true"   FileName="03_FOREIGN_KEYS"/>
        <Folder Name="16_CleanUp"       />
    </Folders>
</Parameters>
```

Figura 26 – Conteúdo do ficheiro de configuração do *SQLExtract*

A Tabela 2 contém a descrição de cada parâmetro.

Tabela 2 – Parâmetros do *SQLExtract*

Parâmetro	Descrição
Options	Existem 3 opções disponíveis: • *GenerateDDLScripts* – gerar os *scripts*. Como a geração de *script* é demorada, foi criada uma opção separada. • *ExtractDDLScripts* – criar os ficheiros em disco com os *scripts*. • *ExtractDependencies* – extrair as dependências de cada objeto. Gera um ficheiro CSV com as dependências.

ScriptExtension	Extensão a usar no ficheiro de *script* gerado.
ExtractPath	Caminho relativo onde os ficheiros serão criados.
Provider	O tipo de base de dados. Existem 2 disponíveis: • *SQL Server*. • *Oracle*.
Database	Nome da base de dados.
ConnectionString	Conexão da base de dados.
Folders	Pastas a criar onde serão guardados os ficheiros com *scripts*. Em cada pasta existem 5 parâmetros: • *Name* – nome da pasta. • *SQLInstaller*Extension – extensão do ficheiro do *SQLInstaller*. • *ObjectType* – tipo de objeto a extrair. • *SingleFile* – indica se é para criar um ficheiro com todos os objeto do mesmo tipo ou um ficheiro por objeto. • *FileName* – nome do ficheiro a usar. Apenas é usado quando é gerado um ficheiro para vários objetos.

5.3.2 Geração de scripts

Não existe uma forma genérica a fim de gerar *scripts* para construir um objeto em *Oracle* e *SQL Server*. Por isso, o programa foi desenhado de forma a ter uma solução diferente em cada tipo de base de dados mas com o objetivo de obter o mesmo resultado.

5.3.2.1 Oracle

Em *Oracle*, a geração de *scripts* acontece na base de dados. Foi usado o pacote da *Oracle DBMS_METADATA*. Como a geração de *scripts* é um processo demorado, estes são guardados à medida que vão sendo executados para uma tabela temporária. Após os *scripts* serem todos gerados, são lidos da base de dados pelo programa e guardados em ficheiros.

A função criada para gerar *scripts* em *Oracle*, Código 9, recebe dois parâmetros. O primeiro parâmetro *P_OBJECT_TYPE* é o tipo de objeto (tabela, procedimento, vista, entre outros). O segundo parâmetro *P_OBJECT_VALUE* é o nome do objeto.

```
FUNCTION GET_DDL_WITHOUT_SCHEMA(P_OBJECT_TYPE IN VARCHAR,P_OBJECT_VALUE IN
VARCHAR)
 RETURN CLOB IS
L_H NUMBER; --HANDLE RETURNED BY OPEN
L_TH NUMBER; -- HANDLE RETURNED BY ADD_TRANSFORM
L_DOC CLOB;
BEGIN
 -- SPECIFY THE OBJECT TYPE.
 L_H := DBMS_METADATA.OPEN(P_OBJECT_TYPE);
 -- USE FILTERS TO SPECIFY THE PARTICULAR OBJECT DESIRED.
 DBMS_METADATA.SET_FILTER(L_H,'SCHEMA',USER);
 DBMS_METADATA.SET_FILTER(L_H,'NAME',P_OBJECT_VALUE);
 IF P_OBJECT_TYPE <> JOB_DDL_TYPE THEN
 -- REQUEST THAT THE SCHEMA NAME BE MODIFIED.
 L_TH := DBMS_METADATA.ADD_TRANSFORM(L_H,'MODIFY');
 DBMS_METADATA.SET_REMAP_PARAM(L_TH,'REMAP_SCHEMA',USER,NULL);
 END IF;
 -- REQUEST THAT THE METADATA BE TRANSFORMED INTO CREATION DDL.
 L_TH := DBMS_METADATA.ADD_TRANSFORM(L_H,'DDL');
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'DEFAULT');
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'SQLTERMINATOR', TRUE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'PRETTY', TRUE);
 IF P_OBJECT_TYPE IN ('INDEX','CONSTRAINT') THEN
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'SEGMENT_ATTRIBUTES', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'STORAGE', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'TABLESPACE', FALSE);
 END IF;
 IF P_OBJECT_TYPE IN (TABLE_OBJECT_TYPE) THEN
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'SEGMENT_ATTRIBUTES', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'STORAGE', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'TABLESPACE', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'CONSTRAINTS', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'REF_CONSTRAINTS', FALSE);
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'SIZE_BYTE_KEYWORD', FALSE);
 END IF;
 IF P_OBJECT_TYPE IN (VIEW_OBJECT_TYPE) THEN
 DBMS_METADATA.SET_TRANSFORM_PARAM(L_TH, 'FORCE', TRUE);
 END IF;
 -- FETCH THE OBJECT.
 L_DOC := DBMS_METADATA.FETCH_CLOB(L_H);
 --FIX SCHEMA NAME FOR SYNONYM
 IF P_OBJECT_TYPE IN (SYNONYM_OBJECT_TYPE) THEN
 L_DOC := REPLACE (L_DOC, '"."', '"');
 END IF;
 -- RELEASE RESOURCES.
 DBMS_METADATA.CLOSE(L_H);
 RETURN L_DOC;
END;
```

Código 9 – Geração de *scripts* em *Oracle*

5.3.2.2 SQL Server

Em *SQL Server*, para gerar os *scripts,* foi usado o *namespace Microsoft.SqlServer.Management.Sdk.Sfc*. Este *namespace* fornece um conjunto de opções para gerar *scripts*. De modo a ter acesso a este *namespace,* é necessário ter o *Microsoft Management Studio* instalado. O Código 10 contem um exemplo de como usar o *namespace.*

```
Server Server = new Server(new ServerConnection(new
SqlConnection(this.Parameters.ConnectionString)));
Database Database = Server.Databases[this.Parameters.Database];
Scripter Scripter = new Scripter(this.Server);

Scripter.Options.SchemaQualify = true;
Scripter.Options.ScriptDrops = false;
Scripter.Options.SchemaQualify = false;
Scripter.Options.WithDependencies = false;
Scripter.Options.AllowSystemObjects = false;
Scripter.Options.IncludeIfNotExists = false;
Scripter.Options.NoCollation = true;
Scripter.Options.ScriptBatchTerminator = true;
Scripter.Options.Permissions = false;
Scripter.Options.Indexes = false;
Scripter.Options.Triggers = false;
Scripter.Options.Statistics = false;
Scripter.Options.ExtendedProperties = true;
Scripter.Options.DriDefaults = true;

foreach (StoredProcedure item in Database.StoredProcedures)
{
 if (item.IsSystemObject == false)
 {

        string script = this.Scripter.Script(new Urn[]
{ item.Urn }).ToStringBuilder().ToString();
 }
}
```

Código 10 – Geração de *scripts* em *SQL Server*

Após serem gerados os *scripts*, quer sejam em *SQL Server* ou *Oracle*, é usada a configuração que é lida do ficheiro de configuração para guardar os *scripts* em ficheiros.

5.4 Gerir dados mestres

Quando se cria uma base de dados, é necessário introduzir alguns dados mestres de modo a que a aplicação que usa a base de dados possa funcionar. Para que a introdução de dados seja rápida e eficiente, deve ser efetuada automaticamente durante a construção da base de dados.

Com a finalidade de serem inseridos dados mestres durante a construção da aplicação, usando o *SQLInstaller*, é necessário que estes estejam em *scripts*. Ao ter os dados mestres em

scripts, não é fácil fazer a sua manipulação. Para solucionar este problema, foi decidido criar um *add-in* no *Excel*. Este *add-in* lê os dados que estão presentes na folha de cálculo e cria *scripts* de inserção de dados mestres. O nome dado a este *add-in* foi *ExcelToSQL*. A Figura 27 mostra como o *add-in* aparece no *Excel*.

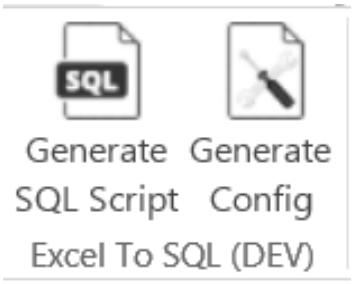

Figura 27 – *Add-in* no *Excel*

Para fazer o mapeamento entre folha de cálculo e uma tabela na base de dados, foram definidas algumas regras no ficheiro que contém os dados mestres. De modo a permitir que um ficheiro *Excel* possa fazer a inserção em múltiplas tabelas, foi definido que cada folha de cálculo corresponde a uma tabela e o nome da folha de cálculo corresponde ao nome da tabela. Dentro de cada folha de cálculo, a primeira linha contém os nomes das colunas. A Figura 28 contem um exemplo desta configuração.

	A	B	C	D	E
1	USER_CODE	FIRST_NAME	LAST_NAME	BIRTH_DATE	
2	1	Shigeaki	Asakura	20/04/1986	
3	2	Toru	Toda	12/07/1979	
4	3	Joseph	Kantarjian	05/01/1989	
5	4	Guy	Dulas	10/07/1986	
6	5	John	Tulett	12/07/1989	
7	6	Alain	Donval	15/01/1988	
8					
9					
10					
11					

ScriptParameters | TUSERS

Figura 28 – Folha de cálculo

Como a geração do *script* tem que suportar as bases de dados em *Oracle* e *SQL Server*, para definir o tipo de dados, não foi usada a formatação das colunas do *Excel*. Com o intuito de definir o tipo de dados das colunas, existe uma folha de cálculo de configurações que é gerada selecionando a opção *Generate Config* no *add-in*.

O nome da folha de cálculo com configurações é *ScriptParameters*. Esta folha contém 3 parâmetros genéricos:

- *LogFileNamePath* – localização do ficheiro de monitorização. Por defeito, é a pasta onde está o ficheiro *Excel*.

- *Provider* – tipo da base de dados alvo. Atualmente, apenas suporta *Oracle* e *SQL Server*.

- *BatchSize* – número máximo de linhas a inserir consecutivamente. Por defeito, o valor é 1000.

O único parâmetro relativo à configuração da folha de cálculo é o tipo de *script*. Este parâmetro indica se é para gerar um *script* de inserção ou atualização de dados. O nome do parâmetro é *Table::{0}::ScriptType* onde *{0}* corresponde ao nome da tabela e tem dois valores possíveis:

- *Insert* – significa que o *script* gerado é de inserção de dados nas tabelas.

- *Update* – significa que o *script* gerado é de atualização de dados nas tabelas.

A configuração de cada coluna tem o formato *Table::{0}::Column::{1}::{2}*, onde *{0}* corresponde ao nome da tabela, {1} ao nome da coluna e {2} à chave de configuração. Existem 3 chaves de configuração:

- *SequenceName* – nome da sequência a usar para inserção de dados nesta coluna.

- *DataType* – tipo de dados da coluna. Existem 7 valores disponíveis: *General, Number, Text, Date, Timestamp, CLOB* e *NVarchar*.

- *IsPrimaryKey* – indica se a coluna faz parte da chave primária. Os valores possíveis são verdadeiro ou falso.

A Tabela 3 mostra um exemplo da folha com configurações.

Tabela 3 – Folha de configurações

LogFileNamePath	TUSERS.txt
Provider	*Oracle*
BatchSize	1000
Table::TUSERS::*Script*Type	Insert
Table::TUSERS::Column::USER_CODE::SequenceName	
Table::TUSERS::Column::USER_CODE::DataType	Number
Table::TUSERS::Column::USER_CODE::IsPrimaryKey	VERDADEIRO
Table::TUSERS::Column::FIRST_NAME::SequenceName	
Table::TUSERS::Column::FIRST_NAME::DataType	Text
Table::TUSERS::Column::FIRST_NAME::IsPrimaryKey	FALSO
Table::TUSERS::Column::LAST_NAME::SequenceName	

Table::TUSERS::Column::LAST_NAME::DataType	Text
Table::TUSERS::Column::LAST_NAME::IsPrimaryKey	FALSO
Table::TUSERS::Column::BIRTH_DATE::SequenceName	
Table::TUSERS::Column::BIRTH_DATE::DataType	Date
Table::TUSERS::Column::BIRTH_DATE::IsPrimaryKey	FALSO

Após configurar as colunas, para gerar o *script,* é apenas necessário selecionar o botão *Generate SQL Script* no *add-in*. O progresso da geração de *script* é mostrado na barra de estados do *Excel*, como ilustra a Figura 29.

Figura 29 – Progresso da geração de *scripts*

Com a configuração presente na Tabela 3, é gerado o *script* ilustrado na Figura 30.

```
DECLARE
    L_NULL CHAR(1) := NULL;
    L_CLOB CLOB;
    L_INT CHAR(84);
BEGIN
    L_INT:= 'INSERT INTO TUSERS (USER_CODE,FIRST_NAME,LAST_NAME,BIRTH_DATE) VALUES(:1,:2,:3,:4)';
    EXECUTE IMMEDIATE L_INT USING 1,'Shigeaki','Asakura',TO_DATE('1986-04-20','YYYY-MM-DD');
    EXECUTE IMMEDIATE L_INT USING 2,'Toru','Toda',TO_DATE('1979-07-12','YYYY-MM-DD');
    EXECUTE IMMEDIATE L_INT USING 3,'Joseph','Kantarjian',TO_DATE('1989-01-05','YYYY-MM-DD');
    EXECUTE IMMEDIATE L_INT USING 4,'Guy','Dulas',TO_DATE('1986-07-10','YYYY-MM-DD');
    EXECUTE IMMEDIATE L_INT USING 5,'John','Tulett',TO_DATE('1989-07-12','YYYY-MM-DD');
    EXECUTE IMMEDIATE L_INT USING 6,'Alain','Donval',TO_DATE('1988-01-15','YYYY-MM-DD');
END;
```

Figura 30 – *Script* gerado pelo a*dd-in* para *Oracle*

Para *SQL Server* o *script* gerado seria o ilustrado na Figura 31.

```
GO
SET NUMERIC_ROUNDABORT OFF
GO
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT, QUOTED_IDENTIFIER, ANSI_NULLS, NOCOUNT ON
GO
SET XACT_ABORT ON
GO
BEGIN TRANSACTION

INSERT INTO TUSERS (USER_CODE,FIRST_NAME,LAST_NAME,BIRTH_DATE) VALUES
(1,'Shigeaki','Asakura',SELECT CONVERT (DATE,1986-04-20)),
(2,'Toru','Toda',SELECT CONVERT (DATE,1979-07-12)),
(3,'Joseph','Kantarjian',SELECT CONVERT (DATE,1989-01-05)),
(4,'Guy','Dulas',SELECT CONVERT (DATE,1986-07-10)),
(5,'John','Tulett',SELECT CONVERT (DATE,1989-07-12)),
(6,'Alain','Donval',SELECT CONVERT (DATE,1988-01-15))

IF @@TRANCOUNT > 0
    COMMIT TRANSACTION
GO
```

Figura 31 – *Script* gerado pelo a*dd-in* para *SQL Server*

Aproveitando o core da aplicação, também foi criada uma aplicação consola e uma *Custom Tool*[10] para o *Visual Studio* com o objetivo de gerar os *scripts* sem abrir o ficheiro *Excel*.

5.5 Integração contínua com o TFS

No novo produto, na construção da aplicação (*website* em *.NET* e base de dados em *SQL Server*), foi definido que seria usado o TFS. O TFS contém diversas funcionalidades para além de servir de repositório de código. Uma das funcionalidades é a possibilidade de execução de *scripts PowerShell*.

De modo a implementar a integração contínua neste produto, foi criado um simples *script PowerShell* que copia o código para o servidor de integração contínua e efetua a construção da base de dados executando o *SQLInstaller*. A Figura 32 ilustra este processo.

[10] http://www.codeproject.com/Articles/31257/Custom-Tools-Explained

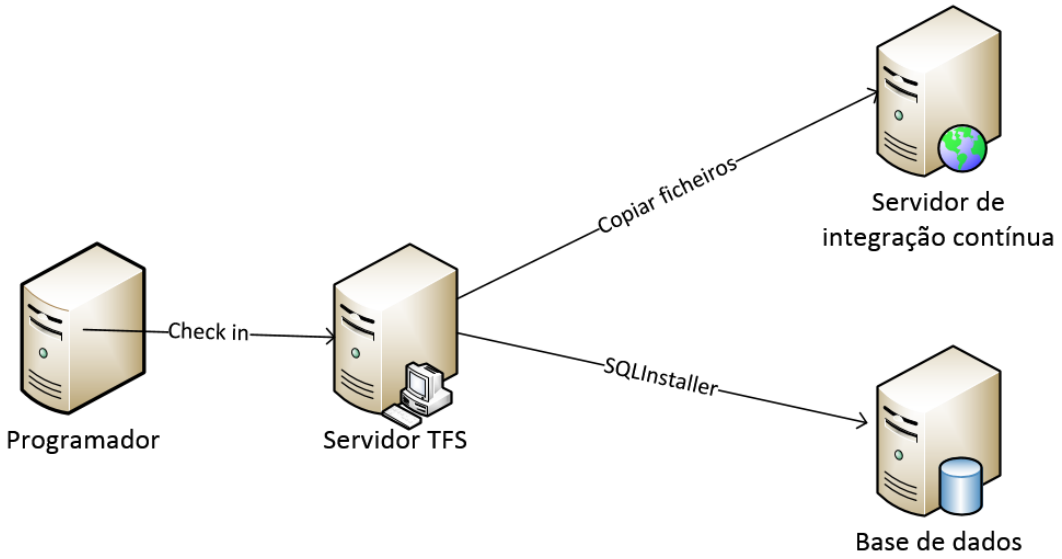

Figura 32 – TFS e integração contínua.

Como o *SQLInstaller* é apenas um executável, para não o instalar no servidor de TFS, foi decidido criar uma pasta nos projetos que o iam usar e copiar para lá o *SQLInstaller* e as configurações que foram efetuadas pela empresa referidas no capítulo 5.2.2.

6 Conclusão

Com a realização desta dissertação, foi possível conhecer aprofundadamente a prática de desenvolvimento de *software*, a integração contínua.

Apesar de existir há algum tempo e ter evoluído de modo a se adaptar às necessidades atuais, a integração contínua ainda não é uma prática muito usada no desenvolvimento de *software*. No entanto, com o desenvolvimento de *software* a evoluir, iremos assistir cada vez mais ao envolvimento desta prática em diferentes projetos.

A inclusão desta prática no desenvolvimento de *software* depende muito de projeto para projeto. Não existe uma solução universal com a finalidade de resolver todos os problemas, visto que, não existem dois projetos iguais e existem imensas tecnologias com o intuito de desenvolver o mesmo tipo de produto. Portanto, haverá sempre necessidades específicas que requerem desenvolvimentos próprios.

Na realização deste trabalho, foi possível observar as dificuldades e soluções encontradas, a fim de responder às exigências da empresa, de modo a incluir a integração contínua no desenvolvimento de um novo produto a ser lançado pela mesma. A solução desenvolvida atingiu todos os objetivos inicialmente definidos.

Apesar de atingir todos os objetivos imediatos, faltou fazer uma análise fora do âmbito do produto em que se enquadrou a criação das ferramentas. Por isso, devem existir problemas que não foram abordados e que irão ser descobertos quando se sugerir o uso desta prática em novos produtos a serem desenvolvidos pela empresa.

Muitas das ferramentas desenvolvidas neste projeto serão certamente reutilizadas em novos projetos. Mesmo que nesses novos projetos não seja usada a integração contínua, é possível mesmo assim usar as ferramentas e isso é uma mais-valia para a empresa.

Para além dos problemas práticos encontrados, também há a mudança de mentalidades a ter em conta e isso depende dos programadores e gestores de projeto. Por exemplo, a inclusão

de testes no desenvolvimento de *software* devia estar sempre presente nas tarefas de cada programador, mesmo que não seja uma exigência obrigatória nem que esteja explicitamente definido como tarefa no plano do projeto. Por outro lado, os gestores de projeto devem ter consciência que a inclusão de testes acrescenta tempo nos desenvolvimentos. Porém, o maior retorno não é imediato mas sim no futuro, quando o produto estiver em produção e for necessário fazer alterações e garantir que não se quebrou nada.

A realização desta dissertação tinha como objetivo demonstrar a importância da integração contínua no desenvolvimento de *software*. Após toda a pesquisa efetuada e a utilização num projeto, concluiu-se que há vantagens em usar esta prática. No entanto, se não for bem aplicada, facilmente se tornará em um problema e não em uma solução.

No primeiro projeto onde será incorporado o uso da integração contínua no desenvolvimento de *software*, haverá um custo suplementar a encontrar soluções e a alterar mentalidades, contudo, o retorno futuro compensará.

6.1 Trabalho futuro

Apesar da construção das diversas ferramentas já estarem terminadas e estarem em funcionamento no projeto que está atualmente a ser desenvolvido pela empresa, pretende-se reutilizar e adaptar essas mesmas ferramentas em futuros projetos a serem desenvolvidos.

Quanto ao trabalho a desenvolver no futuro, poderá passar por efetuar uma melhoria às ferramentas desenvolvidas. A quando da conclusão do produto a ser desenvolvido pela empresa, onde está a ser usada a integração contínua, a equipa terá mais experiência no uso da integração contínua e nas ferramentas desenvolvidas. Nessa altura, serão certamente sugeridas alterações de modo a melhorar as ferramentas ou até criar novas ferramentas a fim de responder a possíveis problemas que possam vir a surgir.

A curto prazo, está planeado a criação de um conjunto de funções e procedimentos na base de dados com o objetivo de tornar a execução dos *scripts* repetíveis e sem falhas. Um *script* que adiciona uma coluna em uma tabela em que essa mesma coluna já existe, deteta que a coluna existe e não dá erro.

No futuro projeto, em dispositivos móveis, está planeado estudar também alternativas para tornar o processo de disponibilização de versões da aplicação mais eficiente e autónomo. Atualmente, o programador necessita de obter o código fonte para a sua máquina e compilar a aplicação para posteriormente a disponibilizar para testes. O desafio neste projeto será configurar o TFS para conseguir compilar aplicações *Android* e *iOS* e disponibilizá-las posteriormente para testes.

Referências

[Alistair Cockburn e Jim Highsmith, 2001]	Agile software development: the people factor, IEEE Computer Society, p. 131–133
[Andrew Bayer, 2011]	Hudson's future, http://jenkins-ci.org/content/hudsons-future [ultimo acesso: setembro 2015]
[Don Wells, 2009a]	The Values of Extreme Programming, http://www.extremeprogramming.org/values.html [ultimo acesso: setembro 2015]
[Don Wells, 2009b]	Introducing Extreme Programming, http://www.extremeprogramming.org/introduction.html [ultimo acesso: setembro 2015]
[Hirotaka Takeuchi e Ikujiro Nonaka, 1986]	The New Product Development Game. Harvard Business Review, p. 137–146
[IBM, 2012]	Continuous integration in agile development, http://www.ibm.com/developerworks/rational/library/continuous-integration-agile-development/ [ultimo acesso: agosto 2015]
[Jenkins, 2015]	Jenkins an extensible open source continuous integration server, http://jenkins-ci.org/node [ultimo acesso: setembro 2015]
[Jennifer Stapleton, 1997b]	Dynamic systems development method – The method in practice. Addison Wesley, p. 3
[Jennifer Stapleton, 1997a]	Dynamic systems development method – The method in practice. Addison Wesley
[Jim Haungs, 2001]	Pair programming on the C3 project, http://faculty.salisbury.edu/~xswang/research/papers/serelated/pp/r2118.pdf [ultimo acesso: setembro 2015]
[Jim Highsmith, 2000]	Extreme Programming, An Overview, e-Business Application Delivery
[Juha Koskela, 2003]	Software configuration management in agile methods, http://www.vtt.fi/inf/pdf/publications/2003/P514.pdf [ultimo acesso: setembro 2015]
[Ken Schwaber e Mike Beedle, 2002]	Agile Software Development With Scrum. Upper Saddle River, NJ, Prentice-Hall. 1st
[Ken Schwaber, 1995]	Scrum Development Process, http://navegapolis.net/files/Scrum_Development_Process.pdf [ultimo acesso: setembro 2015]
[Kent Beck entre outros, 2001]	Principles behind the Agile Manifesto, http://www.agilemanifesto.org/iso/ptpt/principles.html [ultimo acesso: setembro 2015]
[Kent Beck, 1999a]	Embracing Change With Extreme Programming. IEEE Computer 32(10), p. 70–77
[Kent Beck, 1999b]	Extreme programming explained: Embrace change. Addison-Wesley. 1st
[Kent Beck, 1999c]	Extreme programming explained: Embrace change. Addison-

	Wesley. 1st, p. 58
[Kent Beck, 1999d]	Extreme programming explained: Embrace change. Addison-Wesley. 1st, p. 54
[Kohsuke Kawaguchi, 2007]	Hudson, https://www.java.net//blog/kohsuke/archive/20070514/Hudson%20J1.pdf [ultimo acesso: setembro 2015]
[Mark Hoogveld, 2012]	The Scrum Overview, https://markhoogveld.wordpress.com/2015/10/03/the-scrum-overview/ [ultimo acesso: setembro 2015]
[Martin Fowler, 2002]	The Agile Manifesto: where it came from and where it may go, http://martinfowler.com/articles/agileStory.html [ultimo acesso: setembro 2015]
[Martin Fowler, 2006]	Continuous Integration, http://www.martinfowler.com/articles/continuousIntegration.html [ultimo acesso: agosto 2015]
[Microsoft, 2013]	Application Lifecycle Management with Visual Studio and Team Foundation Server, https://msdn.microsoft.com/en-us/library/vstudio/fda2bad5(v=vs.120).aspx [ultimo acesso: setembro 2015]
[Northumbria University, 2005]	Dynamic Systems Development Method (DSDM), http://computing.unn.ac.uk/staff/cgjh1/cm602/dsdm 2005.ppt [ultimo acesso: outubro 2015]
[Paul Duvall, 2007a]	Continuous Integration: Improving Software Quality and Reducing Risk. 1st Edition, p. 5
[Paul Duvall, 2007b]	Continuous Integration: Improving Software Quality and Reducing Risk. 1st Edition, p. 48
[Paul Duvall, 2007c]	Continuous Integration: Improving Software Quality and Reducing Risk. 1st Edition, p. 70
[Paul Duvall, 2007d]	Continuous Integration: Improving Software Quality and Reducing Risk. 1st Edition, p. 12
[Rafeeq Rehman e Christopher Paul, 2007]	The Linux Development Platform, Prentice Hall PTR
[Steve Palmer e Mac Felsing, 2002]	A Practical Guide to Feature-Driven Development. Upper Saddle River, NJ, Prentice-Hall
[Susan Duncan, 2011]	The Future of Hudson, https://java.net/projects/hudson/lists/dev/archive/2011-02/message/0 [ultimo acesso: setembro 2015]
[The Next Web, 2013]	Microsoft acquires InRelease, adding continuous deployment to Visual Studio, Team Foundation Server, http://thenextweb.com/microsoft/2013/06/03/microsoft-acquires-inrelease-adding-continuous-deployment-to-visual-studio-team-foundation-server/ [ultimo acesso: setembro 2015]