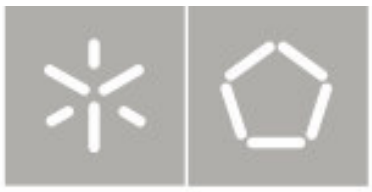
Universidade do Minho
Escola de Engenharia

Nuno Alberto Pereira da Silva

Rejuvenescimento de Aplicações: Uma Experiência com Software de Seguros

Outubro de 2005



Universidade do Minho
Escola de Engenharia

Nuno Alberto Pereira da Silva

Rejuvenescimento de Aplicações: Uma Experiência com Software de Seguros

Tese de Mestrado em Sistemas de informação

Trabalho efectuado sob a orientação do
Professor Doutor Ricardo Jorge de Magalhães Machado
Departamento de Sistemas de Informação

Outubro de 2005

DECLARAÇÃO

Nome: NUNO ALBERTO PEREIRA DA SILVA

Endereço Electrónico: nuno.silva@i2s.pt **Telefone:** 933 610 721

N.° do Bilhete de Identidade: 11043986

Título da Tese de Mestrado:

Rejuvenescimento de Aplicações: Uma Experiência com Software de Seguros

Orientador:

Professor Doutor Ricardo Jorge Silvério de Magalhães Machado

Ano de conclusão: 2005

Designação do Mestrado:

Mestrado em Sistemas de Informação

Universidade do Minho, / /

Assinatura: _____

Abstract

Software development methodologies have been adapting to new organizational demands in order to give effective answers to their needs. However, the effectiveness of such development methodologies can only be achieved through adequate framework and development tools. In the real world, just like financial systems, most information systems are supported by legacy technologies that compromises system evolution. On the other hand, because of complexity and criticism of such systems, it's not possible to rebuild the whole system from scratch. The evolution must be performed step by step, through software refactoring.

Software refactoring appears like the key for software rejuvenate, giving it the capability to integrate new agile development methodologies and emerging technologies.

To develop and evolve robust systems based on legacy software is the big challenge for this dissertation. Having this motivation, we propose to give a little contribute, with a systematic approach to refactoring software based on legacy systems.

To prove the concepts defended in this dissertation, the case study mentioned will refer to an insurance software which is used by most Portuguese life insurance companies and is now in expansion throughout the world.

Resumo

As metodologias de desenvolvimento de software têm vindo a adaptar-se às exigências das organizações no sentido de responderem rapidamente às suas necessidades. No entanto, o recurso a essas metodologias só é possível se o *framework* já estiver preparado com a estrutura e ferramentas adequadas. No mundo real, e nomeadamente nos sistemas financeiros como a banca e seguros, os sistemas de informação estão, na sua maioria, assentes numa estrutura tecnológica legada, que inviabiliza a adopção dessas metodologias e consequentemente compromete a evolutibilidade desses sistemas. Por outro lado, dada a complexidade e criticidade desses sistemas, a migração tecnológica, refazendo tudo de início, é desaconselhada. A evolução deve ser efectuada por etapas e com passos seguros através da refabricação do software.

A refabricação de software surge, então, como factor de rejuvenescimento de aplicações conferindo características que lhes permitam, quer a adopção de metodologias ágeis para o desenvolvimento de software, quer a capacidade de integrar as tecnologias emergentes.

Desenvolver sistemas de informação robustos e evolutíveis com base em software legado é o grande desafio desta dissertação. Com esta base de motivação, pretende-se dar um contributo para sistematização da actividade de refabricação de software em sistemas legados.

Para prova dos conceitos advogados nesta dissertação é estudado um caso real com software de seguros, cuja implementação abrange a quase totalidade das seguradoras em Portugal, encontrando-se em expansão além fronteiras.

Agradecimentos

Quero deixar o meu reconhecimento aos meus pais que, desde sempre, me acompanharam e apoiaram, Felismina e Alberto, e em especial à minha *little syster* Dora.

Ao meu orientador, professor Ricardo J. Machado, pelas sugestões e questões pertinentes e decisivas na assunção do rumo acertado para esta dissertação.

Aos meus colegas da I2S, que contribuíram com as suas vastas experiências e significaram um valor acrescentado neste trabalho, Paulo Bastos, Jorge Miranda e Rui Monteiro. Não esquecendo o eng.º Luís Paupério, que esteve sempre disponível contribuindo com os seus sábios conhecimentos para a valorização desta dissertação.

Aos meus amigos por me terem suportado e ajudado nesta caminhada, Marcela, Juliana, Patrícia, Serafim, Adolfo, Lino, Edgar e Nuno Ferreira.

"Até hoje, o foco da literatura vai no sentido de se ensinar a criar grandes soluções arquitecturais de software, no entanto, e cada vez mais, as necessidades de hoje e amanhã vão no sentido de se ser capaz de evoluir de sistemas existentes, ou legados, para grandes soluções."

Joshua Kerievsky

Conteúdo

1. Introdução 1
1.1. Evolutibilidade em Sistemas de Informação 1
1.2. Objectivos Propostos 4
1.3. Metodologia de Investigação 5
1.4. Estrutura da Dissertação 5
2. Desenvolvimento e Evolutibilidade 7
2.1. Introdução 7
2.2. Metodologias de Suporte ao Desenvolvimento 8
2.3. Refabricação de Software 16
2.4. Testes de Conformidade 24
2.5. Conclusão 27
3. Rejuvenescimento de Aplicações 29
3.1. Introdução 29
3.2. Metodologias Propostas 34
3.3. Tecnologias de Suporte à Evolutibilidade 45
3.4. Implementação de Testes 56
3.5. Conclusão 61
4. Experimentação com Software de Seguros 63
4.1. Introdução 63
4.2. Refabricação do Módulo “Fiscalidade” 67
4.3. Testes Implementados 93
4.4. O Novo Módulo de Fiscalidade 101
4.5. Conclusão 103
5. Conclusões 105
5.1. Síntese do trabalho realizado 105
5.2. Trabalho futuro 106

Figuras

Figura 1 - Evolução do sistema de informação 3
Figura 2 - Compromisso na relação mudança/confiança no sistema 4
Figura 3 - Camadas da arquitectura da aplicação [Alur 2002] 19
Figura 4 - Diferentes camadas de refabricação 22
Figura 5 - Refabricação baseada em padrões 23
Figura 6 - Especificação de casos de teste 27
Figura 7 – Metodologia proposta 35
Figura 8 - Do requisito à implementação 41
Figura 9 - J2EE, Integrador de aplicações 46
Figura 10 - Padrão *façade* 51
Figura 11 - Diagrama de sequência RPG - Java 52
Figura 12 - Padrão *command* 53
Figura 13 - Padrão *lazy load* 54
Figura 14 - Diagrama de classes para o padrão estratégia 55
Figura 15 - Grafo de operações nas coberturas 58
Figura 16 - Derivação de grafo em cenários possíveis 59
Figura 17 - Framework *Junit* 60
Figura 18 - Arquitectura GIS 66
Figura 19 - Resgate de um recibo de indemnização 71
Figura 20 - Comparação das regras de cálculo de IRS 72
Figura 21 - Interligação de componentes 75
Figura 22 - Classes do sistema de cálculo do Imposto 77
Figura 23 - GestorCalculoIRS , serviço de cálculo de imposto 78
Figura 24 - Comunicação entre RPG e Java (RPG) 79
Figura 25 - CalculoIRSPainel 80
Figura 26 - CalculoIrsStrategy.inicializacao 81
Figura 27 – Selecção do cálculo de imposto - *façade* 82

Figura 28 - Padrão estratégia para o calculador do imposto 83
Figura 29 - Leitura de estratégia a usar 84
Figura 30 - GestorCalculoIrsImpl.getCalculadorAuxiliar 84
Figura 31 - Refabricação do cálculo de provisão 85
Figura 32 - Método calcAS400Sld 86
Figura 33 - Recepção de um pedido Java (RPG) 87
Figura 34 - Geração automática de código 88
Figura 35 - XML para geração da tabela APOLVB 89
Figura 36 - Classes de acesso à base de dados 90
Figura 37 - Implementação do lazy load 91
Figura 38 - Diagrama de actividades para resgate dentro das condições 94
Figura 39 - Diagrama de actividades para resgate fora das condições 95
Figura 40 - Sequências de cálculo dentro das condições 96
Figura 41 - Sequências de cálculo dentro das condições 96
Figura 42 - TestCaseCalculoIrs_test_0001.xml 99
Figura 43 - TestCaseCalculoIrs.test_0001() 100
Figura 44 - Resultado da execução dos testes 100

Tabelas

Tabela 1- Organização de padrões de desenho proposta por GoF 17
Tabela 2 - Camadas J2EE 19
Tabela 3 - Quadro resumo de refabricações e padrões utilizados 92
Tabela 4 - Condições de testes 97
Tabela 5 - Associação de condições e sequências de testes 98

1. Introdução

1.1. Evolutibilidade em Sistemas de Informação

É necessário que as organizações tenham a percepção de que um sistema sólido, ou seja, com grande capacidade de resposta e completamente ajustado às necessidades da organização, não se mantém por si próprio, sem modificações. É necessário melhorar constantemente os seus processos de negócio. No entanto, este processo de melhoria tem de ser bem gerido, considerando a gestão da qualidade (na mudança deve-se garantir que os processos que funcionavam antes continuam a funcionar após), a capacidade de responder às necessidades exigidas e a disponibilidade para novas oportunidades de negócio que possam surgir.

As organizações estão sujeitas a pressões ambientes constantes que as obrigam a uma adaptação constante a essas variantes. Para conseguirem responder a essas expectativas externas, os sistemas de informação das organizações Têm de estar preparados para conseguirem responder eficazmente. A transformação organizacional tem como principal objectivo a melhoraria dos processos de negócio, permitindo, assim, que os sistemas de informação sejam capazes de responder mais eficazmente às exigências da organização. A reengenharia de processos é normalmente o meio para atingir esse fim.

Neste processo contínuo de melhoria, é necessário ter em consideração as vicissitudes organizacionais que levam à estagnação dos sistemas de informação, fazendo com que estes fiquem obsoletos. Para evitar este processo, é necessário um conjunto de linhas orientadoras para gerir o processo de evolução contínua do sistema de informação da organização, não afectando a solidez e fiabilidade do sistema. A causa mais comum que leva o sistema de informação de uma organização a tornar-se obsoleto é a sua falta de evolutibilidade, ou seja, quando se chega a um estado em que não é possível evoluir, nem ajustar às necessidades dos processos de negócio da organização. As razões para isto acontecer podem ser várias: (1) as tecnologias de informação não suportam os processos de negócio; (2) os sistemas de informação não são escaláveis; (3) a organização é demasiado rígida, o que leva a que a esta não se consiga adaptar e dar resposta no momento oportuno às pressões de mercado.

Sistemas sólidos são aqueles que atingiram um forte nível de maturação e que conseguem responder eficazmente às pressões externas e internas a que, a organização é sujeita, e têm capacidade de disponibilizar a informação necessária no momento oportuno, a quem dela necessita. Segundo [Nolan 1979], a maturidade atinge-se quando a integração de aplicações reflecte os fluxos de informação de toda a organização. [Bhabuta 1988] e [Galliers 1991] desenvolveram um modelo de estados de maturidade que pretende mapear o desenvolvimento de sistemas de informação como um progresso em direcção ao planeamento estratégico de sistemas de informação. Nesse modelo o nível máximo de maturação dos SI/TI, quando atingido, proporciona suporte sistemático aos processos organizacionais; o planeamento das tecnologias de informação é efectuado por unidades estratégicas de negócio; e existe responsabilidade ao nível do topo da gestão.

No entanto, é comum verificar que sistemas ditos sólidos entram numa curva descendente e acabam por deixarem lentamente de dar resposta eficaz e se tornarem obsoletos (Figura 1).

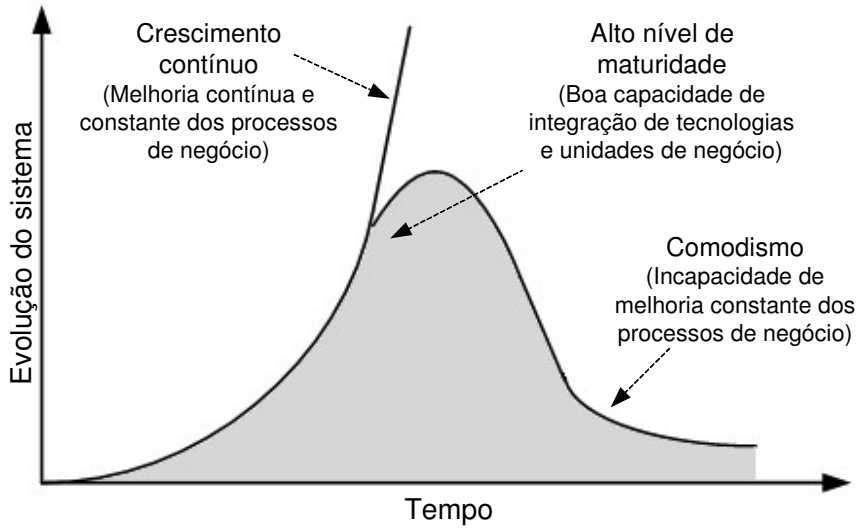


Figura 1 - Evolução do sistema de informação

Os processos devem estar preparados para posteriores alterações e melhoramentos, um sistema escalável não pode viver com procedimentos rígidos. Este é um factor gradual, do qual depende a capacidade do sistema de informação para a inclusão, ou não, de novas unidades de negócio.

As alterações não devem trazer grandes impactos ao sistema. A política deve ser no sentido de garantir muitas alterações com poucos impactos, em vez de poucas alterações com muitos impactos. Este princípio permite manter a confiança no sistema, o que, de outra forma, seria difícil. As alterações com grande impacto, num sistema com alto índice de maturidade, com profundas modificações dos processos de negócio, podem causar grande instabilidade e consequentemente, desconfiança sobre o sistema. O seguinte esquema representa como se deve gerir a relação alterações/confiança no sistema ERP (Figura 2).

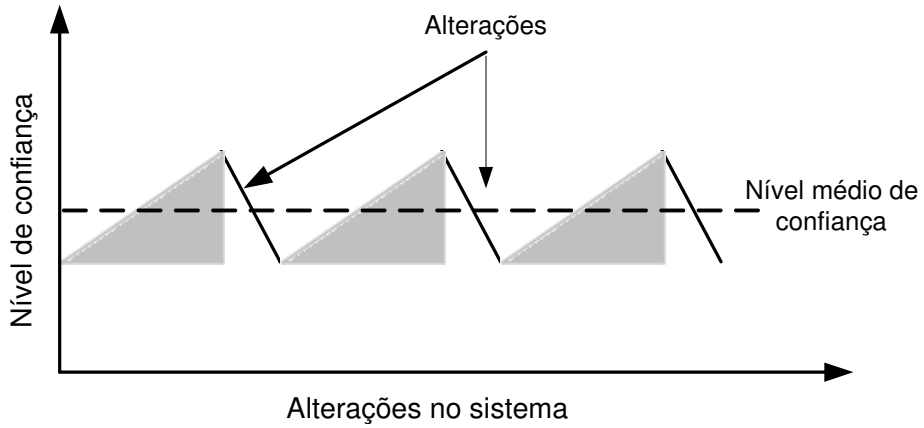


Figura 2 - Compromisso na relação mudança/confiança no sistema

1.2. Objectivos Propostos

A experiência vivida, numa fase atribulada em que os sistemas de informação passaram a partir da década de 90, nomeadamente com o problema do ano 2000, mas principalmente com o *boom* da "webização" das aplicações, permitiu observar o modo como os sistemas de informação foram obrigados a evoluir para alcançarem essas metas. Sistemas legados, sem capacidade para disponibilizar as tecnologias emergentes tiveram de se adaptar à nova realidade imposta pelo mercado.

Tendo o autor desta dissertação participado directamente na evolução de um sistema de informação que esteve (e está) sujeito a essa enorme pressão para um rápido ajustamento tecnológico, propôs-se a prestar um pequeno contributo para solução do problema do rejuvenescimento de aplicações em sistemas legados, em que os objectivos assumidos são:

(1) Propor uma metodologia para adequar tecnologias de refabricação de software a um contexto específico, onde coexistem plataformas e linguagens divergentes, sendo necessário sempre salvaguardar o grau de confiança no sistema, pelo que a mudança deve ser gradual e controlada.

(2) Demonstrar com a aplicação GIS (Gestão Integrada de Seguros, I2S) a possibilidade de preparar um sistema enraízado à 20 anos, agilizando e

rejuvenescendo-o com o auxílio da refabricação do código e recorrendo à metodologia referida no ponto anterior .

1.3. Metodologia de Investigação

Dado o pragmatismo do tipo de problema aqui discutido, o case study é a metodologia de investigação que melhor se adequa nesta dissertação. O case study é uma abordagem empírica para a investigação de um fenómeno contemporâneo no seu contexto real, em que, as fronteiras entre o fenómeno e o contexto não são evidentes [Myers 1997; Benbasat 1999; Choudrie 2005]. Considerando que o problema do rejuvenescimento reside na preparação tecnológica de um sistema de informação, da sua plataforma e das linguagens de programação em que está desenvolvido, a aplicabilidade, ou não, das metodologias propostas consegue-se através de um case study. O uso de um caso real permite analisar a aplicabilidade das técnicas propostas, e comparar os resultados obtidos com os propósitos iniciais.

1.4. Estrutura da Dissertação

Este documento distribui-se por cinco capítulos, sintetizando-se o conteúdo de cada um dos restantes em:

Capítulo 2: contém uma revisão de literatura fundamental para enquadrar as opiniões e contributos de diversos autores com as tecnologias aplicadas neste trabalho.

Capítulo 3: é proposta uma metodologia que pretende ser uma contribuição de uma possível solução para a resolução da problemática em estudo.

Capítulo 4: composto pelo *case study*, onde é feita uma instanciação da metodologia sugerida no capítulo 3.

Capítulo 5: apresentam-se as considerações finais sobre o trabalho, e uma análise dos objectivos atingidos.

2. Desenvolvimento e Evolutibilidade

2.1. Introdução

Actualmente há uma grande pressão para que os sistemas de informação das organizações se adaptem de uma forma reflexa às exigências que lhes são impostas por entidades concorrentes, partes interessadas, e outras variáveis de ambiente. Para que seja possível um grau tão elevado de eficácia na resposta a essas necessidades, os sistemas de informação devem ser ágeis e ter capacidade de absorver as alterações impostas de uma forma natural, evitando o aumento da complexidade do sistema.

As propostas mais recentes para o desenvolvimento de software são as metodologias ágeis, que incluem, nos seus ciclos de desenvolvimento, técnicas que permitem a alteração estrutural do código, sem comprometer o comportamento do sistema, dotando o código de simplicidade e capacidade de absorção de alterações que lhe sejam impostas. As metodologias ágeis de desenvolvimento assentam os seus princípios em muitas iterações, em que cada iteração contém poucas alterações e é disponibilizada em ciclos curtos. Para que sejam possíveis essas rápidas iterações, o código deve estar preparado e receptivo à inclusão de novas funcionalidades.

A refabricação de código [Fowler 1999] é uma técnica fundamental para o rejuvenescimento de aplicações. Esta técnica, mais comunmente referida na literatura como *refactoring*, consiste em alterar código existente com a intenção de o simplificar e reorganizar. Nesse sentido, o *refactoring* surge muitas vezes associado ao conceito *design pattern* (padrões de desenho) [Gamma 1994]. Estes sugerem os modelos para os quais se pretende evoluir. Por outras palavras, os *design patterns* acabam por ser o objectivo do processo de *refactoring*.

No entanto a preparação da infra-estrutura de suporte para a conjugação destas características deve ser complementada com a especificação de testes apropriados de forma a garantir todos os pressupostos inerentes a essas técnicas, tal como a perseverança do comportamento do sistema.

2.2. Metodologias de Suporte ao Desenvolvimento

Na abordagem tradicional aos ciclos de vida de software, o clássico modelo em cascata, proposto por *Royce* em 1970 [Davis 1988] define o ciclo de vida como sendo constituído pelas fases de análise de requisitos do sistema, análise de requisitos do software, desenho prévio, desenho detalhado, desenvolvimento (codificação e depuração), testes e manutenção. Por sua vez, *Bohem* em 1980 [Boehm 1988] sugeriu o modelo em espiral, onde incluía a fase de prototipagem numa abordagem que se distinguia da anterior, principalmente pelas repetidas iterações em ciclo de todas as suas fases do ciclo de vida.

Recentemente, as já referidas, metodologias ágeis trouxeram um grande impacto às organizações e departamentos de desenvolvimento de software com uma nova abordagem [Boehm 2002]. Este tipo de abordagem utiliza técnicas que permitem, de uma forma contínua, o ajuste e adaptação a novas alterações. De entre as metodologias ágeis existentes, o *Extreme Programming* (XP) [Beck 1999; Heinecke 2002] é o que tem tido maior aceitação e protagonismo [Abrahamsson 2002].

Brian Foot [Foote 1994] associou as fases do ciclo de vida a padrões, baseando-se no ciclo em espiral de *Bohem* [Boehm 1988], para a definição das quatro camadas que compõem, na sua perspectiva, o ciclo de vida: (1) prototipagem; (2) expansão do protótipo (novas funcionalidades); (3) consolidação do programa para suportar evolução e reuso; (4) evolução das agregações entre as hierarquias de heranças; (5) criação de super classes abstractas. Nesta abordagem, *Foot* define como *padrão* os métodos comuns que se verificam ocorrer no desenvolvimento de software em diferentes linguagens. Esta perspectiva está mais orientada às linguagens orientada a objectos porque o reuso, e os princípios de evolução contínua, estão inerentemente associados aos objectos. Estes permitem, devido às suas características de hierarquia de heranças, a reutilização de código de uma forma muito mais elegante que numa linguagem não orientada a objectos.

A proposta de ciclo de vida de *Brian* caracteriza-se pela importância dada à necessidade de se alterar o software nas várias iterações por que passa o código, e a capacidade que o software possuí para suportar essas alterações. A fase de prototipagem pretende ser curta com a apresentação rápida de resultados. Nesta fase não se pressupõe a existência de uma estrutura de objectos já hierarquizada. Pretende-se, antes uma análise funcional, que responda às necessidades do cliente e em que a concepção não representa mais que um esboço inicial do sistema. Na seguinte fase (expansão), o objectivo é efectuar algumas alterações que não foram inicialmente visionadas na fase inicial do protótipo. Normalmente, as ligeiras modificações acrescentadas ao código originadas pela implementação dessas funcionalidades, podem originar a deturpação da estrutura do modelo inicialmente concebido (por exemplo, através da inclusão de *flags* no meio de código). A evolução do software desta forma desorganizada faz com que a estrutura e organização dos objectos fiquem afectadas. A reutilização do software, por consequência, fica também afectada. Existem várias razões para que isto aconteça, desde a construção do protótipo, em que o foco principal é a funcionalidade em vez da estrutura dos objectos, o que não está errado, uma vez que o objectivo nessa fase é proporcionar rapidamente algo executível para o cliente. Também a fase de expansão proporciona essa degradação,

quando, por vezes, surgem requisitos que geram conflitos com a organização dos objectos existente.

Verifica-se, então, a necessidade de incluir no ciclo de vida do desenvolvimento de software, uma fase que proporcione a melhoria da estrutura do software ou aumente a sua futura reutibilidade. Este é um problema que as noções dos ciclos de vida tradicionais não resolveram. Não é possível que os objectos reutilizáveis surjam de uma análise inicial para o domínio de um dado problema. Pelo contrário, essa capacidade de reutilização adquire-se com a evolução do sistema. Como resultado, os objectos devem ser alterados para proporcionar essa capacidade, devendo esta actividade ser considerada como uma fase intrínseca ao ciclo de vida.

Estas alterações ao código, que visam a melhoria da sua estrutura, através da sua reconstrução, caracterizam-se por técnicas de refabricação. Essas técnicas visam a modificação da estrutura do software (quer de alto, quer de baixo nível), com o objectivo de o simplificar, eliminar duplicação de código e corrigir deficiências devidas a falhas na análise de requisitos, ou a alterações desses mesmos requisitos durante o ciclo de desenvolvimento. Dota-se, assim, o software de características que lhe permitem adquirir facilmente novas funcionalidades e alterações necessárias ao rejuvenescimento de aplicações.

A necessidade de rejuvenescer as aplicações está principalmente associada à resolução de problemas, ou a necessidades funcionais reportadas pelo utilizador. As funcionalidades pretendidas pelos utilizadores nem sempre são possíveis implementar com o sistema legado, por dificuldade de resposta deste. Surge, assim, a necessidade de restruturar as aplicações, através de técnicas de refabricação [Tokuda 1999], garantindo que o comportamento da aplicação se mantenha. O conceito de refabricação pode ser definido como: uma alteração ao sistema que não altera o seu comportamento, mas potencializa a sua qualidade, simplicidade, flexibilidade, e compreensão [Fowler 1999].

Metodologias Ágeis e eXtreme Programming

O termo *Agile* denota a qualidade de ser ágil e responder ao mundo empresarial que reclama por processos rápidos no desenvolvimendo de software [Abrahamsson 2002]. Os valores advogados no manifesto *Agile* [Beck 2002] baseiam-se na necessidade de interacção entre a equipa de desenvolvimento (mais do que a necessidade de processos bem definidos), na existência de documentação útil e apenas a necessária, na colaboração entre equipa de desenvolvimento e o cliente, e na necessidade de reagir à mudança em vez de seguir um plano. Os 12 princípios constados nesse manifesto residem no grande objectivo de responder às necessidades do cliente e de como preparar e motivar a equipa na actividade de desenvolvimento de software, para o conseguir eficazmente. No entanto, o conceito *Agile* ainda não reuniu o consenso e daí tem resultado o aparecimento na literatura de várias metodologias de desenvolvimento que derivam da famíla *Agile*. De todas essas, o XP (eXtreme Programming) foi o ponto a partir do qual surgiram outras abordagens, tais como: *Crystal Methods* [Crystal]*, Feature-Driven Development* [FDD] e *Adaptive Software Development* [HighSmith 2000].

O desenvolvimento em XP tem como objectivo a satisfação de uma determinada necessidade identificada pelo cliente, ignorando tudo o resto que não é identificado como uma necessidade de negócio. Os requisitos são obtidos como descrições que os utilizadores (cliente) fazem das suas necessidades. Essas descrições são depois decompostas em tarefas. A filosofia do XP quanto à fase de concepção é minimalista e pragmática [Deursen 2002], em que o objectivo é uma análise rápida de todo o sistema e em que a construção de código começa logo na primeira iteração. O XP não distingue analistas de programadores, porque todos são responsáveis por fazer a análise e programar. O princípio orientador é que a fase de análise apenas deve responder aos requisitos funcionais do cliente e devem ser construídos os testes necessários para garantir a qualidade dessa funcionalidade. O ciclo de desenvolvimento segundo esta metodologia envolve as seguintes iterações [Beck 1999; Crispin 2002]:

(1) *Fase de codificação*. O código é o principal veículo de comunicação do XP, por isso deve ser suportado por ferramentas de edição que possibilitem a representação gráfica do sistema, assim como a geração automática de código. A programação deve ser feita por *pair programming*. O objectivo da programação por pares é garantir que a propriedade de conhecimento esteja distribuída e a validação do código feita em conjunto seja mais eficiente. A existência de uma pessoa a desenvolver e outra a assistir permite que quem assista consiga captar erros, a que o programador possa não estar atento.

(2) Refabricação [Fowler 1999; Roberts 1999]. Para manter o código simples, o ambiente de desenvolvimento deve estar suportado em técnicas e ferramentas de refabricação que simplificam e estruturam o código. A refabricação consiste em modificar a estrutura do software, tornando-o mais legível e sem quebrar o comportamento do sistema com essas alterações [Opdyke 1992]. As operações típicas de refabricação são [Astels 2002]: renomear classes e métodos, alterar tipos de dados, generalização, etc. Outro tipo de refabricação é a transformação e adaptação de partes de código baseado em *design patterns* [Gamma 1993; Cinnéide 2000] que melhor se ajustam à estrutura do código [Kerievsky 2004].

(3) *Fase de testes automatizada*. Os testes são um veículo para a especificação de requisitos. Os programadores em XP devem começar por escrever os testes que vão garantir a qualidade do software. Ao começar por escrever os testes, os programadores são obrigados a pensar nos resultados esperados pelas funcionalidades que vão implementar [Beck 1998; Beck 1999].

(4) *Diálogo com o cliente*. As funcionalidades do sistema são definidas através das necessidades descritas pelo cliente. É através das *user stories* [Grunbacher 2002] que se devem identificar claramente os requisitos. Esta prática permite que a equipa de desenvolvimento e o utilizador partilhem de uma visão comum. Para isso, é sugerido que esteja sempre um utilizador disponível perto da equipa de desenvolvimento para esclarecer qualquer questão.

(5) *Fase de concepção*. Documentação mínima necessária que permita simplificar e corrigir qualquer incoerência decorrente das histórias descritas.

A refabricação pode ser incluída no ciclo normal de desenvolvimento de software, sendo esta a fase de preparação do software que irá permitir posteriormente a implementação de um novo requisito (funcionalidade) no sistema. Um dos meios de desenvolvimento mais favoráveis para a aplicação de técnicas de refabricação verifica-se em o*bject oriented frameworks* [Opdyke 1992], porque o uso de ferramentas automáticas nestes ambientes é mais apropriada.

Qualidade do Software

O XP mostra-se como uma abordagem para resolução de problemas em curto prazo, permitindo responder de uma forma eficiente ao desenvolvimento num ambiente em mudança. No entanto, como o desenvolvimento não é sustentado com uma boa análise de requisitos, não é possível especificar os testes adequados que permitam garantir a qualidade do código considerado crítico.

As metodologias desenvolvimento de software baseadas nos clássicos ciclos de vida do software possuem características que se ajustam melhor para intervenções em projectos de médio/longo prazo. O tipo de especificações que este modelo fornece permite testes rigorosos e outras técnicas de avaliação incluídas na engenharia de software, capazes de fornecer melhores (mas também mais caros) mecanismos para o desenvolvimento de software crítico e com qualidade controlada [Manzoni 2003].

O RUP [Rational 1998] é um processo de engenharia de software que se define numa abordagem disciplinada para a associação entre tarefas/responsabilidades e o processo de desenvolvimento, descrevendo um conjunto de actividades necessárias para a transformação de requisitos em software. É assim, um processo genérico que permite gerir de uma forma controlada e calendarizada a produção de software que satisfaça os requisitos dos utilizadores.

O RUP usa o UML para modelar o software e descrever como aplicar as melhores práticas da engenharia de software para todas as fases do ciclo de vida de software crítico. É também um processo extensível, isto é, pode ser adaptado a um ambiente com necessidades específicas, consistindo num processo iterativo que segue o padrão cascata composto por levantamento de requisitos, análise, modelo (desenho), implementação, testes e desenvolvimento. No entanto, esta sequência de processos tem alguns inconvenientes, sendo um desses, a inflexibilidade no ajustamento a uma eventual modificação de requisitos depois de iniciado o processo. Outro inconveniente é o facto dos testes entrarem muito tarde dentro do processo, sendo uma fase crítica, quer para o utilizador quer para o programador, em que o primeiro só tem contacto com o produto na fase final do processo, e do outro lado, o programador não tem o necessário *feedback* que o possa alertar para algum desvio dos requisitos identificados na concepção do sistema. Isto obriga que, a análise de requisitos descreva fidedignamente os requisitos dos utilizadores. A concepção do sistema por este processo assenta na construção de casos de uso, conseguindo-se uma descrição do funcionamento do sistema bem documentada, onde se descreve a interacção entre actores e os artefactos que compõem o sistema. Esta visão do sistema permite a análise dessas possíveis interacções dos utilizadores com o sistema, e assim especificar um conjunto de testes apropriado considerando todas as possíveis combinações.

Uma abordagem possível para a especificação de testes é a orientada à funcionalidade [Fraikin 2002], em que o conceito de funcionalidade é basicamente a necessidade (pedido) do cliente. Esta metodologia sugere uma implementação por funcionalidade, não devendo esta ser muito complexa, e caso seja necessário, esta deve ser subdividida de modo a que se reduza a complexidade. Desta metodologia faz parte, também, a obrigação de se especificarem os diagramas de sequências que mostram o modo de interagir das funcionalidades a desenvolver. *Labiche* [Labiche 2001] por outro lado, propõem para especificação de testes as seguintes fases: (1) desenho dos diagramas de sequências; (2) derivação para detalhar os diagramas de sequências; (3) desenho do correspondente diagrama de actividades; (4) representar sob a forma de grafo as actividades identificadas nos diagramas de sequências; (5) definir restrições de acção

(através por exemplo de OCL- *Object Constraint Language*) [Rational 1997] ; (6) definir cenários possíveis através de modelos; (7) definir condições iniciais e resultados esperados.

No oposto, a esta abordagem orientada ao processo, surgem os princípios propostos pelas metodologias ágeis. Sendo o desenvolvimento num ambiente ágil regido pelos princípios já descritos, surge a dúvida de como planear um projecto deste tipo, em que não havendo uma análise inicial dos requisitos de uma dada funcionalidade à partida seria impossível estabelecer prazos para entregas. *Kent Beck* e *Martin Fowler* [Beck 2000] advogam o princípio de que os projectos são contra-natura relativamente aos programadores, e por isso, não devem ser aplicados os princípios de gestão de projectos no ciclo de desenvolvimento de software. Isto porque, sendo o desenvolvimento sempre realizado por pessoas, o importante é garantir a sua motivação, evitando-se dispersar as suas atenções em assuntos que não lhes interessam. Se tal for garantido, então os resultados surgem naturalmente e consequentemente, estão criadas as condições para que o cliente tenha o seu produto em tempo útil. As entregas urgentes com prazos no limite (o típico: "fazer hoje o que devia ser para ontem") não fazem com que os programadores façam o seu trabalho mais depressa, antes pelo contrário, a urgência leva ao pânico, à fadiga e à quebra de comunicação. Por outro lado, sendo o objectivo do planeamento conseguir prever o futuro, e sendo este cada vez mais imprevisível, a tentativa de o prever é cada vez mais um esforço inglório.

Em vez das quatro variáveis da gestão (custo, qualidade, tempo e âmbito) *Beck* e *Fowler* defendem que o planeamento deve ser suportado pelas histórias dos utilizadores (descrições das suas necessidades relativas ao sistema de informação), em que o custo é relativo a cada história ou necessidade descrita pelo utilizador. Por sua vez, o conjunto das histórias e a sua complexidade permitem estimar o tempo de execução do plano. É, no entanto, necessário considerar também as restrições de negócio e tecnológias que se vão encontrando à medida que decorre o projecto. Segundo esta perspectiva, o plano concretiza-se pela compra ou não de histórias e quanto maiores e complexas forem essas histórias mais tempo demorará a sua concretização.

2.3. Refabricação de Software

Tendo a refabricação como principal objectivo a simplificação e estruturação do código, é necessário identificar quais são essas estruturas-alvo que permitem dotar o código com essa arrumação e simplicidade. Essas estruturas referidas são os padrões de desenho (*design patterns*) [Gamma 1993].

Os padrões de desenho são estruturas de código que se podem encontrar nos mais diversos ambientes e que não estão dependentes do contexto em que são usados. Por exemplo, ao codificar um programa por vezes há certas soluções "tipo" que permitem a resolução de diferentes problemas. Esses modelos aplicados enúmeras vezes aos mesmos problemas são designados padrões de desenho. As vantagens do uso de padrões são várias, desde a simplificação da compreensão do código, até à redução do tempo de codificação pelo programador, isto porque para resolver determinados problemas pode recorrer a bibliotecas de padrões onde pode facilmente encontrar uma solução *tipo* que lhe indica a solução, sem necessidade de "reinventar a roda".

Existem diversos padrões de desenho identificados e publicados por vários autores [Gamma 1993; Tokuda 1999; Kerievsky 2004]. No entanto, é sempre possível que num contexto específico de desenvolvimento de aplicações e em determinadas áreas de negócio surjam novos padrões. Sempre que se verificar que um determinado *template* de código possui diversas aplicações, pode-se estar perante um provável candidato a um novo padrão de desenho. *Erich Gamma* [Gamma 1993] caracterizou os padrões de desenho sob três tipos, de acordo com a sua função:

(1) *Concepcionais*. São relativos aos padrões usados nos construtores. Todo ou parte da criação de uma entidade em que as partes são polimórficas.

(2) *Estruturais*. Padrões para a composição de objectos simples em objectos complexos. Implementam as conexões, interfaces e gerem as comunicações entre os objectos. Consegue-se ganhar flexibilidade quando há a capacidade de mudar a composição de objectos em tempo de execução.

(3) *Comportamentais*. Descrevem como um grupo de objectos coopera em tempo de execução para concretizar uma tarefa que nenhum objecto poderia concretizar por si só.

No célebre livro do *Gang of Four* [Gamma 1994], os autores avançaram para um agrupamento de vários padrões (23) segundo os três grupos atrás definidos (Tabela 1).

Tabela 1- Organização de padrões de desenho proposta por GoF

Padrões concepcionais	*Padrões comportamentais*
Abstract factory	Chain of responsability
Build	Command
Factory method	Interpreter
Prototype	Iterator
Singleton	Mediator
	Memento
Padrões estruturais	Observer
	State
Adapter	Strategy
Bridge	Template method
Composite	Visitor
Decorator	
Facade	
Flyweight	
Proxy	

No entanto, estes são apenas alguns padrões que num determinado contexto foram identificados por estes autores [Gamma 1994]. Existem muitos outros, não se podendo limitar e quantificar os padrões existentes, porque em qualquer situação é possível identificar novas situações não previstas e consequentemente novos padrões. Para além destes padrões, pode identificar-se, ao nível da arquitectura de aplicação, um outro nível de abstracção com a padronização da estrutura da aplicação. Segundo *Martin Fowler* [Fowler 2002], existem três camadas distintas no desenho da arquitectura de uma aplicação, a de base de dados, a de lógica de domínio e a de serviços. Esta visão do autor, agrupando os padrões em diferentes camadas, permite a organização destes de acordo com a sua utilização no contexto da arquitectura global da aplicação.

Na camada de base de dados realçam-se os padrões usados na manipulação de tabelas relacionais, mapeamento das tabelas para os respectivos objectos e a comunicação com o sistema que suporta a base de dados [Fowler 2002]. O armazenamento de informação pode ser suportado em diferentes formatos, sendo o XML, ou tabelas relacionais, os meios mais comuns de suporte. Em ambos os casos são necessários estabelecer padrões que permitam: No caso da comunicação com as respectivas bases de dados, é necessário obter uma conexão para se aceder à informação, considerando que as plataformas onde estão a ser carregados os objectos e a origem da informação com que esses objectos são carregados podem não ser as mesmas. É necessário mapear a informação obtida, com o objecto destino (tabela) e seleccionar os registos das tabelas que são necessários para o carregamento do objecto origem. O uso do SQL é uma forma de obter esses registos. Relativamente ao sincronismo, este deve ser garantido entre o estado dos objectos carregados e o estado destes na base de dados. É necessário garantir que processos concorrentes não carreguem o mesmo objecto, e em vez disso haja apenas um único objecto, sendo este instanciado o número de vezes necessárias. Uma boa prática é o carregamento dos objectos só quando estes forem necessários, evitando-se, assim, dois possíveis problemas: o gasto desnecessário de tempo com o carregamento desses objectos, e possíveis problemas de concorrência com processos paralelos a alocarem os mesmos recursos.

Os padrões são, na sua essência, assunções de boas práticas catalogadas e que podem ser utilizados sempre que se adoptar a mesma solução em contextos diferentes. O grande benefício dos padrões está na possibilidade da sua reutilização, melhorando a estrutura do código ao aplicar as boas práticas embutidas nos padrões adoptados, e evitando estar constantemente a "reinventar a roda" para os mesmos problemas.

O catálogo de padrões em J2EE [Johnson 2002] divide-se em três camadas [Alur 2002]: (1) A camada de apresentação engloba os padrões relacionados com a interface das aplicações. (2) A camada de negócio, por sua vez agrupa os padrões que suportam a lógica de negócio, sendo esta camada que trata os objectos que representam a informação

persistente da aplicação. (3) A camada de integração contém os padrões que são utilizados para integrar as aplicações J2EE com outros sistemas legados.

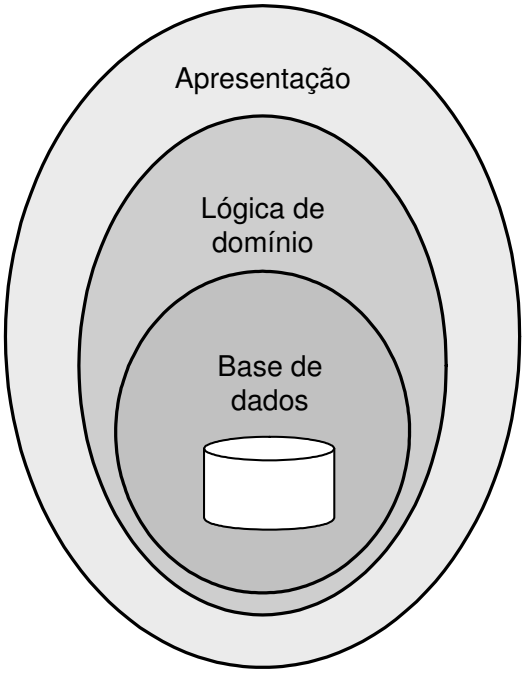


Figura 3 - Camadas da arquitectura da aplicação [Alur 2002]

No entanto, outras abordagens, como em [Marinescu 2002], detalham em cinco as camadas do J2EE, de acordo com a Tabela 2.

Tabela 2 - Camadas J2EE

Camada	Responsabilidades	Tecnologia
Apresentação	Interface com o utilizador	*Jsp/Html/Java Script*
Aplicação	Processos de trabalho (workflow), Interacção com serviços	*Servlets*
Serviços	Lógica dos processos de workflow, função idêntica à de uma fachada (padrão)	Sessões EJB (*session beans*)
Domínio	Lógica de negócio	Entidades EJB (*entity beans*), *Java Objects*
Persistência	Armazenamento físico do estados dos objectos de negócio	Mapeadores de bases de dados

Sendo os padrões de desenho facilitadores de codificação, verifica-se, no entanto, inconvenientes pelo seu uso. O principal inconveniente verifica-se quando se analisa com mais profundidade o significado do próprio termo '*pattern*'. Este termo evidência rigidez e falta de flexibilidade. Ao aplicar padrões de desenho no desenvolvimento de software está-se a replicar pequenos blocos de código com o mesmo comportamento ao longo de toda a aplicação. No entanto, as aplicações evoluem e os padrões adoptados no início do desenvolvimento dessas aplicações podem não suportar a evolução do sistema, devido a novas exigências, ou a novas funcionalidades e alterações, ou outros quaisquer motivos que obriguem a mudanças no código já desenvolvido. Pode, então, surgir a necessidade de reajustar padrões que, dada a evolução do sistema, verifica-se não serem já os padrões mais adequados.

A Necessidade de Refabricar

Quanto à necessidade de refabricar para evoluir, as metodologias de desenvolvimento clássicas propõem que as fases prévias de análise devem prever a evolutibilidade do sistema e, desta forma, prepará-lo logo de início para as possíveis alterações (evoluções) que o sistema venha a sofrer. O processo de engenharia de software quando instanciado pelo RUP inclui as fases de: especificação de tarefas, calendarizar, entregas de produtos e atribuição de responsabilidades com o objectivo de desenvolvimento de software com grande qualidade e que responde às necessidades dos utilizadores finais. No entanto, a experiência diz que devido às enúmeras variáveis e factores externos, ditados pelas leis de mercado, políticas, mercado, etc., é impossível antever com alguma razoabilidade a evolução que o sistema venha a sofrer a longo/médio prazo.

Alternativamente ao proposto pelas metodologias clássicas de desenvolvimento de software, as metodologias ágeis aparecem propondo uma abordagem menos analítica, quanto à fase de estudo prévia de necessidades para o sistema a desenvolver, e mais

próxima do desenvolvimento para a funcionalidade pretendida no momento, numa orientação de evolutibilidade do sistema passo a passo.

Como o desenvolvimento segundo uma metodologia de desenvolvimento ágil propõe que se foque a parte (funcionalidade) em vez do todo (sistema integrado), é necessário o auxílio de técnicas que permitam alterar a estrutura do software, para que em cada iteração do ciclo de desenvolvimento do software seja possível incorporar as novas funcionalidades sem modificar o comportamento do sistema observado até então, ou seja, recorrendo à refabricação.

No entanto, a refabricação pode conter múltiplos significados no âmbito da restruturação de aplicações, dependendo dos princípios e dos objectivos com que se aplicam essas técnicas. No trabalho de *Opdyke* [Opdyke 1992], as técnicas apresentadas contribuem para definir formas de manipular o código existente com o objectivo genérico de o simplificar, sem no entanto a preocupação de enquadrar essas modificações com padrões já formalizados [Cinnéide 2000]. Alguns exemplos dessas técnicas genéricas de simplificação de código são a movimentação de métodos entre classes, renomeação de variáveis e métodos, criação de novos métodos através da extracção de código de um método mais complexo. Existem vários tipos de granularidades na aplicação de refabricação. As alterações que envolvem um volume inferior ao tamanho de uma classe podem ser consideradas de menor volume, como por exemplo, a renomeação de variáveis, extracção de métodos, etc. Elevando o grau de abstracção, aumenta igualmente o nível de aplicabilidade de refabricação no que concerne à arquitectura da aplicação. *Fowler* [Fowler 1999] refere-se a quatro grandes refabricações: (1) separação de heranças; (2) conversão de procedimentos em objectos; (3) separação do domínio da apresentação; (4) extracção de hierarquias.

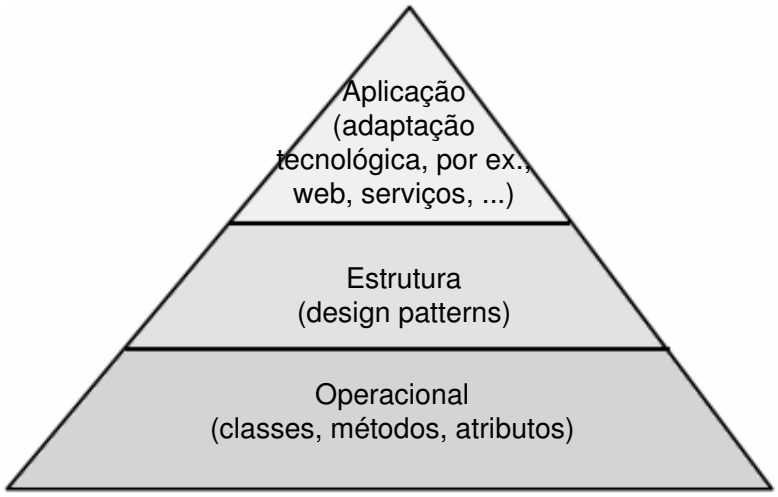


Figura 4 - Diferentes camadas de refabricação

A necessidade de restruturar a base de codificação, passando de uma linguagem procedimental para uma linguagem OO (o*bject oriented*) é uma actividade normalmente muito complexa, relativizando à dimensão da aplicação (Figura 4). Uma operação deste tipo obriga a que se conheçam todos os processos da aplicação e inter-relações entre outros subsistemas.

Frequentemente, verifica-se ser necessário fazer um levantamento de requisitos do sistema actual, tal como nos processos de reengenharia, de modo a identificar e analisar os processos de negócio implementados. Isto acontece principalmente quando se refere a sistemas de grande dimensão, mais precisamente, de sistemas maturos que sofreram várias alterações ao longo do seu processo de evolução, mas que não foram complementadas com a respectiva documentação, obrigando, assim, a extrair as regras de negócio directamente do código.

Parecendo uma reengenharia, a refabricação nada tem a ver com isso. Comparativamente, a reengenharia é uma actividade mais radical, em que o objectivo é um corte com os processos e regras previamente instauradas e (re)formular tudo (regras e processos) de uma forma mais eficiente. Por sua vez, a refabricação é uma linha de continuidade, em que se assume que os processos e regras de negócio existentes são

eficientes e funcionam de acordo com o esperado, pretendendo-se, no entanto, restruturar o sistema de modo a permitir que as intervenções nesse sistema se tornem mais eficientes na resposta às novas necessidades e exigências de negócio [Kerievsky 2004].

Antes de se efectuar a refabricação sobre o código existente, é necessário identificar as áreas onde actuar e como actuar. Existem vários indícios no código que indicam a necessidade de aplicar estas técnicas [Fowler 1999]. Esses indícios como a existência de código duplicado, métodos longos, etc., sugerem que se intervenha no código de forma a clarificá-lo e simplificá-lo. O UML [Rumbaugh 1998], quando suportado por uma ferramenta, pode ser um excelente utilitário para analisar graficamente o código, de forma a identificar as zonas do código carenciadas de refabricação [Astels 2002]. O diagrama de classes e o diagrama de sequências são especialmente úteis para a refabricação. O primeiro permite uma visualização estática das classes que constituem o sistema e respectivas relações, enquanto o segundo fornece uma perspectiva dinâmica das sequências de interacção entre classes. A acção de refabricação pode ser auxiliada com a utilização de padrões catalogados. Tendo a refabricação como motivação a simplificação de código com a implementação de boas práticas de codificação, a utilização de padrões de desenho já conhecidos fornecem esses requisitos (Figura 5).

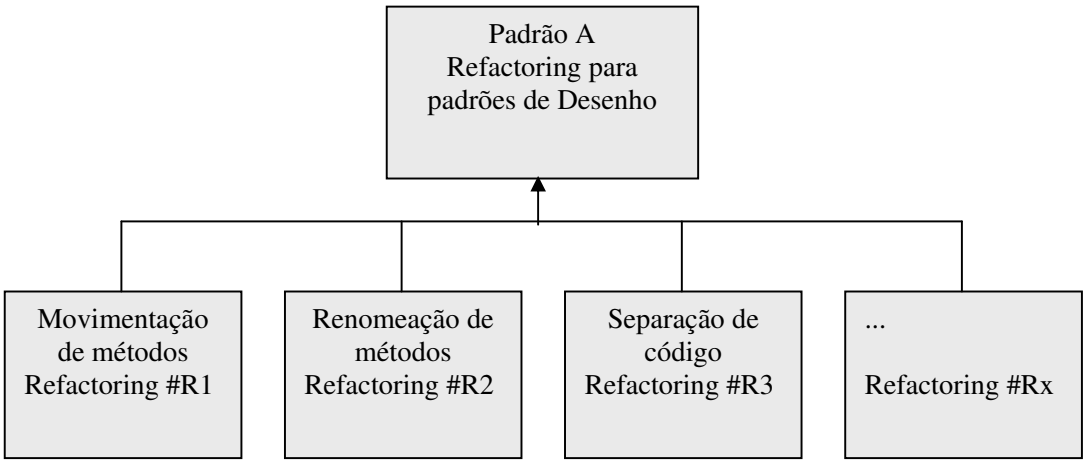


Figura 5 - Refabricação baseada em padrões

2.4. Testes de Conformidade

A refabricação origina alterações profundas da estrutura do software, e tendo como princípio a não alteração do comportamento do sistema, é necessário garantir que esse comportamento se mantém através da implementação de testes. Por isso, é necessária uma metodologia que permita identificar quais os testes que devem ser executados sobre um componente.

O desenvolvimento baseado numa metodologia XP tem como requisito a realização de testes de unidade, com forte ênfase na implementação e especificação de testes na fase inicial do ciclo de desenvolvimento. O objectivo deste tipo de especificação de testes, além de servir de documentação ao desenvolvimento realizado, é permitir que em cada ciclo de desenvolvimento seja possível implementar novos testes sem se perderem os anteriores [Harrold 2001]. Garante-se, assim, que as funcionalidades antigas não se perdem ou se alteram com os novos desenvolvimentos.

O XP divide os testes em duas categorias [Fowler 2003]: testes de unidade e de aceitação. Os testes de unidade são normalmente escritos para testarem pequenas unidades constituídas por uma classe ou por um pequeno grupo de classes. Os testes de aceitação são executados e definidos pelos utilizadores e testam todo o sistema do início ao fim. Uma outra classificação possível para os tipos de casos de teste, é definida de acordo com a origem pela qual se consegue fazer a especificação de testes [Zhu 1997]. Assim, de acordo com esse tipo de classificação podem existir:

(1) *Testes baseados em especificação*. Os casos de teste são especificados com base nos requisitos do software. Neste caso os testes são adequados se forem executadas e validadas todas as funcionalidades para que o software foi desenvolvido.

(2) *Testes baseados no programa*. Os requisitos de testabilidade são sobre o programa a testar e definem que o programa é adequado conforme o propósito para que foi desenvolvido.

Para a execução de testes de unidade foi desenvolvida uma ferramenta , o *Junit* [Junit 2003]. Esta ferramenta facilitadora da implementação de testes, fornece o meio que permite, de uma forma simples, escrever testes e executá-los [Beck 1998]. A codificação de um teste de unidade recorrendo a esta ferramenta consiste em: criar e instanciar os objectos que vão interagir durante a execução do teste; codificar as acções a executar com os objectos; e validar o resultado, através da verificação do estado final dos objectos.

Num desenvolvimento em XP a especificação de testes com recurso ao *Junit* fica facilitada porque permite, tal como requisito da metodologia, escrever os testes na fase inicial do desenvolvimento[Labiche 2001; Fraikin 2002; Stotts 2002]. Em XP os testes são um veículo para a especificação de requisitos [Crispin 2002]. Os programadores devem começar por escrever os testes que vão garantir a qualidade do software. Ao começar por escrever os testes, os programadores são obrigados a pensar nos resultados esperados pelas funcionalidades que vão implementar. No entanto, deve-se implementar uma estratégia de especificação de testes para garantir que todos os casos possíveis de execução de uma funcionalidade estão previstos. Com uma sequência bem definida, pode-se aumentar o número de execuções de cada teste e incluí-los mais cedo no modelo de testes [Parrish 2001].

Uma alternativa ao *Junit* é a inclusão de validações embutidas nas classes [Wang 1997; Wang 1999]. Isto consiste num *framework* em que as classes, para além das funcionalidades implementadas através dos seus métodos, contêm, também, métodos que permitem validar o comportamento da classe. Nesta abordagem, o *framework* tem que possuir um ambiente de execução paralelo (ambiente de testes) ao ambiente de produção, que permita a execução das validações sobre o comportamento dos objectos.

De acordo com a natureza dos testes que se está a implementar, estes podem ser divididos em diferentes tipos. O objectivo é definir testes que permitam garantir a qualidade do software a vários níveis: como *parte* isolada com uma função específica, como *parte* integrada que interliga com outros componentes, e como *parte* que faz parte

de um sistema como um todo. Considerando isto, os testes podem ser classificados em três grupos [Boehm 1988] :

(1) *Testes unitários*: pretendem garantir a correcta implementação dos requisitos especificados para cada módulo.
(2) *Testes de integração*: pretendem garantir o correcto funcionamento entre módulos, quando estes se integram no sistema.
(3) *Testes de aceitação*: devem garantir que o software satisfaz os objectivos de negócio de acordo com os requisitos do cliente. Estes testes são da responsabilidade do cliente.

Relativamente aos testes unitários pode-se identificar vários casos de teste [Tsai 1999], tais como:

(1) *Testes sobre partições/conjuntos*: Consiste em definir conjuntos de acordo com os tipos de objectos ou grupos de objectos para um determinado número para execução de testes.
(2) *Testes de stress:* sobrecarregar o sistema, para ver como este se comporta. Analisar possíveis problemas com memória.
(3) *Testes negativos:* gerar propositadamente erros, para analisar como o sistema se comporta perante uma situação de erro. Desejavelmente deveria comunicar uma excepção com o respectivo erro.
(4) *Testes aleatórios:* instanciar objectos aleatoriamente para executar os mesmos testes.
(5) *Tipos de objectos dinâmicos:* fazer os mesmos testes com tipos de objectos diferentes, para garantir que os *casts* estão a ser propriamente executados.
(6) *Teste de extensibilidade:* Com *design patterns* [Tsai 1999; Labiche 2000] dinâmicos, garantir que em execução a aplicação carrega os objectos desejados.

A especificação de testes com base numa análise de requisitos efectuada previamente, consiste em, para cada sequência de acção identificada nos diagramas de sequências, instanciar o estado inicial dos objectos e definir o estado esperado como válido no fim da acção.

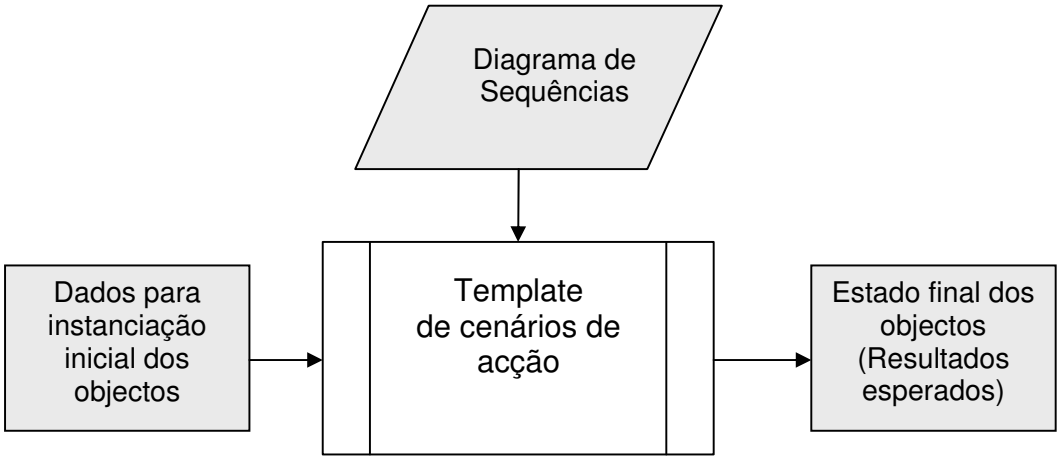


Figura 6 - Especificação de casos de teste

2.5. Conclusão

Apesar de se verificar que a evolutibilidade no desenvolvimento de software é uma área com grande interesse de estudo, como demonstrado pelas enúmeras obras relacionadas publicadas, verifica-se uma falha quando se procuraram obras de referência que abordem o triângulo composto por refabricação, padrões e testes de conformidade, como um todo aplicado ao contexto de evolução de sistemas legados. Estes três vértices estão directamente inter-relacionados, e devem ser estudados com o objectivo de sistematizar as suas dependências no âmbito do rejuvenescimento de aplicações.

Neste capítulo foram apresentados os fundamentos que servirão de base às propostas detalhadas no próximo capítulo deste trabalho, abrindo caminho para as falhas na literatura existente em que esta dissertação pretende contribuir, com uma solução para parte do problema.

3. Rejuvenescimento de Aplicações

3.1. Introdução

Tem-se verificado uma evolução de diversos paradigmas relativos à Engenharia de Software que permitem acompanhar as necessidades cada vez mais exigentes dos sistemas de informação das organizações. A capacidade que actualmente os sistemas de informação possuem para se adaptarem às constantes necessidades das organizações, quer ao nível de processos, das regras de negócio, e ao nível da informação, determinam a longevidade e evolutibilidade do sistema. Esta característica de adaptabilidade dos sistemas às necessidades de negócio está fortemente condicionada pela base tecnológica em que estes estão suportados.

As características tidas como necessárias a um código saudável são as comunmente referidas, como: (1) o grau de reutilização do código; (2) a não duplicação do código; (3) testabilidade do código. Estas características são fundamentais para que o produto esteja preparado para rápidas alterações provocadas por modificações nos processos de negócio. Os conceitos das linguagens OO vão de encontro a estes requisitos,

sendo, por isso, a base tecnológica de excelência para a implementação de sistemas que pretendem ser ágeis e, assim, usufruírem uma boa longevidade.

No entanto, verifica-se que grande parte dos sistemas implementados no mundo financeiro (banca e seguros), está sobre uma base tecnológica não orientada a objectos. Linguagens como o RPG [IBM 2001, 2002] e o COBOL [IBM 1999] continuam a ter um forte enraizamento nessas organizações. Existem vários factores para que isso aconteça, tais como, o receio à mudança, ou porque simplesmente, os sistemas instalados são fiáveis e vão respondendo às necessidades. Em sistemas de informação críticos, como os das instituições financeiras, estes factores ganham um peso extraordinário, tornando-se um entrave à evolutibilidade do sistema de informação.

Conjugam-se vários factores que entram em colisão quando se pretende a agilização destes sistemas de informação. Sendo estes sistemas muito críticos, são, por natureza, adversos à mudança quando estão implementados e estáveis em produção. No entanto, num forte mercado concorrente, como é o caso do mercado financeiro, em que se verifica constantemente novas necessidades de informação e de constantes mudanças nos processos de negócio, torna-se evidente a necessidade de evoluir. Por outro lado, a base tecnológica dos sistemas financeiros continua a ser predominantemente suportada por linguagens procedimentais como o RPG e o COBOL, que, pela sua natureza, não suportam os princípios descritos como necessários para uma fácil evolutibilidade dos sistemas.

Neste capítulo, pretende-se, mostrar como preparar um sistema legado a adaptá-lo para um modelo OO orientado a serviços, sendo que as principais preocupações desta fase de transição são a necessidade de se manter a fiabilidade no sistema e garantir que as especificidades e as regras implementadas não se alterem, isto é, pretende-se que o modelo de negócio se mantenha inalterável. A forma mais segura de aplicar estas modificações é através de uma intervenção gradual no sistema. Devem-se evitar alterações em grande escala que podem levar a uma perda de confiança no sistema

previamente instalado e que, apesar dos condicionalismos, possuía um elevado grau de fiabilidade.

As Motivações de Intervenção no Sistema

A forma mais segura de intervir é com pequenas modificações, de forma a manter o comportamento existente e alterar gradualmente o código para a nova base tecnológica. Deve-se, então, aproveitar pedidos de novas funcionalidades que surjam para intervir, refazendo o código de acordo com as novas especificações. Aproveita-se, assim, a necessidade de desenvolvimento de novas funcionalidades para, ao mesmo tempo, aplicar as alterações necessárias à estrutura do sistema de uma forma gradual. O desenvolvimento segundo este método torna-se mais complexo nesta fase, e o tempo investido também é consideravelmente maior, do que realizar o novo desenvolvimento sem a preocupação de modificar a estrutura do código. No entanto, este é um custo que se pode ver como um investimento inicial, com futuros proveitos ao facilitar as intervenções que venham a ser necessárias efectuar com futuros desenvolvimentos. Esta intervenção inicial não surge enquadrada no contexto das metodologias ágeis de desenvolvimento, no entanto, esta fase inicial de preparação do sistema, é uma fase transitória de preparação, para possibilitar a adopção do ciclo de vida do software proposto por essas metodologias.

O sistema legado pode ser descrito como um artefacto desenvolvido e mantido para responder a um conjunto de requisitos e objectivos que foram surgindo ao longo do tempo. Esse artefacto está construído sob um conjunto de pressupostos (requisitos) estabelecidos quando desenvolvido, mas quando em uso, esses requisitos são deturpados pelas intenções dos utilizadores, que não coincidem. Isto é consequência das diferentes percepções que os utilizadores possuem do sistema, comparativamente com quem o concebeu e desenvolveu. Ou seja, um sistema legado pode ser o resultado de uma evolução desorganizada do software, ou de uma incompatibilidade tecnológica entre a plataforma existente e as tecnologias emergentes que são necessárias para a evolução do sistema. A evolução desorganizada do software pode resultar em:

(1) *Software desestruturado.* O software está interligado sem obedecer a qualquer padrão preestabelecido. O código não está desenvolvido segundo a perspectiva da reutilização da mesma solução para problemas similares. A utilização de padrões tem duas grandes vantagens no desenvolvimento de software: minimiza o tempo de codificação e simplifica o código tornando-o mais legível e fácil de analisar.
(2) *Ausência de separação entre camadas.* O código desenvolvido não respeita nenhuma organização de arquitectura de software, em que lógica de negócio, interfaces com o utilizador, acessos a bases de dados, etc., aparecem, por vezes, todos no mesmo espaço.
(3) *Código "Spagheti"*. Impossível retirar um componente isoladamente sem que isso afecte outros componentes. O desenvolvimento deve ser realizado com a intenção de modularizar o código por componentes e, dessa forma, evitar dependências e potencializar a reutilização do código.

Em termos futuros, um sistema legado significa incapacidade de gerar valor acrescentado para a organização que o possui, mas com potencial para evoluir e sobreviver em relação aos requisitos que emergem. Isto é, um sistema diz-se legado quando a organização em que está implementado necessita dele para funcionar, mas surgem novos requisitos aos quais esse sistema não consegue responder de uma forma efectiva.

Para um determinado requisito, um artefacto pode ser legado ou não de acordo com os objectivos a que este esteja submetido, ou seja, não se pode determinar estaticamente o estado de um artefacto a este nível. Um artefacto define-se como legado quando existe uma determinada ou um conjunto de determinadas intenções às quais o artefacto não tem capacidade de responder. Estas situações representam a não existência de um alinhamento entre o binómio requisitos/artefacto.

A refabricação surge, então, como uma fase do ciclo de vida destes artefactos, que visa adaptá-los aos novos requisitos emergentes. Alguns argumentam que a refabricação não é mais do que corrigir o que foi mal definido na fase de concepção, análise e

identificação de requisitos. Segundo esta argumentação, um sistema bem concebido deverá estar preparado para prever todas as mutações que venha a estar sujeito no futuro. No entanto, esta intenção é irrealista quando confrontada com a realidade das organizações e das mudanças a que estas estão sujeitas.

A previsão de todos os cenários na fase de análise de requisitos reflecte-se no alongamento do tempo de execução do projecto, representando assim um elevado custo. Mesmo investindo muito tempo na fase de análise, tentando-se prever "todos" os cenários possíveis, surgem sempre requisitos que saem fora desses parâmetros ao longo da vida do sistema. Por outro lado, muitos dos cenários previstos no modelo do sistema nunca chegam, de facto, a acontecer. Ao investir muito tempo na previsão de situações hipotéticas que poderão ocorrer, para além da funcionalidade em concreto que o cliente pretende, o mais provável de acontecer é que, quando disponibilizada essa funcionalidade, não representará o mesmo valor potencial para o cliente que teria se disponibilizada mais cedo. Acabam, assim, por serem disponibilizadas funcionalidades sem o valor acrescentado esperado.

A refabricação surge, então, como uma necessidade de preparar, durante o tempo de vida de uma aplicação, o código para a integração rápida de novos requisitos que venham a surgir. Permite-se, assim, que o desenvolvimento de uma nova funcionalidade seja realizado sem preocupações com outros hipotéticos cenários de utilização, para além da necessidade concreta pretendida e solicitada pelo cliente. Desta forma, ganha-se tempo na fase de análise e acelera-se o tempo de entrega da funcionalidade requerida, que poderá, assim, representar para o cliente valor acrescentado e vantagem competitiva ao poder usufruir em tempo oportuno de mercado dessa funcionalidade.

Como já referido, o processo de refabricação em grande escala é um processo arriscado e de elevado custo. Redesenhar e rescrever todo um sistema, modificando e aplicando padrões, separando lógica de domínio de acessos à base de dados e interface, alterando formatos de mensagens, etc., é uma operação complexa e que representa um elevado custo e risco. Quando não controladas, essas acções podem originar em

alterações do comportamento do sistema. Para muitos sistemas complexos, a compatibilidade com o sistema existente é muito importante, como, por exemplo, nos sistemas financeiros. Neste tipo de sistemas é necessário ter alguns cuidados especiais, mantendo a compatibilidade com o sistema existente e evitando perdas de funcionalidades com a rescrita total do sistema. Surge, então, a necessidade de evoluir tendo por base os requisitos do sistema legado.

3.2. Metodologias Propostas

O processo de desenvolvimento segundo uma metodologia ágil pressupõe ciclos rápidos de iteração contendo pequenos desenvolvimentos em cada ciclo. Grandes modificações devem ser separadas em pequenas alterações, gerando-se uma nova versão em cada iteração. As alterações incluídas em cada versão podem corresponder, ou não, a novas funcionalidades. Podem ser geradas novas versões que não alteram o comportamento do sistema, contendo apenas melhorias à estrutura do software. No entanto, sempre que se gera uma nova versão há a necessidade de se disponibilizar mecanismos de controlo da qualidade do produto final. Assim, a geração de novas versões deve ser validada através da execução de testes. Para que tal seja possível é necessário definir e construir os testes adequados sempre que uma nova funcionalidade seja adicionada ao sistema. Os testes devem ser definidos com o objectivo de incluir todas as possibilidades de execução e de interacção dessa funcionalidade, abrangendo todos os cenários possíveis, testando inclusive as situações de falha.

Uma adequada especificação de testes permite garantir que as versões geradas estão livres de erros e que, entre cada iteração, as características de funcionalidades implementadas não se alteram. Os testes permitem, assim, assegurar que a aplicação de técnicas de refabricação não provoca efeitos indesejáveis ao comportamento do sistema.

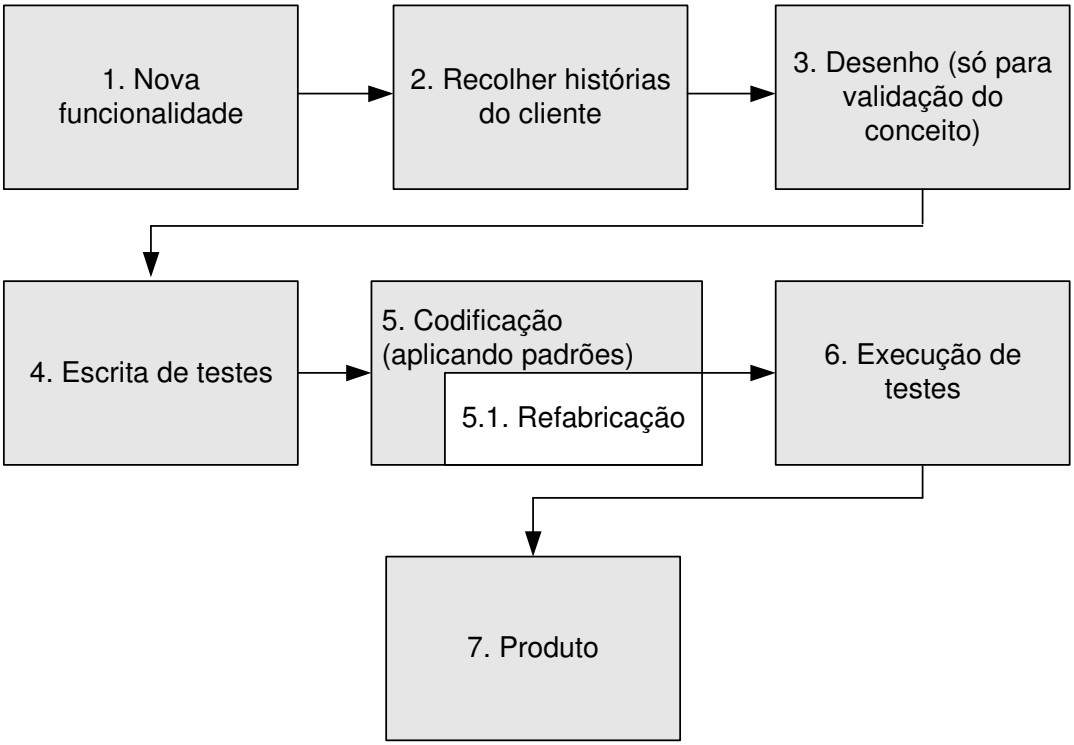


Figura 7 – Metodologia proposta

A metodologia proposta na Figura 7 agrupa-se nas seguintes fases:

(1) *Nova funcionalidade*. Necessidade de implementação de uma nova funcionalidade, ou eventual alteração dos requisitos de uma funcionalidade já implementada. Normalmente é o cliente que identifica essa nova necessidade para o qual o sistema não é capaz de responder, dando origem, assim, a uma nova intervenção no sistema com o objectivo de responder a essa necessidade.

(2) *Recolher histórias do cliente*. Com o intuito de clarificar os requisitos para a nova funcionalidade e que estes sejam compreendidos na usa plenitude pela equipa de desenvolvimento, as metodologias ágeis de desenvolvimento promovem a realização de reuniões entre estas duas partes, equipa de desenvolvimento e cliente, em que o resultado é uma lista de requisitos, acordada por todos, da nova funcionalidade a ser implementada.

(3) *Desenho*. A representação da nova funcionalidade através de notações visuais, como o UML, deve ser utilizada para validação do conceito, ou seja, através dessa representação conseguir uma melhor percepção dos impactos que o desenvolvimento dessa nova funcionalidade possam originar no sistema.

(4) *Escrita de testes*. Antes de iniciar a codificação devem-se especificar os testes, descrevendo o comportamento esperado para a nova funcionalidade. Quando se iniciar a codificação, os testes previamente especificados servem de documentação para o programador, que sabe claramente qual o objectivo da funcionalidade que está a ser codificada.

(5) *Codificação*. O uso de padrões na codificação permite clarificar o código, conferindo-lhe maior legibilidade, além de propor uma solução para problemas comuns. Normalmente, o código produzido nas primeiras iterações do ciclo de desenvolvimento não reflecte estes princípio, nascendo de uma forma desorganizada, verificando-se, por vezes, a necessidade de refabricação do software.

(5.1) *Refabricação*. A refabricação é a fase em que se pretende estruturar o código, conferindo-lhe maior simplicidade, e garantindo a reutilização deste. Quando o objectivo principal no desenvolvimento de uma nova funcionalidade é a disponibilização desta em tempo oportuno para o cliente, pode acontecer que por vezes o código fique mal formado estruturalmente, apesar de funcionar como pretendido.

(6) *Execução de testes*. A execução de testes é uma fase essencial do ciclo de desenvolvimento, pois é o que garante que o comportamento do sistema continua a funcionar de acordo com os requisitos para o quais foi concebido. Num ambiente de desenvolvimento ágil em que actividades como a refabricação são constantes torna-se essencial a existência de conjuntos de testes completos que validem todo e qualquer comportamento do sistema.

(7) *Produto.* Cada ciclo de desenvolvimento termina com a concretização de uma nova versão de software a que se denomina de produto. Cada produto não deve ter uma grande volume de alterações, isto é, não se deve aglomerar um conjunto significativo de alterações. A geração de muitas versões com poucas alterações

permitem fazer facilmente o *rollback* sempre que algo corra mal. Isto porque, a instalação de um produto que contenha um grande volume de alterações, em relação à versão previamente instalada, não facilita a reposição do estado inicial, se em caso de uma eventual falha grave do produto gerado, houver necessidade de o fazer.

Numa perspectiva evolutiva de um sistema legado, dos processos aqui apresentados, as fases de refabricação e de testes são as mais relevantes. A melhoria constante do código, adoptando padrões de desenho através de sucessivas refabricações clarificando e potencializando a sua reutilização, permite a evolutibilidade do sistema. Por outro lado, os testes garantem que as modificações realizadas em cada iteração de desenvolvimento não alteram o comportamento do sistema. Por este motivo, estas duas fases serão tratadas mais em detalhe neste capítulo, e no próximo através da sua concretização no *case study*.

A necessidade de aplicação de técnicas de refabricação sobre o código pode surgir sob a forma de duas necessidades de intervenção distintas:

(1) *Imposição*: necessidade de intervir no sistema, quando se verifica que o software não se encontra convenientemente estruturado, de modo a facilmente acomodar novas funcionalidades (sem modificar o seu comportamento).

(2) *Prevenção*: pela simples análise do código existente, por vezes facilmente se detectam princípios de má estrutura do código ou, simplesmente, má codificação. Essas anomalias provocam, normalmente, dificuldades na reutilização do código. A acção de refabricação, neste caso, tem como objectivo "limpar" o código dessas impurezas, preparando-o, desde logo, para futuras intervenções que se venham a realizar no sistema.

Reconhecer Maus Indícios no Processo (bad smells)

No processo de desenvolvimento de software numa abordagem baseada em metodologias ágeis é possível identificar alguns estigmas que podem indiciar problemas

com o processo de desenvolvimento [Elssamadisy 2002]. As fases do processo de desenvolvimento apresentados na Figura 7, nomeadamente em *5.1 Refabricação*, *6 Execução de* testes, *2 Recolher histórias do cliente*, são passíveis de se identificarem alguns sintomas de que algo não está bem.

A aplicação de refabricações em grande escala é um forte indício de que algo está mal. O princípio de várias iterações com pequenas modificações por ciclo está, em princípio, quebrado quando ocorre a necessidade de aplicação de grandes refabricações. Normalmente, esta situação verifica-se quando os programadores deixam para mais tarde a refabricação do código e dão prioridade à finalização da sua tarefa. Este princípio, segundo as metodologias ágeis, não está de todo incorrecto, no entanto, o problema coloca-se quando o "fazer mais tarde" se atrasa indefinidamente no tempo originando que, as refabricações que deveriam ser realizadas, se vão acumulando até chegar o momento em que não há outra saída senão as aplicar todas de uma vez só.

Já na execução de testes, esse indício verifica-se quando os testes passam, mas o sistema não funciona. A automatização de testes, tal como nos testes unitários, permite resolver este problema. Ao automatizar os testes garante-se a sua execução em cada iteração do processo de desenvolvimento, validando-se, assim, a integridade do sistema.

Finalmente, referindo os maus indícios relacionados com as histórias recolhidas, estes revelam-se quando se perde confiança na relação entre as histórias retractadas pelos clientes e as respectivas partes desenvolvidas para responder a essas necessidades. Esta situação ocorre normalmente em sistemas de elevada complexidade, em que as histórias recolhidas são em elevado número, o que é comum acontecer nos sistemas financeiros, devido aos inúmeros factores que influem essas histórias tais como: decretos de lei, regulamentações internas, configurações específicas, etc. Nestes casos, a solução é documentar as histórias recolhidas e representá-las graficamente (através da análise do sistema) permitindo, assim, ter uma percepção holística de como se relacionam todos os requisitos do sistema (prevista na fase 3 da Figura 7).

Ao descer o nível de abstracção do processo de desenvolvimento para o código, encontram-se os maus indícios que motivam a refabricação do software. Na análise do código para detectar características menos próprias de um sistema que se pretende reutilizável, pode-se seguir um conjunto de orientações que permitem detectar facilmente essas falhas (*bad smells*), tais como [Fowler 1999]:

(1) *Código duplicado*. Situações em que o mesmo código aparece repetido entre métodos da mesma classe ou entre classes distintas.

(2) *Métodos longos*. Quando os métodos contêm muitas linhas de código, normalmente significa que devem ser separados. A solução seria a extracção de partes para métodos mais pequenos

(3) *Classes largas*. Quando se identifica o mesmo problema dos métodos longos nas classes, normalmente é uma indicação de que a classe tem demasiada decisão, devendo-se intervir para delegar responsabilidade noutras classes.

(4) *Muitos parâmetros*. A existência de muitos parâmetros nos métodos significa, eventualmente, que esses parâmetros deveriam estar agrupados num objecto.

(5) *Estruturas "case"*. Quando se detectam estruturas lógicas do tipo "*case*" no código. O mais provável é que essas estruturas devam ser substituídas pelo uso de polimorfismo no código.

(6) *Dependência de informação entre objectos*. Caso se identifiquem métodos que necessitam de muita informação de outro objecto é de considerar a hipótese de criar uma herança entre classes.

(7) *Clusters de dados*. São grupos de dados usados sempre em simultâneo. Normalmente esta constatação significa que esse bloco pode ser uma estrutura própria.

(8) *Herança paralela em hierarquias diferentes*. Nestas situações, alterações numa hierarquia originam alterações na outra.

(9) *Métodos pouco explícitos quanto à sua acção*. Os métodos devem ser claros quanto à sua acção, mesmo que os nomes se tornem muito extensos. Nestas situações, os métodos devem ser renomeados.

(10) *Código duplicado entre classes relacionadas*. Devem ser extraídos os métodos comuns, colocando-os numa superclasse.

(11) *Código duplicado entre classes distintas.* Para o código nesta situação, deve ser criada uma superclasse comum, movendo os métodos comuns para essa classe e, finalmente, deve ser extraída a lógica de cálculo (padrão estratégia).

(12) *Complexidade de condições lógicas*. Solucionar o problema, substituindo a lógica implícita nos cálculos condicionados pelo padrão estratégia.

Esta é só uma pequena lista de conhecidos "maus cheiros" que indicam a necessidade de aplicar refabricação, para corrigir situações de código potencialmente mal concebido relativamente à sua estrutura.

A Necessidade do Modelo

As metodologias de desenvolvimento ágeis não seguem os princípios das metodologias orientadas no modelo. Seguem antes uma filosofia mais agressiva, incentivando a implementação do código logo nas etapas iniciais do ciclo de desenvolvimento. No entanto, e adequando ao *case study* em análise neste trabalho, existem sistemas muito complexos que para se intervir com segurança é necessária uma aproximação inicial ao problema a partir do modelo, e dessa forma, se conseguir capturar com alguma exactidão os seus processos. Isto é, capturar as interacções que as funcionalidades poderão ter com actores intervenientes no sistema e com outros componentes/subsistemas. A partir desse modelo inicial, que fornece um conhecimento mais aprofundado do sistema, pode-se intervir com alguma segurança no sistema legado.

Existindo a necessidade de desenvolver com base nos requisitos existentes e impostos pelo sistema onde se pretende intervir, é necessário analisar o funcionamento desse sistema para extrair as regras existentes. A aplicação da refabricação pressupõe que o comportamento não se altere e para garantir esse princípio deve-se garantir o conhecimento de todas as regras implementadas e documentá-las, usando esse conhecimento para a especificação dos testes que vão garantir a previsibilidade do comportamento do sistema.

Para construir um sistema que corresponda às exigências e necessidades dos clientes e utilizadores, é necessário coleccionar o conjunto de requisitos que descrevem as funcionalidades a serem implementadas no sistema. Essas funcionalidades são executadas pelos utilizadores de acordo com uma sequência e respectiva informação necessária para a sua realização. Para o sistema integrar essa funcionalidade é necessário recolher esses dados, e conhecer o resultado esperado após a sua execução (Figura 8). Este conjunto de informação é, na sua essência, a descrição de um requisito do sistema.

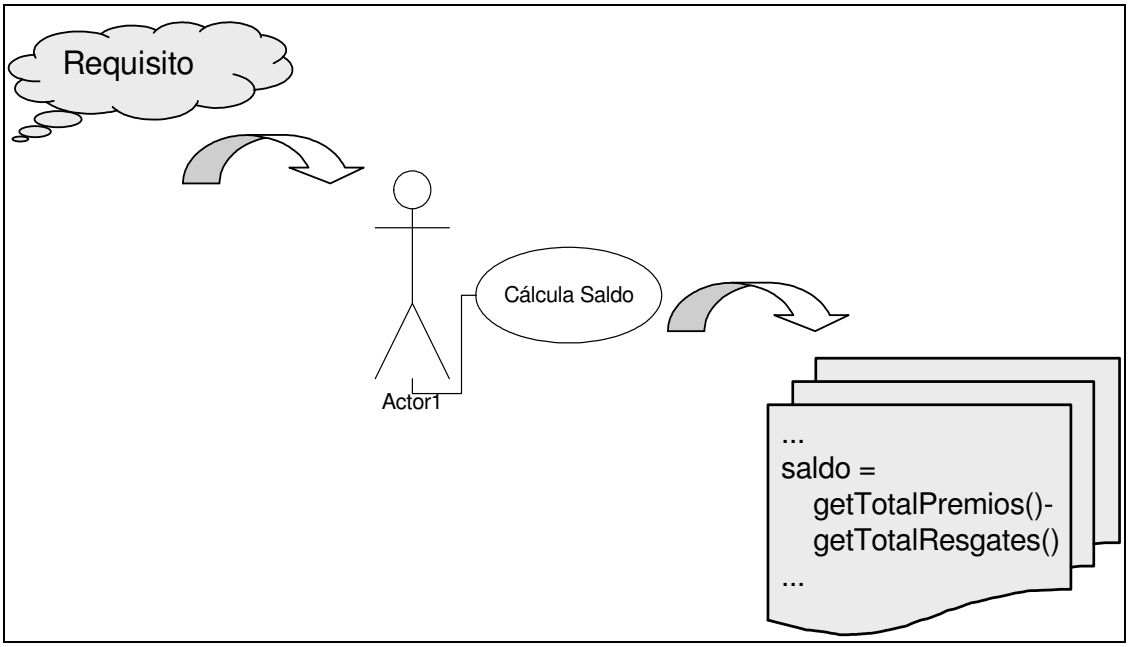


Figura 8 - Do requisito à implementação

No entanto, os requisitos, ao serem fornecidos pelos clientes e utilizadores, são normalmente descritos numa linguagem informal, através de histórias que descrevem o comportamento do sistema. Esta situação leva a que, quando se desça o nível de abstracção para a fase de análise e implementação, não seja possível estabelecer uma correspondência directa entre a descrição do requisito, o modelo e a implementação [Evans 2003]. Com a implementação segundo a abordagem OO, esta correlação fica mais estreita, no entanto, apesar disso, não é possível ter sempre uma relação totalmente fidedigna, dependendo da complexidade do problema em análise.

Ao capturar o modelo, pretende-se representar os intervenientes (actores) e as possíveis interacções com esses intervenientes e outros artefactos. O objectivo é obter uma visão simplificada do comportamento do sistema, descartando os pequenos detalhes. Para auxiliar esta fase de concepção do modelo, recomenda-se o uso do UML[1]. Os cenários nos casos de uso descrevem as comunicações entre o sistemas e as entidades externas.

Após analisados e identificados os requisitos, é possível utilizar os casos de uso resultantes dessa fase de análise como base para o modelo do sistema, com os diagramas de classes e actividades que irão indicar o caminho para a sua implementação. No entanto, ao passar do modelo de negócio, para a concepção da aplicação, com as especificações técnicas relevantes para a fase implementação vai-se perdendo a correlação entre o modelo da aplicação e os requisitos que lhe deram origem [Leffingwell 1999].

Existem, no entanto, contrariedades que as metodologias orientadas ao modelo não conseguem resolver. Na sua maioria, estas metodologias não prevêem a possibilidade de ocorrer uma alteração de requisitos e, por outro lado, verifica-se que, em muitos casos, que o modelo concebido pouco tem a ver com a implementação em concreto.

A modificação dos requisitos representa um problema nas metodologias clássicas de desenvolvimento de software. As alterações de requisitos podem ocorrer por diversos motivos, quer seja por erro na análise inicial, em que a falha é detectada já na sua implementação e testes, ou mesmo por vontade do cliente, que durante o período de implementação pode decidir modificar os requisitos, eventualmente por novas necessidades influenciadas por alterações do mercado. Quando surge uma modificação de requisitos obriga a reiniciar todo o ciclo havendo a necessidade de fazer uma reavaliação

[1] Esta é uma notação que facilita a representação funcional de um sistema com um alto nível de abstracção, através dos casos de uso. Permite representar as interacções de actores (externas) com o sistema sem necessidade de um elevado nível de detalhe.

dos requisitos. Este custo não está só relacionado com o tempo necessário para refazer o processo, mas principalmente com a necessidade de disponibilizar uma funcionalidade dentro da "janela de oportunidade" útil para o cliente, que a caba por se perder.

É, em grande parte, devido a essa desvantagem das metodologias orientadas ao modelo, que as metodologias ágeis aparecem conquistando o seu espaço, com uma abordagem agressiva que privilegia a disponibilização da funcionalidade pretendida pelo cliente, em detrimento da fase de análise de requisitos e da modelação do sistema.

No entanto, com as metodologias orientadas ao modelo surge um outro problema, o alinhamento entre o modelo e a implementação. Existindo a necessidade de direcionar eficientemente o alvo de eventuais alterações a efectuar na aplicação, no âmbito de melhorias ou correcções que se pretendam realizar, torna-se importante que a relação entre o modelo de negócio e o modelo da aplicação seja transparente.

Sendo o modelo obtido pela observação de um sistema e representação dos processos de uma determinada área de negócio, este processo de análise não deve ignorar o modelo da aplicação e os detalhes tecnológicos que a caracterizam. Caso isso se verifique, o mais provável é que, ao se implementar os requisitos e regras de negócio descritas no modelo, não seja possível proceder à modelação da aplicação em concordância com o respectivo modelo de negócio. Muitas vezes, a quebra entre conhecimento e análise acontece quando se desce do nível de abstracção do modelo para o modelo da aplicação e respectiva implementação.

Apesar da modelação do sistema não ser uma parte relevante no ciclo de desenvolvimento das metodologias ágeis, torna-se evidente que quando se pretende evoluir um sistema legado, mais do que conceber um novo sistema, é importante capturar as regras aí implementadas, com o objectivo de: (1) documentar as funcionalidades e requisitos do sistema legado; (2) possibilitar a especificação de testes com base nos requisitos existentes; (3) garantir que as intervenções aplicadas no sistema legado com intenção rescrever código mais simples, não irão modificar nem interferir no seu actual

comportamento. (4) quando surgir a necessidade de implementar uma nova funcionalidade, conhecer os intervenientes e possíveis interacções com os artefactos do sistema.

Mas a principal motivação para que a modelação do sistema não seja excluída do processo de desenvolvimento está na complexidade e criticidade do tipo de sistemas que é a base de estudo neste trabalho, os seguros. Sendo os sistemas de informação financeiros sistemas críticos por natureza e devido à sua extrema complexidade, torna-se evidente a necessidade de documentar a representar os requisitos do sistema para garantir a sua fiabilidade.

Variações de Regras de Negócio em Seguros

Analisando especificamente a área de negócio da banca e seguros, observa-se que é possível existirem diferentes variações de regras de negócio de um processo. Esta variação ocorre por factores políticos, dependendo do país em que a companhia de seguros está sediada, ou até mesmo entre diferentes companhias sediadas num mesmo país que, por motivações de concorrência de mercado, têm necessidade de implementação de diferentes regras. As leis e políticas do país impõem, em grande parte, as regras de negócio que as companhias são obrigadas a implementar, como, por exemplo, ao legislar sobre impostos, ao regulamentar as transferências bancárias ou ao legislar sobre os benefícios fiscais. Estas especificidades intrínsecas à área de negócio em questão conferem uma enorme complexidade ao sistema, originando que o comportamento do mesmo código seja diferente de acordo com a companhia que se está a analisar. Torna-se, por isso necessário em diversas áreas de negócio extrair as regras por companhia e, com base na informação recolhida, especificar os testes adequados.

3.3. Tecnologias de Suporte à Evolutibilidade

Hoje, o interface *web browser* é considerado unanimemente como o interface preferido para a maioria das aplicações, incluindo também as aplicações de uso pessoal. Assumindo-se por força da maioria que é o ponto de entrada para toda e qualquer aplicação, surgiu um problema para as aplicações RPG [IBM 2001] suportadas pelo iSeries [Sundft 2002], que consistiu na necessidade de interagir com outro tipo de interfaces, que não os seus tradicionais écrans verdes. No início da era da "webização" das aplicações, foi referida a hipótese de se rescrever totalmente o código, eliminando, desta forma, o *back office*. Mas rapidamente se percebeu que os antigos programas RPG continham uma importante parte da lógica de negócio das organizações, sendo impossível uma transição imediata entre plataformas distintas. Todos estes acontecimentos tiveram como resultado a evolução do iSeries, levando-o a suportar Java e aplicações web [Coulthard 2003].

As novas aplicações cliente/servidor com interface gráfico têm assim necessidade de usar a mesma lógica de negócio das aplicações iSeries. O J2EE [Sharma 2001] surge como o *framework* que suporta tudo isto. O âmbito de desenvolvimento do J2EE é obviamente o Java, mas não é excluído o uso de outras tecnologias como o RPG e COBOL. Aliás, as aplicações J2EE recorrem aos programas desenvolvidos nessas duas linguagens muitas vezes para a realização de determinados processos de negócio.

O J2EE pode ser separado em 2 áreas: (1) a parte web que consiste em *JavaServer Pages* (JSPs), *Java Sevlets* [Hall 2001], e *Java Beans*; (2) a parte lógica composta por *Enterprise JavaBeans* (EJBs) e *Message-Driven Beans* (MDBs). O J2EE pode ser visto como uma plataforma integradora (*Enterprise Application Integration*- EAI) dos serviços web com o EIS (*Enterprise Information System*) [Sharma 2001]. O serviço web é o cliente que acede aos serviços que, por sua vez, suportam um conjunto de normas e ferramentas comuns que podem ser usadas entre cliente e serviço e, finalmente, o EIS, onde estão implementados os processos de negócio, com as bases de dados e processos

existentes que podem ser acedidos pela camada de serviços [Johnson 2002]. Na Figura 9 está representada esta interacção.

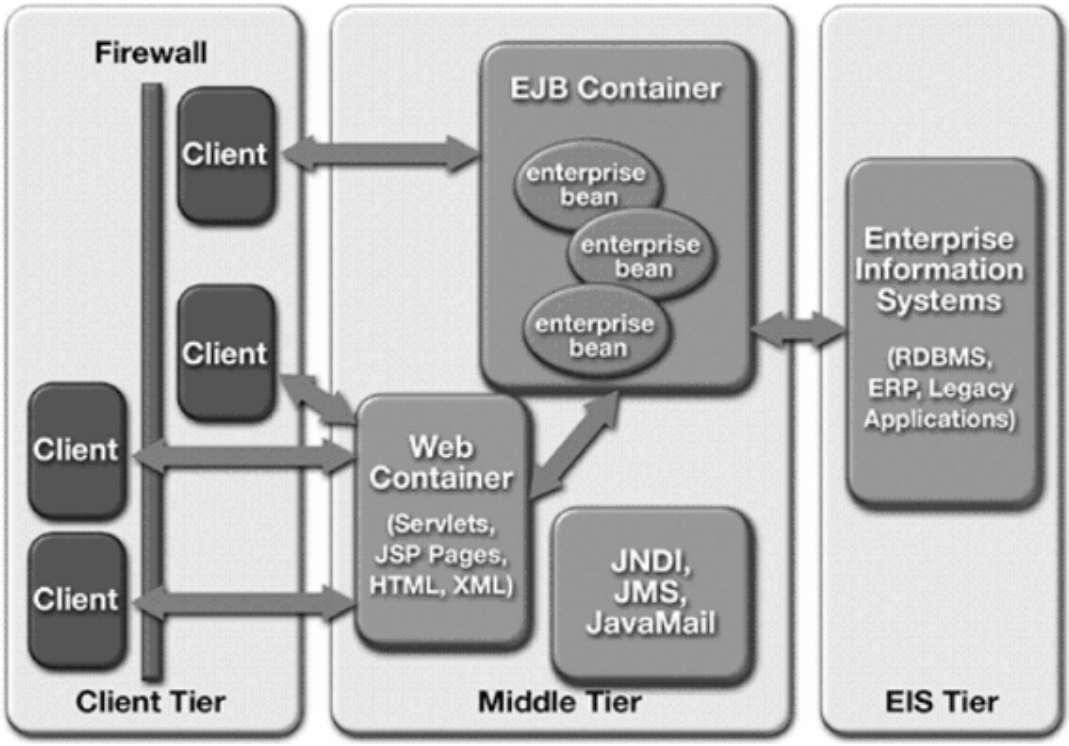


Figura 9 - J2EE, Integrador de aplicações

O repositório de componentes do J2EE é um serviço que fornece a infra-estrutura necessária de suporte para que um componente exista e para que este forneça os seus próprios serviços a um cliente. Os componentes que se podem incluir nesse repositório são:

(1) *Aplicações Java*: são as aplicações genéricas que se encontram dentro do repositório.

(2) *Applets*: são suportadas via *web browser*, e também correm no repositório de aplicações.

(3) *Servlets e JSPs*: fornecem os mecanismos para geração automática de conteúdos.

(4) *Enterprise Java Beans (EJB)*: consistem em componentes de negócio que correm no repositório de EJB. Os componentes EJB podem ser de dois tipos: sessão e

entidade. Os de sessão são têm como função o processamento e gestão da sessão. Este tipo de *beans* pode ter ou não estado. Os que têm estado guardam o estado do cliente entre invocações. Quando não há necessidade de guardar o estado do cliente, então sugere-se a utilização de sessões sem estado, que têm um desempenho melhor que os anteriores. Os *beans* de entidades são usados quando um componente de negócio deve existir e ser partilhado entre múltiplos utilizadores.

A plataforma J2EE fornece também um conjunto de serviços *standard* que podem ser acedidos pelos componentes. Esses serviços são:

(1) *HTTP e HTTPS*: protocolo *standard* para comunicações web, com ou sem segurança.
(2) *JDBC*: é uma API *standard* de acesso à base de dados.
(3) *JavaMail*: consiste numa API para construção de *mails* e mensagens entre aplicações em Java.
(4) *RMI (Remote Methos Invocation)*: é um protocolo que permite a invocação remota de métodos (gerindo a invocação de métodos entre diferentes máquinas).
(5) *JMS*: é uma API para gestão de mensagens entre sistemas, geridos pedidos e subscrições de mensagens.
(6) *JNDI*: é utilizado para registar e procurar componentes de negócio num ambiente J2EE.

O J2EE na sua essência é uma plataforma que consiste na especificação de um padrão para a arquitectura de aplicações, suportado por ferramentas e protocolos. Ao nível de aplicação existem outros padrões que podem ser adoptados para melhorar a estrutura do código [Alur 2002]. De acordo com o nível em observação do sistema, torna-se mais ou menos adequado o uso de determinados padrões.

Aplicação de Padrões

Quando está presente a intenção de reutilizar módulos críticos existentes num sistema legado, é necessário garantir o cuidado e rigor nas intervenções aí realizadas. O único objectivo deve ser a disponibilização do modelo de negócio para outras aplicações, evitando qualquer outro tipo de alteração que possa pôr em causa o correcto funcionamento desses módulos. Este tipo de intervenção acaba por ser repetitivo e padronizado.

No caso concreto do *case study* abordado nesta dissertação, a intervenção no módulo de fiscalidade é uma tipificação do *modus operandis* de outras acções com o mesmo cariz que têm sido realizadas no GIS. O conhecimento assimilado no conjunto dessas acções permite identificar um conjunto de padrões que se evidenciam pela recorrência com que são utilizados.

A intervenção tipificada no módulo "fiscalidade" tem por objectivo a concepção de uma nova lógica de negócio (no caso, caracterizada por uma nova estratégia de cálculo de imposto), mas que necessita de componentes ainda residentes em RPG para realizar todo o processo necessário à obtenção do resultado final (neste caso, é o valor de imposto apurado para um determinado resgate).

Esta intervenção "tipo" permite constatar quais os padrões mais relevantes (mais recorrentemente utilizados) para a estruturação do código, permitindo a reutilização de componentes do EIS na sua interacção com os EJBs. Assim, identifica-se, de entre todos os padrões utilizados, a existência de um grupo de padrões que estão nessas condições (de frequente utilização), realçando-se os padrões como o padrão estratégia, comando, *façade* e *data mapper*.

Os padrões devem ser adoptados de acordo com a natureza do problema que se pretende solucionar. Estes (os padrões) são descrições genéricas para problemas específicos identificados no modelo da aplicação que combinam um conjunto de elementos (classes e objectos) para solucionar esses problemas.

No caso descrito com o *case study* apresentado, com a implementação de novas regras de cálculo de fiscalidade, o padrão *strategy* é o que melhor descreve a solução para o problema de cálculo em concreto. Este permite definir dinamicamente as relações necessárias entre os objectos para apurar um determinado valor de imposto, identificando-se, em tempo de execução, qual a regra que deve ser aplicada. Por outro lado existe também a necessidade de aceder à base de dados existente do sistema existente. Neste caso, verifica-se a aplicação de padrões para mapeamento, acesso e transacção de dados entre aplicação e base de dados. Os outros padrões (*façade* e *command*) permitem esconder a complexidade de negócio na interacção entre serviço e cliente.

O Padrão Façade

Ao modificar um sistema legado com o objectivo de inserir um novo comportamento sem influenciar o existente, torna-se necessário estruturar o código de forma a permitir que isso seja possível. Considere-se que um determinado processo do sistema legado escrito em RPG que executa um conjunto de acções para a realização de uma tarefa, a intervenção visa acrescentar uma nova acção imperceptível ao actual processo. O padrão *façade* esconde a complexidade das interacções entre os objectos de negócio para o cliente que faz um pedido [Gamma 1994]. Isto é, trata-se da implementação de uma fina camada de software que faz a interface entre um pedido externo e as operações que são necessárias realizar para satisfazer esse pedido. Este padrão ganha particular interesse quando aplicado em problemas que têm como necessidade a reutilização de componentes em sistemas legados. A sua utilização possibilita a extensão do código legado para a sua reutilização noutros contextos, fora do processo actual.

Aplicando este conceito ao *case study*, em que o cliente é o processo de criação de um recibo que, estando implementado em RPG, necessita de obter o valor do imposto usando um novo serviço (em Java). Fica, assim, o método de cálculo isolado por uma fina camada no programa RPG que lhe permitirá assumir a nova acção de uma forma "silenciosa".

Na Figura 10 esquematiza-se a interacção entres os processos que compõe a criação de um recibo de indemnização. Em *Calcular Imposto (P3)*, está a utilização de um *façade*, cuja função é esconder a complexidade para o cálculo da parcela do recibo relativa ao valor do imposto. A sua utilização permite que os actuais processos ignorem o facto de existir uma nova regra de cálculo. Estes continuam a pedir o valor de imposto através de *P3* que, por sua vez, irá usar o método de cálculo apropriado para, em cada caso, obter o valor de imposto.

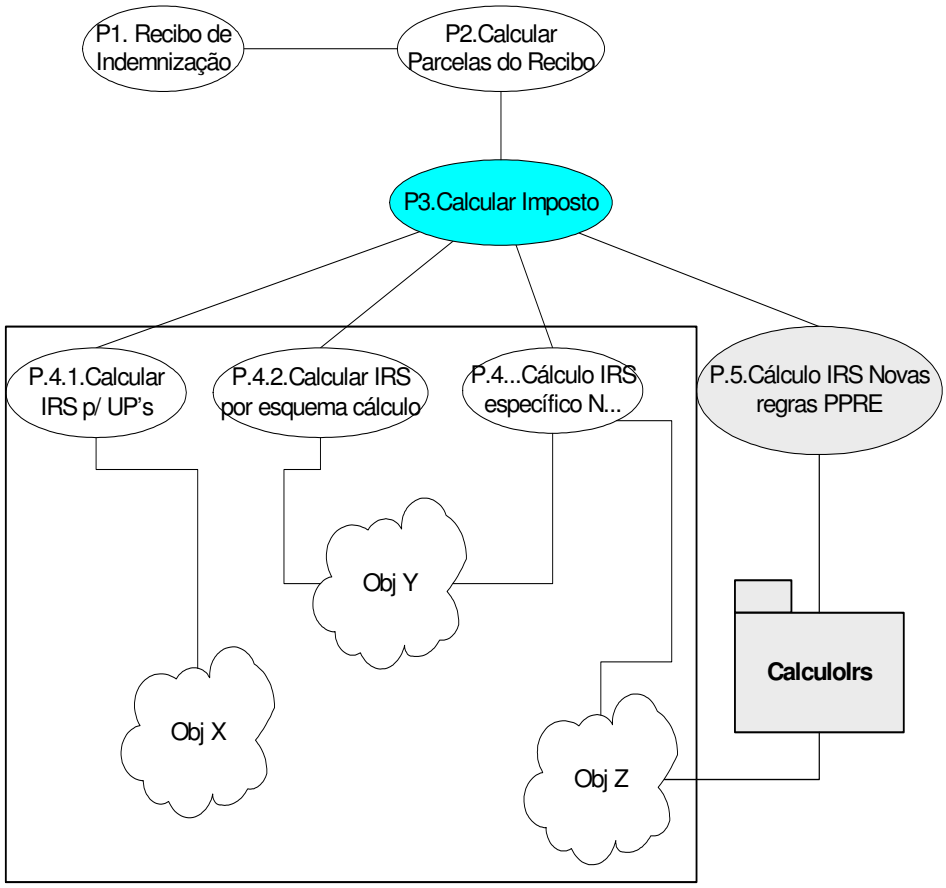


Figura 10 - Padrão *façade*

Ao combinarem-se tecnologias diferentes como o RPG e o Java, torna-se necessário definir protocolos de comunicação para o envio de mensagens entre programas escritos nestas diferentes linguagens. Para isso, desenvolveu-se um conjunto de APIs genéricas de comunicação para permitir a invocação de Java através de um programa RPG. Este padrão é também uma instanciação do *façade*. O programa RPG não conhece a complexidade pelo módulo de comunicação (YASP), limitando-se a enviar um pedido para esse serviço e a receber a mensagem com a resposta a esse pedido. O padrão de comunicação aplicado a este caso, consiste em: (1) iniciar o serviço Java; (2) enviar uma mensagem com o pedido pretendido para ser processado; (3) receber uma mensagem com o resultado da acção; (4) finalizar o serviço.

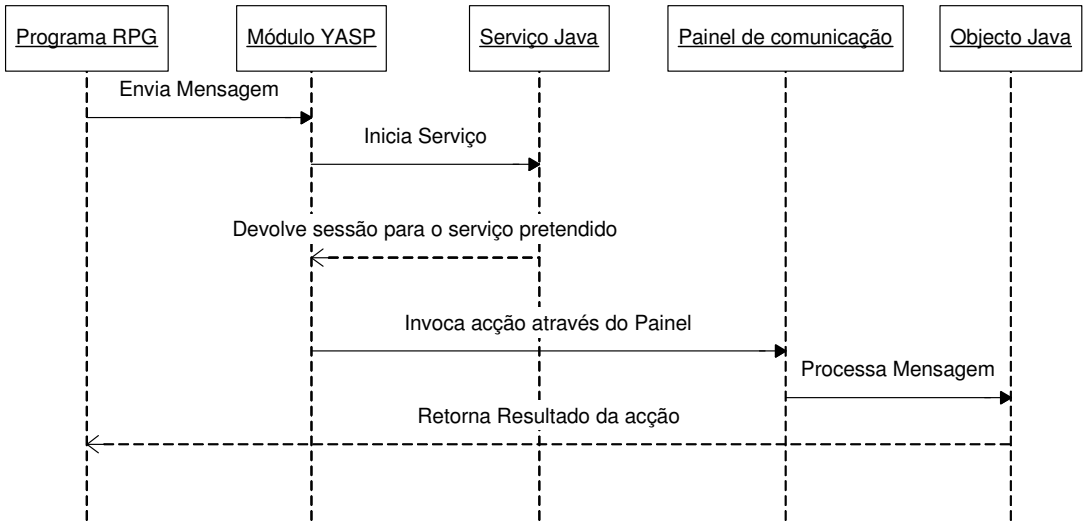


Figura 11 - Diagrama de sequência RPG - Java

Por outro lado, a comunicação em sentido inverso, ou seja, quando um processo Java pretende utilizar um programa RPG, o padrão é diferente. Neste caso, o Java invoca um programa RPG com um conjunto de parâmetros *standard*, esse programa RPG é chamado de *slave*, em que a única tarefa é instanciar um conjunto de parâmetros recebidos e invocar o programa pretendido.

O Padrão Command

O padrão *command* (Figura 12) encapsula um pedido como um objecto, permitindo parametrizar clientes com diferentes pedidos. No caso da fiscalidade, existe a necessidade de saber o valor do saldo da apólice que está a ser resgatada. No entanto, o componente que faz esse cálculo é um componente RPG, pelo que é usada a classe *AS400Slave* que encapsula todos os pedidos de invocação de processos RPG.

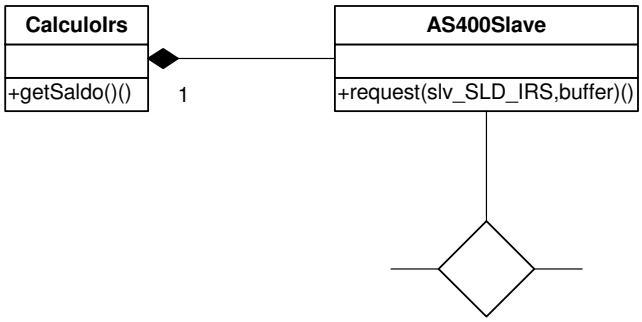


Figura 12 - Padrão *command*

Padrões de Acesso à Base de Dados

Independentemente da existência, ou não, de um sistema legado, a base de dados de um qualquer sistema poderá ser relacional, sendo que, no exemplo do *case study* exposto neste trabalho, é de facto essa a realidade. É , assim, necessário "traduzir" dados de tabelas em objectos e, para isso, existem diversos padrões que sugerem como fazer esse mapeamento [Fowler 2002].

Os objectos de relação e os atributos que os compõem normalmente correspondem aos campos que compõem as tabelas da base de dados. Sendo que os métodos que compõe estes objectos são repetitivos, tendo basicamente os atributos das tabelas, os respectivos métodos *get* e *set* e os métodos de pesquisa e acesso à base de dados. Toda esta estrutura se repete para qualquer objecto de relação com a base de dados. O padrão *Metadata Mapping* sugere que se especifiquem os mapeamentos entre atributos e campos da tabela (por exemplo, através de XML) e se gere o código genérico através dessa informação, como inserções, alterações, selecções e pesquisas à base de dados.

Ainda sobre os padrões de acesso a bases de dados relacionais, tipicamente coloca-se o problema da degradação de desempenho do sistema com o acesso às tabelas dessas bases de dados. O *lazy load* (Figura 13) é um padrão que pode optimizar o desempenho do sistema. Este padrão consiste em retardar a obtenção da informação até ao momento em que esta for necessária. Assim, evita-se tempo gasto desnecessariamente em ler dados que eventualmente nunca serão usados.

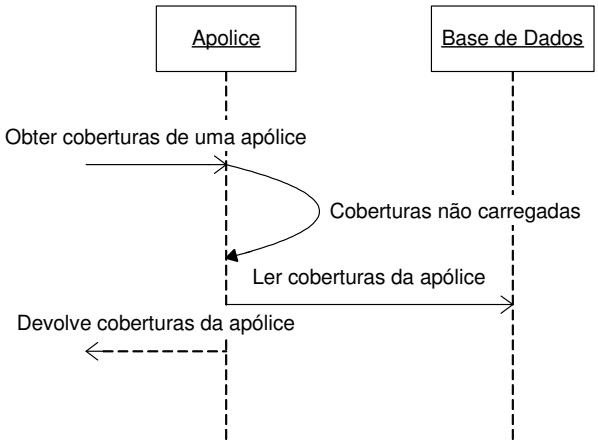


Figura 13 - Padrão *lazy load*

O Padrão Estratégia

A implementação de um qualquer sistema de cálculo, como o exemplo do cálculo de imposto aplicado a produtos de seguros financeiros, assume invariavelmente diferentes formas de cálculo, de acordo com as especificidades dos produtos, clientes, países e respectivas políticas. O código implementado para solucionar este tipo de problemas fica normalmente muito complexo e, consequentemente, difícil de reutilizar e modificar. A solução passa, então, por extrair os diferentes métodos de cálculo de imposto em diferentes objectos, delegando em cada um a execução de cálculos simples sem condições complexas. O contexto de execução de um pedido de cálculo irá determinar qual o método a usar e, assim, em tempo de execução, será utilizado o método apropriado (Figura 14).

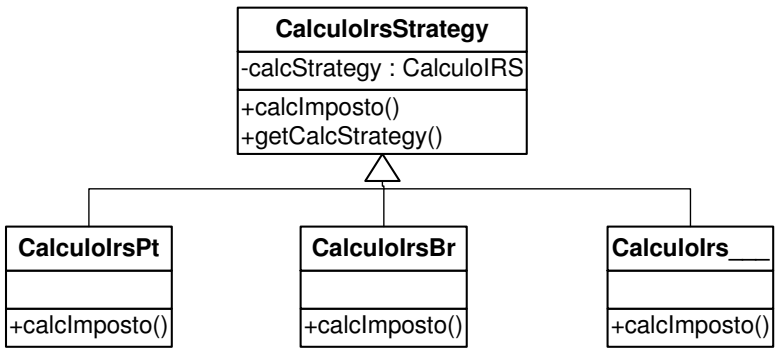


Figura 14 - Diagrama de classes para o padrão estratégia

3.4. Implementação de Testes

Os tipos de testes tipicamente adoptados em *extreme programming* são baseados na simplicidade e devem garantir unicamente os propósitos funcionais do componente para que são desenvolvidos. No entanto, há a necessidade de validar possíveis combinações de acções que estejam relacionadas na execução de uma funcionalidade. Exemplificando com o caso das coberturas de um contrato de seguro, verifica-se a existência de acções que são interdependentes. Desta forma, para se conseguir a validação do sistema torna-se necessário ter uma visão global do comportamento desse componente.

O caso das coberturas das apólices é um bom exemplo deste problema, porque possui dados cuja validação depende de outros dados. Com a metodologia XP propõe-se a validação por unidade, mas não se sugere qualquer solução para resolver questões desta natureza. As coberturas da apólice possuem os seguintes atributos que são interdependentes: duração, data início, data fim e duração de pagamento. Este problema pode ser solucionado com o recurso ao UML para a especificação de testes. As histórias dos clientes (*user stories*) ao serem representados em casos de uso e diagramas de sequência, iriam permitir identificar sequências de acções e dependências funcionais entre os componentes da aplicação.

Existem algumas propostas de implementação de casos de teste com recurso a UML [Labiche 2001; Fraikin 2002]. A metodologia que se propõe para a resolução do problema aqui tratado é baseada na proposta de *Labiche*:

(1) Definir os casos de uso que mostram o modo de relacionamento entre funcionalidades e actores do sistema.

(2) Especificar sequências de acção através de digramas de sequência. Assim, consegue-se identificar dependências entre acções e que devem ser garantidas na especificação dos testes.

(3) Representar diagramas de actividades sob a forma de grafo onde se mostra como as actividades se relacionam, com base na sequência da acção representada nos diagramas de sequência.
(4) Especificação formal do diagrama de actividades, através do qual se pode identificar os cenários de teste, identificando-se também as sequências possíveis de acção.
(5) Definir a situação inicial dos objectos, com os dados que os irão instanciar.
(6) Definir as condições finais com as quais os resultados finais da execução dos testes devem ser comparados.
(7) Implementar os testes com recurso a uma ferramenta que permita automatizar a execução de testes, de forma a que seja possível executá-los em cada iteração do ciclo de desenvolvimento XP, ou seja, em cada versão final de um produto.

Diagrama de Actividades

Para a identificação de todas as possibilidades de teste relativas a um determinado componente, deve-se descrever todas as interacções possíveis de serem processadas por esse componente na execução de uma determinada acção. A descrição dessas interacções pode ser auxiliada graficamente com a representação através de um diagrama de sequências, ou diagrama de actividades ou até por um grafo de actividades. A escolha dos diagramas a serem utilizados é determinada pelo tipo de problema que está a ser analisado.

No exemplo da Figura 15 pretende-se com o grafo identificar as actividades que agem sobre os atributos: data início, data fim, duração e duração de pagamento no processo de coberturas da apólice.

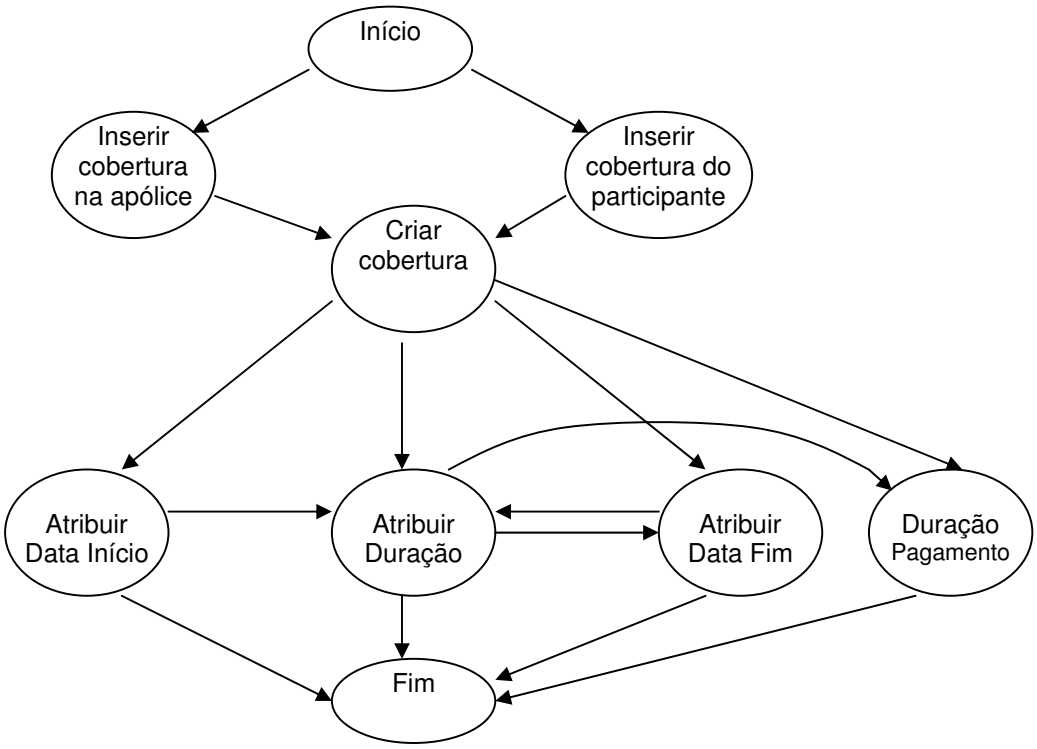


Figura 15 - Grafo de operações nas coberturas

Especificação Formal do Grafo

Com base no grafo anterior, pode-se representar a sequência de actividades através de uma especificação formal. Esta forma depois de derivada permite identificar todos os cenários possíveis de acção (Figura 16).

(Apolice$_{CreateCbt}$ || Participante$_{CreateCbt}$) . Cobertura$_{Create}$. ((Cobertura$_{SetDataIni}$) || (Cobertura$_{SetDataIni}$. Cobertura$_{SetDuracao}$) || (Cobertura$_{SetDataFim}$) || (Cobertura$_{SetDataFim}$. Cobertura$_{SetDuracao}$) || (Cobertura$_{SetDuracao}$) || (Cobertura$_{SetDuracao}$. Cobertura$_{SetDataFim}$) || (Cobertura$_{SetDuracao}$. Cobertura$_{SetDataFim}$. Cobertura$_{SetDuracaoPagamento}$) || (Cobertura$_{SetDuracaoPagamento}$))
Notação usada: Ex: Cobertura$_{SetDuracao}$, em que: Cobertura -classe $_{SetDuracao}$ -método . -Ligação lógica dependente entre objectos/métodos || -Ligação lógica opcional entre objectos/métodos

Figura 16 - Derivação de grafo em cenários possíveis

A implementação aplicada neste exemplo é sobre o ramo da árvore (grafo) que deriva do ramo participantes da apólice. Após se terem identificado as possíveis sequências de acção, o próximo passo seria a definição dos testes onde essas seriam aplicadas. No próximo capítulo é usado o *case study* para a aplicação desta metodologia de especificação de testes.

O desenvolvimento baseado em XP tem como requisito os testes de unidade [Beck 1998] com forte ênfase na implementação e especificação de testes na fase inicial do ciclo de desenvolvimento. O XP além de ter uma orientação para uma implementação prévia dos casos de testes, indica também que esses casos de teste devem ser escritos de forma a que possibilitem a sua automatização.

Os autores do *Junit* são *Kent Beck*, e *Erich Gamma* [Beck 1999]. Estando estes autores também na origem da metodologia XP, torna-se evidente que esta ferramenta permitiu em grande parte suportar essa metodologia de desenvolvimento.

Junit [Junit 2003] é um *framework* que facilita o desenvolvimento e execução de testes de unidade em código Java [Beck 1998]. É uma API para construção de testes, em que as classes *Test*, *TestCase*, *TestSuit*, oferecem a infra-estrutura necessária para a criação de testes. Os métodos *assertTrue(), assertEquals(), fail()*, etc., são usados para testar os resultados. A aplicação *TestRunner* executa testes individuais e *suites* de testes apresentando os resultados de sucesso ou falha de execução (Figura 17).

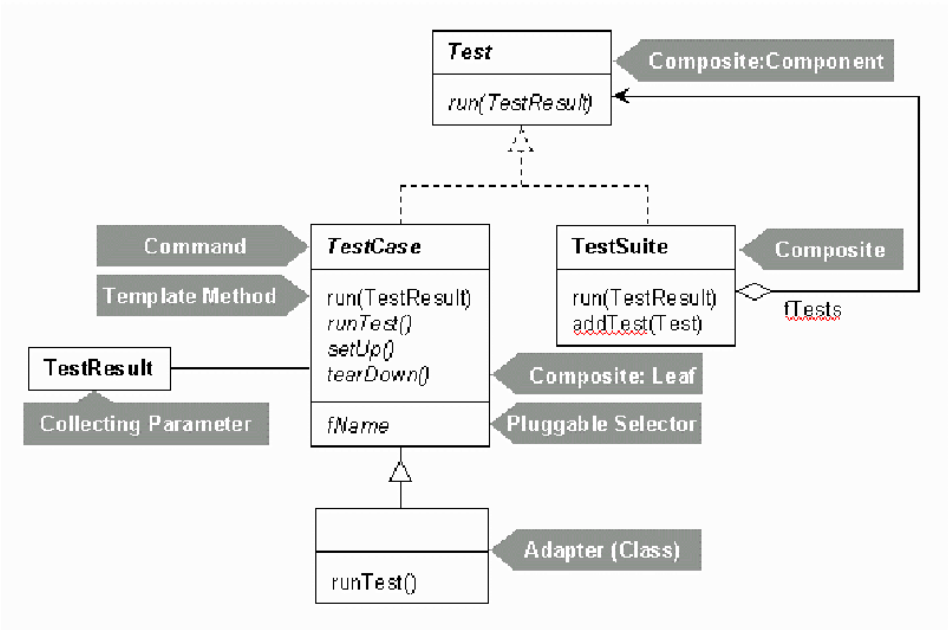


Figura 17 - Framework *Junit*

Como ferramenta para a execução de testes, esta potencializa a encoraja a implementação de testes porque [Beck 1998]:

(1) *Facilita a criação e execução automática de testes*. É fácil para o programador escrever e executar um teste, encorajando-os, assim, a escrever todos os testes necessários à respectiva classe.

(2) *Boa apresentação dos resultados*. Os resultados de teste são apresentados de forma a que se compreenda facilmente como correu a sua execução. O *Junit*

apresenta os testes que sucederam bem, e os que tiveram erros, mostrando a causa do erro em caso de falha.

(3) *Testes de unidade*. A finalidade do *Junit* é a execução de testes sobre um componente ou classe. Um *test case*, deve incidir apenas sobre uma classe.

(4) *Diversos tipos de teste*. É tão importante testar o código para situação válidas como para situações de erro (por exemplo, através do método *fail()*). O sistema deve estar preparado para responder em situações de erro e por isso é importante sujeitá-lo a erros, verificando assim se são detectados.

A restruturação de código para um modelo OO, sendo uma modificação que torna o código mais legível, melhor reutilizável e mais ágil, provoca, também, que o sistema fique menos eficiente em relação ao seu desempenho. O código nativo, sendo de mais baixo nível, é também mais eficiente, e quando o sistema de informação em causa é de uma companhia de seguros, onde a quantidade de informação de circula nas bases de dados é de grandes proporções, este factor ganha uma importância especial. É importante, assim, incluir no circuito de teste, os testes de desempenho, de forma a avaliar e detectar partes de código menos eficientes e melhorá-las.

3.5. Conclusão

A coabitação de duas plataformas distintas, suportadas por linguagens muito diferentes, uma procedimental (RPG), a outra orientada a objectos (Java), levanta novos problemas, cujas soluções se devem sistematizar. O contexto retractado neste trabalho é de elevada complexidade, pois, além da migração de tecnologia, há a necessidade de preservar componentes essenciais que estão em código RPG, mas que, dada a sua criticidade dentro de um sistema financeiro (como o GIS), é conveniente proceder a uma migração gradual desses componentes.

Ajustar os problemas e equacioná-los à luz de soluções descritas sob a forma de padrões, é uma boa forma para começar um processo de sistematização de soluções para problemas que surgem da necessidade de migração de código, dentro dos

constrangimentos já referidos. Neste capítulo, foram propostas metodologias que permitem sistematizar a resolução dos problemas dentro do contexto exemplificado pela realidade do GIS. No âmbito das metodologias ágeis de desenvolvimento, a solução sugerida consiste na necessidade de restruturar o código existente com auxílio de padrões e dotar o novo código de conjuntos de testes que garantam a correcta implementação de novas funcionalidades.

Os padrões são importantes no sentido de fornecerem boas soluções para problemas típicos. Desta forma, evita-se que se esteja constantemente à procura de soluções para os mesmos problemas. Por outro lado, uma boa especificação de testes, garantindo a qualidade do código implementado é igualmente importante. Uma boa especificação de testes deve prever todas as combinações possíveis na execução de uma acção, testando, não só, as situações válidas, mas também as situações de excepção, devendo o sistema detectar e reportar eventuais erros.

As propostas sugeridas são concretizadas no próximo capítulo, onde se pretende demonstrar a aplicabilidade dos conceitos facilitadores do rejuvenescimento de aplicações a um caso de software de seguros.

4. Experimentação com Software de Seguros

4.1. Introdução

Este trabalho pretende abordar o uso das técnicas propostas pelas metodologias ágeis de desenvolvimento, nomeadamente pelo *eXtreme Programming*, aplicadas à restruturação dos sistemas legados.

As técnicas alvo são a refabricação e os padrões de desenho como motor para o rejuvenescimento de aplicações. Para fundamentar estes propósitos recorre-se, nesta dissertação, à implementação de um novo componente para o cálculo de imposto, que é aplicado a um produto específico de seguros Vida, os Planos Poupança Reforma e Educação (PPR/E). Este *case study* possui características especiais que permite instigar e validar as vantagens da refabricação de software como meio para o rejuvenescimento de aplicações. Assim, entre as características que este caso possui, verifica-se:

(1) Necessidade de disponibilização imediata do componente nas companhias de seguros, pois as novas regras publicadas por decreto de lei entram em vigor após a data de publicação. A capacidade, ou não, de responder a esta necessidade permite validar

efectivamente se a abordagem de desenvolvimento, focada na funcionalidade primeiro e na arquitectura depois, trás de facto vantagens na redução do tempo de entrega do produto ao cliente e, assim, garantir a sua disponibilização em tempo útil de mercado.

(2) O sistema onde foi desenvolvido este componente, o GIS (Gestão Integrada de Seguros) da I2S (sistema e empresa que são descritos mais em pormenor nos próximos tópicos deste capítulo), permite caracterizar também outros sistemas desenvolvidos por outras empresas e noutras áreas de negócio. É um sistema com forte implementação no mercado há longos anos, e onde se pretende evoluir de um sistema com um elevado grau de maturidade, mas pouco ágil, para um sistema mais ágil nos processos de desenvolvimento e implementação de novas funcionalidades, potenciando a reutilização de código. Por outro lado, a intervenção no sistema não deve ser drástica, de modo a manter a fiabilidade e a confiança no sistema de informação.

O GIS – Uma Solução para o Mercado de Seguros

O GIS (Gestão Integrada de Seguros) é uma solução para o mercado de seguros, que tem vindo ao longo dos últimos 20 anos a fortalecer a sua presença nesse ramo, abrangendo todas as suas ramificações desde os balcões, mediadores, e as próprias companhias de seguros. Em Portugal esta solução está na maioria das companhias, encontrando-se em expansão para o mercado internacional, nomeadamente, para o Brasil, Polónia, Espanha, Cabo Verde, Macau e mais recentemente Angola. O GIS suporta assim, o circuito integral de um contrato de seguro, desde o balcão que angaria o contrato, até à companhia que o concretiza.

Além de suportar o ciclo de vida do contrato de seguro, o GIS permite que as companhias especifiquem as regras, os circuitos e os seus produtos específicos. Ou seja, o GIS fornece os processos e as regras genéricas de um contrato de seguro e ferramentas de extracção e consulta de informação, permitindo que as companhias configurem e especifiquem as suas particularidades do negócio. Assim, diferentes companhias que utilizam como sistema o GIS podem apresentar diferentes produtos no mercado, de

acordo com as suas apostas e ofertas, através da capacidade de configuração que o GIS oferece.

O GIS apresenta-se no mercado como uma solução capaz de implementar as necessidades do cliente com grande rapidez e com sentido de oportunidade. Permite que as companhias que o implementem tenham a possibilidade de lançar novos produtos sem a necessidade de esperarem por desenvolvimentos específicos que dêem suporte a esses novos produtos de seguros. O grande propósito do GIS define-se, assim, pela necessidade de responder eficazmente às necessidades dos clientes em tempo oportuno de mercado.

Analisar e compreender a evolução desta solução para seguros, que foi inicialmente desenvolvida sob a plataforma do AS400, conseguindo ultrapassar enúmeras barreiras, e vencer durante 20 anos num mercado de forte competitividade, como é o mercado de seguros, constitui uma das motivações desta dissertação. O GIS assume-se, então, como o *case study*, por excelência, para a validação da estratégia de evolutibilidade dos sistemas de informação proposta no capítulo anterior.

Infra-estrutura Tecnológica

Tecnologicamente, o GIS tem evoluído no sentido de se tornar multiplataforma, com a integração de vários canais (web, balcões, *front office*, mediadores, …) e para isso houve a necessidade de criação de serviços que respondessem aos pedidos desses diferentes canais (Figura 18). Esses serviços têm a função de receber um pedido e aceder aos diferentes componentes do GIS para responder à solicitação.

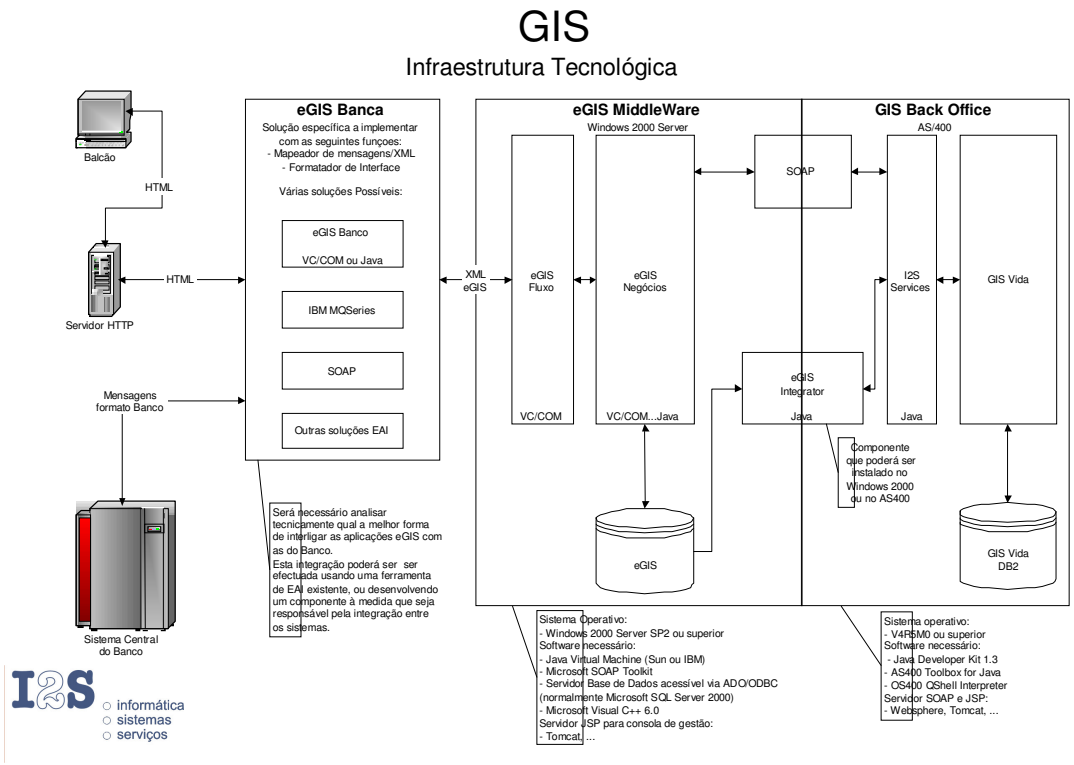


Figura 18 - Arquitectura GIS

Actualmente, o GIS em está plena fase de evolução de um sistema baseado em linguagem procedimental com código escrito principalmente em RPG para código *Object Oriented*. Essa evolução tem como finalidade a concepção de objectos de negócio em Java, que contêm as regras de negócio e permite adequar o seu comportamento às regras definidas por configuração, à semelhança do que acontecia com as aplicações RPG, mas conferindo as vantagens intrínsecas às linguagens OO com a implementação em Java, e sob uma camada de serviços facilitadora da interoperabilidade entre sistemas heterogéneos.

A possibilidade de definir regras através de configuração significa que, os mesmos objectos podem conter comportamentos diferentes, de acordo com as configurações que o definem e moldam. Por exemplo, uma apólice é um só objecto de negócio, independentemente se esta corresponder a um produto de risco, ou a um produto financeiro (renda). Esta característica, entre outras, dá a percepção da complexidade que

o sistema pode assumir e da necessidade do elevado nível de configurabilidade que um sistema deste tipo obriga.

Estas regras de negócio quando implementadas em linguagens procedimentais, como o RPG, perdem agilidade à medida que vai crescendo a complexidade das mesmas. Ficando cada vez mais difícil fazer alterações ao código, perde-se tempo de resposta e o risco de novas alterações introduzirem erros aos processos existentes é cada vez maior. Por estes motivos, a tendência hoje é transpor todas as regras de negócio existentes em RPG para Java.

No entanto, não é possível a migração total do sistema de uma linguagem para a outra, devido, não só, ao elevado número de objectos envolvidos, mas também devido à criticidade de alguns componentes que compõem o GIS. Existem, assim, componentes em RPG que têm todo o interesse em serem reutilizados devido à sua fiabilidade e à elevada importância que esses componentes representam para o sistema. Por serem áreas sensíveis (como por exemplo, o sistema de cálculo), torna-se melindroso efectuar alterações significativas, pelo que, o processo de evolução nestas áreas em concreto deve ser incremental e seguro.

Por isso, apesar desta tendência atrás descrita em modificar o suporte de codificação para uma estrutura *Object Oriented*, continua a haver a necessidade de coabitação do novo código Java e os existentes RPG, C e Cobol.

4.2. Refabricação do Módulo "Fiscalidade"

Após uma breve descrição do GIS, da sua evolução histórica e da sua estrutura tecnológica, apresenta-se agora o caso que constitui o alvo de estudo neste trabalho. Para melhor se compreender o *case study* é necessário localizá-lo no contexto GIS e em que área de negócio se enquadra.

O GIS é composto por diferentes áreas de negócio. Resumidamente, é composto pelo módulo de Vida (GIV), pelo módulo de Reais (seguro do ramo Real) e pelo módulo de e-*business*. O módulo GIV é onde se encontra o objecto específico de estudo neste trabalho, nomeadamente os resgates por processo de sinistro de seguros do ramo vida. Este módulo (GIV) é composto pelas seguintes áreas de negócio:

(1) *Produção*: área da aplicação que trata da criação de novos contratos, anulação e reposição de apólices. A inserção de um contrato pressupõe a introdução de toda a informação relativa ao contrato de seguro, como agentes cobradores, titulares, participantes, pessoas seguras, capital de risco, prazos de contrato, coberturas, etc. Na lógica do GIS, a unidade de produção é também responsável pela criação e emissão de recibos de prémio, e emissão de apólices.

(2) *Cobranças* (Recibos e Indemnizações): módulo que trata da cobrança de recibos de prémio e de indemnização. A gestão de cobrança trata também o tipo de cobrança, inclusive, o processamento de cobranças bancárias.

(3) *Sinistros*: abertura e alteração de processos de sinistro. Criação de recibos de indemnização, e gestão de pagamentos às respectivas entidades recebedoras (beneficiários, segurado, etc.).

(4) *Resseguro*: na gestão de risco da aplicação é necessário controlar o limite de risco suportado pela companhia. Este processo é suportado pelo módulo de resseguro.

(5) *Configuração*: todas estas áreas lógicas são suportadas por um subsistema de configuração. Este subsistema contém as informações para:

- Validação: este tipo de informações contém valores e formatos válidos dos dados introduzidos e/ou calculados.
- Regras de cálculo: especificação dos formulários de cálculo para prémio, comissões, capitais, indemnizações, provisões.
- Formatos de *output* (relatórios): configuração de *outputs* de relatórios e mapas.
- Valores predefinidos: permite a configuração de valores predefinidos, definindo o comportamento base dos processos.

- Eventos de cálculo: especificação de cálculos disparados por eventos (*triggers*), em que, associados a um determinado evento podem configurar um conjunto de acções a serem executadas.
- Produtos (modalidades): toda a informação necessária ao suporte da produção, tal como, tipo de produto (risco/financeiro), fraccionamento, datas limites, validade do produto e coberturas associadas.

A vida de uma apólice desde o seu início até ao fim envolve diferentes actividades, como, o vencimento, criação de recibos, alterações diversas dos dados da apólice (capital, moradas, fraccionamentos,...), cobranças, transferências para outras companhias, abertura de processos de sinistro, e outras.

Em relação à abertura de um processo de sinistro, normalmente, este dá origem à criação de um recibo de indemnização. Esta actividade é algo complexa, nomeadamente no apuramento do valor a resgatar. O valor de resgate é condicionado por um conjunto de variáveis, tais como: (1) valor das entregas de prémios (pagamento de recibos) realizadas durante a vida da apólice; (2) indemnizações pagas; (3) capitalizações dos prémios pagos a diferentes taxas de juros; (4) o valor do imposto (IRS).

No apuramento do cálculo do imposto, o GIS permite que seja o cliente a definir por configuração, qual o método de cálculo a ser utilizado para um determinado produto, de acordo com o tipo de sinistro declarado. O GIS disponibiliza diferentes métodos para o cálculo de imposto, tais como: (1) por esquema de cálculo (fórmula matemática configurada no GIS); (2) programa específico para o cálculo do valor de imposto.

Para a generalidade dos produtos de vida, o cálculo do imposto é processado pelo programa predefinido (*default*), pois as regras são quase sempre idênticas. No entanto, para os produtos PPR/E as regras de cálculo do imposto foram modificadas recentemente por decreto de lei, aumentando a sua complexidade. Assim, é possível num único resgate ter diferentes taxas de imposto de acordo com as características das entregas efectuadas

(recibos pagos), com as informações do tomador de seguro (idade) e com as razões para o pagamento de indemnização (motivo de resgate).

Esta complexidade de regras faz com que os métodos de cálculo de imposto até então suportados pelo GIS não consigam responder a estas novas necessidades. Torna-se então necessário modificar o sistema de cálculo de imposto de forma a incluir as novas regras, mas sem pôr em causa os outros métodos de cálculo que eram suportados e que continuam a ser válidos para os restantes produtos que não PPR/E.

Descrição do Sistema de Resgates

Para se identificarem as alterações que são necessárias ao GIS, na implementação do novo sistema de cálculo, é necessário efectuar uma análise prévia do funcionamento actual do sistema. Esta análise irá permitir identificar os componentes reutilizáveis e as novas funcionalidades a desenvolver.

A realização do resgate é processada através da abertura de um processo de sinistro onde se definem as condições gerais do resgate (beneficiários, tipo de cobrança, razão do sinistro, etc.). Essas condições do sinistro associadas às condições do contrato irão determinar a taxa de imposto a aplicar nesse resgate.

O seguinte diagrama de casos de uso da Figura 19, representa as interacções que ocorrem entre os utilizadores (actores) e os subsistemas abrangidos pela operação de resgate.

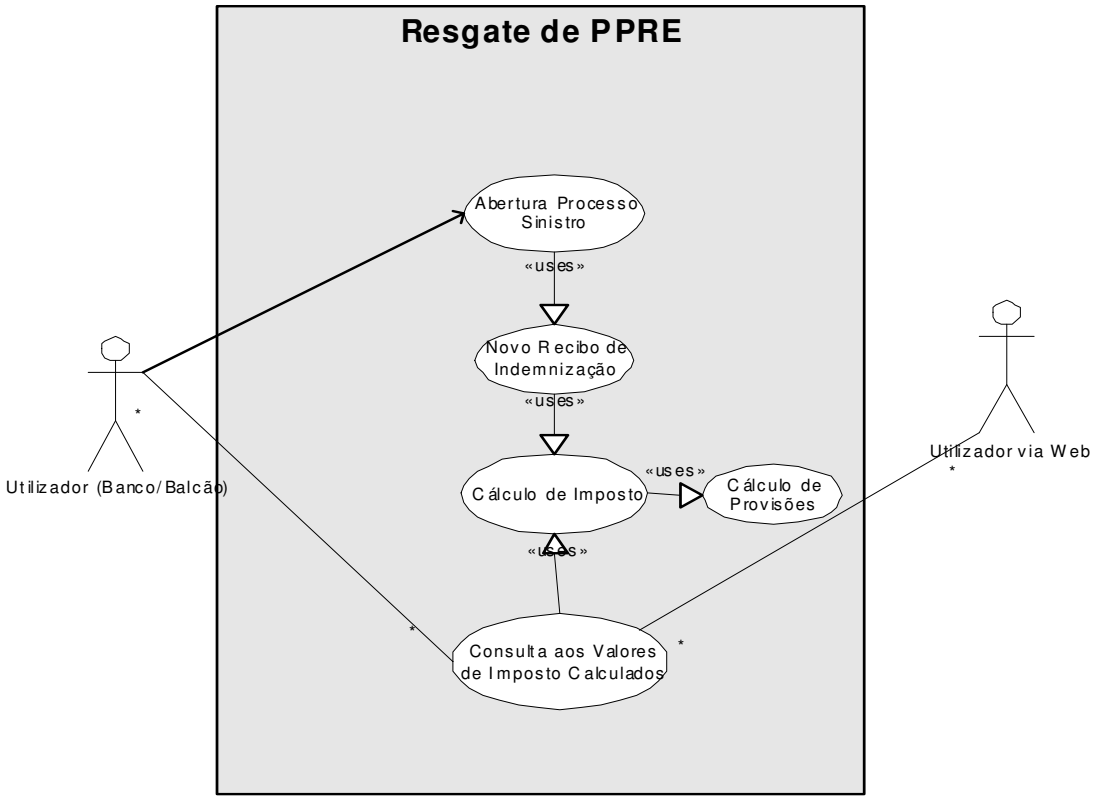


Figura 19 - Resgate de um recibo de indemnização

O caso de uso "Cálculo de Imposto" é o módulo que se pretende intervir, adaptando-o às novas regras publicadas no novo decreto de lei, que legisla sobre as novas fórmulas para o cálculo do IRS neste tipo de produtos.

Histórias do Cliente

Após a identificação das funcionalidades do sistema existente, é necessário analisar os artigos do decreto de lei e traduzir essa informação em requisitos para o novo processo cálculo. Essas regras estão resumidas na tabela anexa constante no Anexo A. Neste caso, as histórias do cliente estão definidas pelo decreto de lei, no entanto, dada a possibilidade de poderem ocorrer diferentes interpretações das leis publicadas, é conveniente reunir com o cliente (companhia de seguros) de forma a garantir que as regras identificadas coincidam com a perspectiva do cliente.

Do quadro resumo com as regras para resgate de PPR/E, constantes no anexo referido, pode-se analisar o impacto que as novas regras representam em relação ao processo existente. Esse impacto incide na possibilidade de, no mesmo recibo de indemnização, serem incluídas diferentes taxas de imposto, ao contrário do que era efectuado anteriormente, onde só era aplicada uma única taxa de imposto num resgate. Esta alteração implica mudanças profundas no modo de apurar os rendimentos e obriga a agrupá-los diferenciadamente de acordo com as taxas a que serão sujeitos. O diagrama de actividades da Figura 20 descreve estas duas formas de cálculo de imposto.

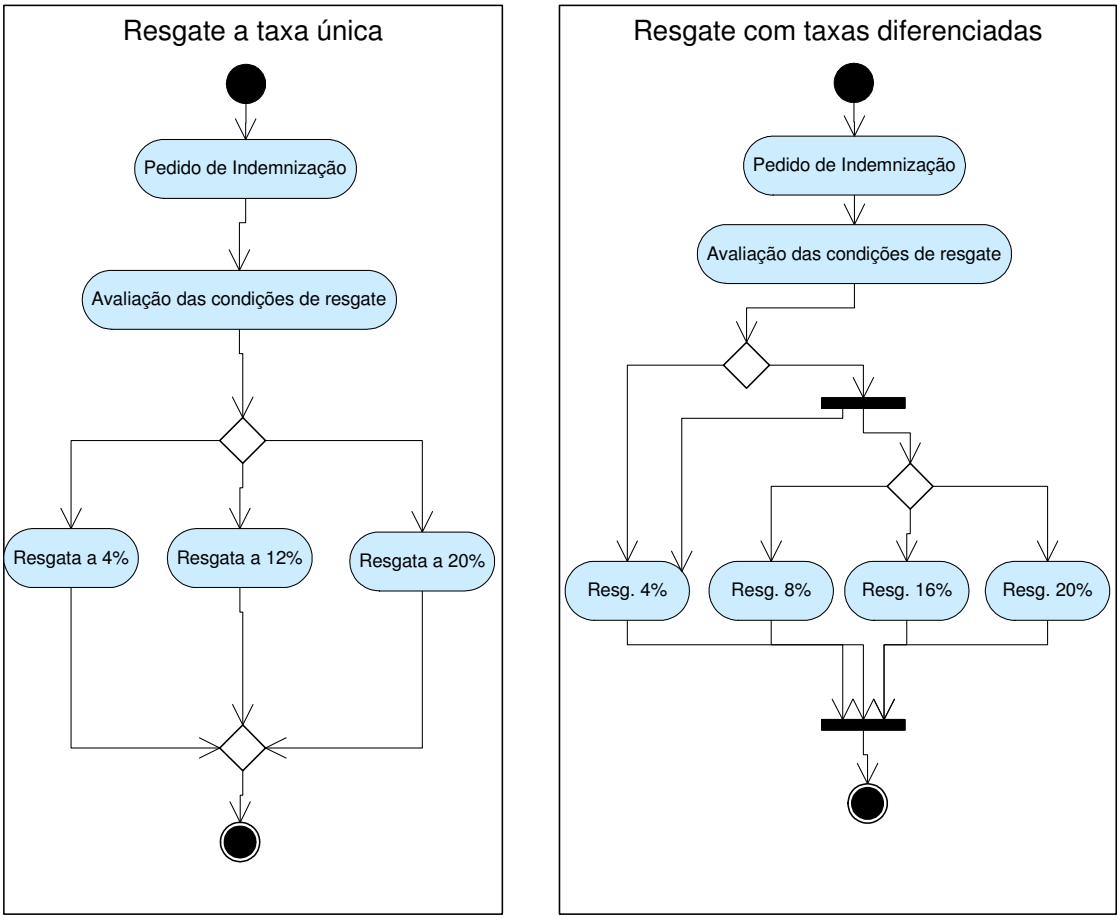


Figura 20 - Comparação das regras de cálculo de IRS

O novo modo de cálculo com taxas diferenciadas no mesmo resgate, para além de implementar uma nova lógica de cálculo significa, também, uma alteração drástica no modo de cálculo de rendimento no GIS. O imposto num recibo de indemnização incidia sobre o valor do rendimento global da apólice. Esse valor de rendimento é apurado ao obter a diferença entre o saldo da apólice na data do resgate e o total das entregas (prémios pagos) realizadas até esse momento. Este era o *modus operandis* do GIS para obter o rendimento. A alteração ao cálculo do imposto pelo decreto de lei não é suportada por este sistema, por não permitir diferenciar os rendimentos por categorias. As possíveis categorias são condicionadas, basicamente, pela idade da pessoa segura, pelo tipo de sinistro declarado e pela idade das respectivas entregas (ou prémios pagos).

Considerando os requisitos do decreto de lei, resumidos no Anexo A, os processos necessários à criação do recibo de indemnização (Figura 19), e identificadas as diferenças entre o sistema de cálculo imposto pelo decreto e o sistema de cálculo actual (Figura 20), estão identificadas as principais necessidades de desenvolvimento para a implementação dos novos requisitos. Agrupando essas necessidades pelas três camadas da aplicação (serviço, domínio e base de dados):

I) Camada de serviço.
   (1) Desenvolver componente responsável por fornecer os serviços para o cálculo de imposto. Estes serviços serão utilizados por qualquer cliente que necessite de apurar o valor de imposto, como por exemplo, na consulta do IRS para um montante a resgatar ou na criação de um recibo de indemnização.
   (2) Desenvolver interface para tratar pedidos de cálculo via GIS. A necessidade de implementação deste componente deve-se ao pedido ser oriundo de uma plataforma diferente (iSeries/OS400) daquela onde o serviço de cálculo vai correr (iSeries/Unix).

II) Camada de domínio.
   (1) Implementar as regras publicadas pelo decreto de lei que regulamenta o cálculo de imposto para os resgates de produtos PPR/E.

(2) Refabricar o cálculo de provisões, de modo a que este permita obter o saldo de uma apólice para um grupo de entregas por data, não considerando apenas a data efeito do resgate, como é efectuado actualmente.
(3) Refabricar o módulo de sinistros para inclusão do método alternativo de cálculo de imposto, no momento da criação do recibo de indemnização.

III) Camada de base de dados.
(1) Criar uma estrutura física onde se guardem os valores do imposto calculados para as diferentes taxas.
(2) Mapear tabelas existentes no iSeries para objectos, de modo a que seja possível o seu acesso pela nova classe de cálculo.

Depois de identificados os requisitos necessários à implementação do novo sistema, pode-se adequar a estratégia de desenvolvimento às necessidades envolvidas no novo cálculo de IRS.

Abordagem de Desenvolvimento

Seguindo a metodologia proposta no capítulo anterior (ver Figura 7 do capítulo 3.2. Metodologias Propostas) e tendo sido recolhidas as histórias do cliente e modelado o processo de negócio (Figura 19, Figura 20 e Anexo A), os próximos passos são a especificação de testes, seguida da implementação. A especificação de testes está apresentada na próxima secção deste capítulo (4.3 Testes Implementados). Segue-se nesta secção, a descrição da codificação implementada de acordo com os requisitos identificados para a implementação do novo cálculo.

É necessário agora enquadrar os requisitos com os componentes GIS, quer os já existentes, quer os que necessitam ser desenvolvidos para suportar o novo sistema de cálculo. Assim, na Figura 21 estão representados os novos componentes a serem desenvolvidos e a sua interligação com os componentes existentes.

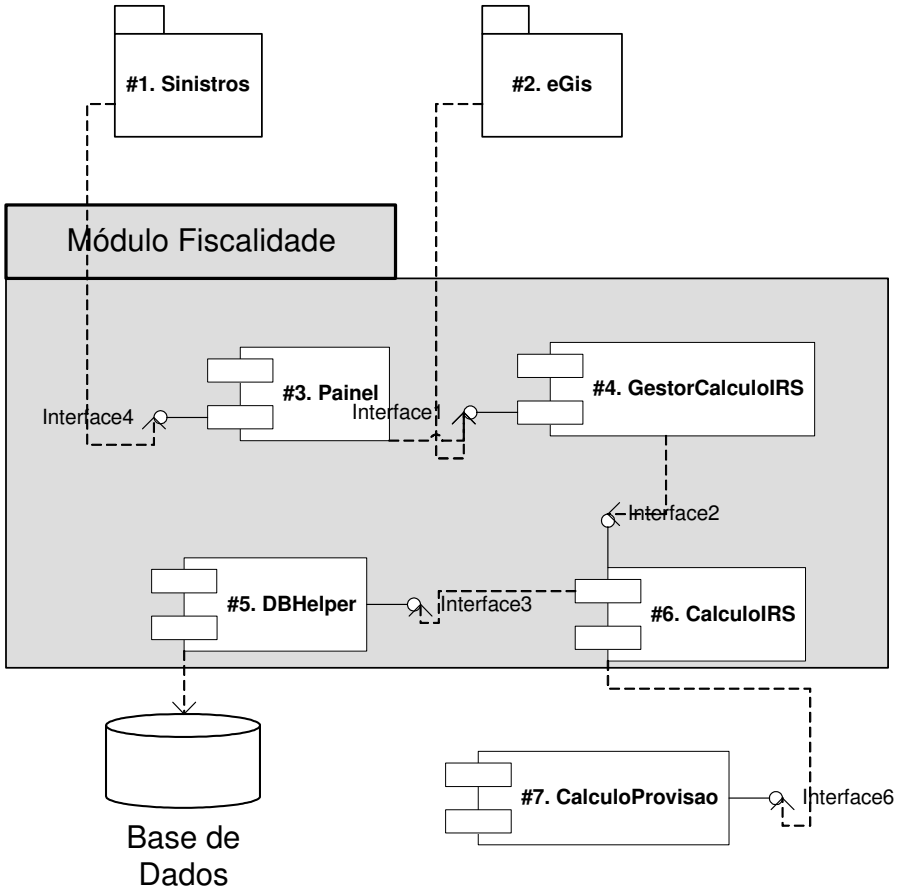


Figura 21 - Interligação de componentes

Os componentes existentes no GIS (Figura 21) são o módulo *sinistros* (#1), o *eGis* (#2) e o sistema de cálculo de provisões, *CalculoProvisao* (#7). Todos os restantes componentes não estão ainda implementados, pelo que, devem ser desenvolvidos. Enquadrando os componentes com as respectivas camadas da aplicação, identifica-se que: os componentes *Painel* (#3) e *GestoCalculoIRS* (#4) são relativos à camada de serviços (tratando pedidos de #1 e #2); os componentes *CalculoIrs* (#6) e *CalculoProvisao* (#7) referem-se à camada de domínio; o componente *DBHelper* (#5) é relativo à camada de base de dados. Segue-se a descrição destes componentes:

(#1) *Sinistros*: responsável pelos processos de abertura de processo de sinistros e criação de recibos de indemnização. Interage com o serviço de cálculo de imposto através do interface *Painel* (#3). Após obter o valor do imposto dá origem à criação de um recibo de indemnização para o valor do resgate em causa.

(#2) *EGis*: representa a vertente do GIS dedicada aos processos via web, em que a plataforma tecnológica de suporte é o Windows. Interage com o sistema de cálculo de imposto com o objectivo de obter o valor do imposto para as simulações de resgate via web.

(#3) *Painel*: é um componente de interface e comunicação entre o componente de sinistros (#1) e o serviço de cálculo (#4). Trata os pedidos de cálculo de imposto oriundos do componente de sinistros.

(#4) *GestoCalculoIRS*: é o serviço que fornece os interfaces necessários para o cálculo do imposto. Qualquer pedido de cálculo de imposto é processado através deste serviço.

(#5) *DBHelper*: componente de suporte ao carregamento e actualização de informação oriunda da base de dados central, actualmente totalmente localizada no iSeries. Carrega a informação necessária ao cálculo de imposto a partir das tabelas da base de dados e faz o mapeamento da informação dessas tabelas relacionais para objectos.

(#6) *CalculoIRS*: contém a lógica de negócio para o cálculo do IRS dos produtos PPR/E, implementando as regras descritas no respectivo decreto de lei. Apenas pode ser acedido através do componente de serviço de cálculo (#4).

(#7) *CalculoProvisao*: sistema de cálculo de provisão que utiliza o motor de cálculo do GIS. Obtém a provisão de uma apólice considerando os montantes das entregas e resgates efectuados até ao momento do resgate, capitalizando à respectiva taxa de juro.

Codificar e Refabricar com Padrões

Considerando os requisitos identificados, é necessário agora implementá-los de acordo com os componentes descritos. Sendo a refabricação uma intervenção que visa melhorar o código existente, tem como alvo os componentes já existentes. Por outro lado, a implementação de novos componentes deve ser realizada, sempre que seja aplicável, com recurso a padrões de desenho conhecidos. A abordagem de desenvolvimento que se segue está organizada de acordo com as camadas da aplicação, serviço, domínio e base de dados.

Camada de Serviço

Nesta camada surge a necessidade de disponibilizar métodos de acesso ao cálculo de imposto para tratamento de pedidos externos. Todo e qualquer pedido de cálculo deve ser efectuado através do serviço disponibilizado. Na lógica dos padrões EJB [Marinescu 2002] esta é uma camada de fronteira entre a interface de acesso e a lógica de domínio. Deve-se implementar aqui a camada de interface que define os métodos que os clientes poderão aceder ao serviço para o cálculo do imposto.

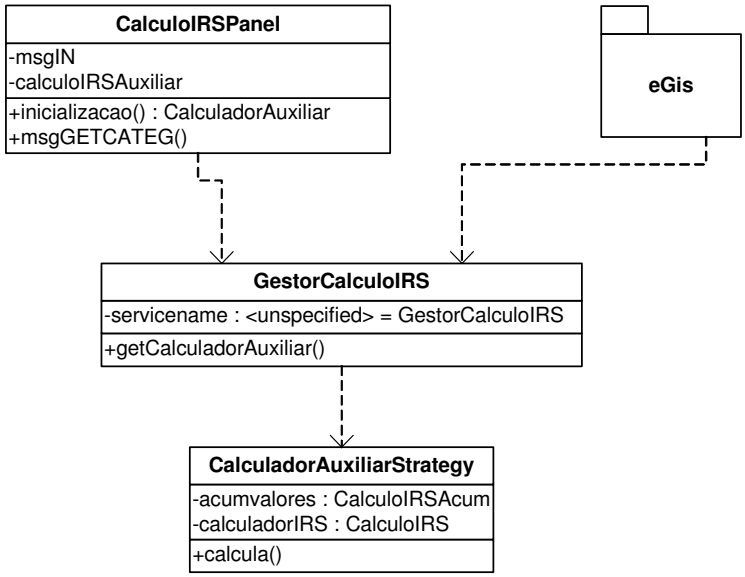


Figura 22 - Classes do sistema de cálculo do Imposto

A classe *GestorCalculoIRS* (Figura 22) é uma máscara que esconde a complexidade do processo de negócio, de quem o invoca. Sendo esta uma camada entre o cliente e as classes que implementam a lógica de negócio, possui as características que permitem enquadrar a sua implementação no contexto do padrão *façade*.

```
1. @author: Nuno Silva
2. */
3. public interface GestorCalculoIRS extends GestorCalculoIRSInternal {
4. public static final String SERVICENAME = "GestorCalculoIRS";
5. public HashMap calculoIRSByApolice(
6. int modalidade,
7. int numApolice,
8. int numAdesao,
9. int nord,
10.   int cobertura,
11.   char tipoProvisao,
12.   double valorResgate,
13.   double valorIndMax,
14.   Data dataResgate,
15.   char tipoResgate,
16.   int tipoInternoSinistro)
17.   throws RemoteException, ServiceShuttingDownException,
18.   UnAbleToCreateClassException;
19.
20.   public String calculoIRSByParms(
21.   int dataApolice,
22.   double saldoApolice,
23.   double capitalApolice,
24.   double valorResgate,
25.   double valorIndMax,
26.   int dataResgate)
27.   throws RemoteException, ServiceShuttingDownException,
28.   UnAbleToCreateClassException;
29.   public CalculoIrsStrategy getCalculadorAuxiliar(
30.   int modalidade,
31.   int numApolice,
32.   int adesao,
33.   int nord,
34.   int cobertura,
35.   char tipoProvisao,
36.   double valorResgate,
37.   double valorindMax,
38.   Data dataRes,
39.   char tipoResgate,
40.   int tipoInternoSinistro);
41.   }
```

Figura 23 - GestorCalculoIRS , serviço de cálculo de imposto

Como *façade*, esta classe (Figura 23) contém apenas os métodos que os clientes necessitam conhecer para processar o cálculo de imposto.

Na camada de serviço, surge também a necessidade de tratar pedidos com origem numa plataforma diferente daquela onde se encontra disponibilizado o serviço de cálculo de imposto. O pedido de cálculo originado pela criação do recibo de indemnização, sendo um processo implementado em RPG e que é executado no iSeries, é um destes casos. Para a implementação deste serviço é necessário: (1) criar uma camada que faça a

comunicação entre o componente de sinistros escrito em RPG e o serviço de cálculo escrito em Java; (2) refabricar o componente de sinistros, para obter o valor de imposto através do novo serviço de cálculo de imposto.

Para a comunicação entre os programas RPG e o novo código em Java que corre numa máquina virtual, é necessário recorrer ao módulo de comunicação que a I2S denominou de YASP (*Yet Another Service Provider*). O YASP fornece as APIs necessárias para envio e recepção de mensagens dos objectos Java. No entanto, é necessário criar um programa que receba o pedido de cálculo do componente de sinistros e, usando as APIs disponibilizadas pelo YASP, encaminhe o pedido para o componente *Painel* (#3) que se responsabilizará de invocar o serviço de cálculo.

```
1.  * MSGGETVCAT - Enviar msg para obter valor de uma categoria de IRS
2.  PMSGGETVCAT       B                   EXPORT
3.  DMSGGETVCAT       PI             1A
4.  D P#CATEG                       10A   VALUE
5.  D P#VALORI                      17P 5
6.  D P#VALORC                      17P 5
7.  ,* Declaração das estruturas que são passadas para este programa
8.  D DSPARM          DS
9.  D DS_CATEG                        10
10. D TAMMSG          S               10I 0 INZ(*ZEROS)
11. D DATAOUT         S             1000A
12. ,* Declaração das estruturas que são passadas para este programa
13. D DSOUT           DS
14. D DSO_TAM                          2S 0 INZ(*ZEROS)
15. D DSO_VIMP                        17S 0 INZ(*ZEROS)
16. D DSO_VCAP                        17S 0 INZ(*ZEROS)
17. D DSO_SINL                         1A   INZ(*BLANKS)
18. D TMP             S               17A
19. D TMPVALOR        S               17  5
20. D MAXTAMMSGOUT    S               10I 0 INZ(%SIZE(DSOUT))
21. C* Construir mensagem
22. C                   MOVE      P#CATEG       DS_CATEG
23. C                   EVAL      TAMMSG = %SIZE(DSPARM)
24. Envia a mensagem
25. C                   EVAL      YSMSGID = GETVCATEG
26. C                   EVAL      PNLNAME = CALIRS_PNL
27. C                   CALLP     YSENVMSG(SESSION :PNLNAME :YSMSGID
28. C                                      :DSPARM :TAMMSG)
29. C                   CALLP     YSRCVMSG(SESSION :PNLNAME :YSMSGID
30. C                                      :DATAOUT :MAXTAMMSGOUT)
31. C
32. C                   MOVEL     DATAOUT       DSOUT
33. C                   IF        DSO_TAM = *ZEROS
34. C                   RETURN    *OFF
35. C                   ENDIF
36. ,* Valor Imposto
37. C                   EVAL      TMPVALOR = DSO_VIMP
38. C                   EVAL(H)   P#VALORI=  TMPVALOR
39. C                   IF        DSO_SINL = '-'
40. C                   EVAL      P#VALORI = P#VALORI* (-1)
41. C                   ENDIF
42. ,* Valor Capital
43. C                   EVAL      TMPVALOR = DSO_VCAP
44. C                   EVAL(H)   P#VALORC=  TMPVALOR
45. C     FIM_MSG       TAG
46. C                   IF        DSO_TAM <> *ZEROS
47. C                   RETURN    *ON
48. C                   ENDIF
49. C                   RETURN    *OFF
50. P                 E
```

Figura 24 - Comunicação entre RPG e Java (RPG)

Na Figura 24, pode-se verificar nas linhas 27 e 29, o modo como são usadas as APIs do YASP para invocar um serviço Java. Os métodos *YSENVMSG* e *YSENVMSG* respectivamente enviam e recebem uma mensagem que depois é formatada e retornada para os programas que os invocaram.

Há a necessidade de desenvolver a classe de comunicação *CalculoIrsPanel* que processa os pedidos de cálculo com origem nos processos RPG. Essa classe é responsável pela implementação de métodos para a recepção de mensagens num formato conhecido e posterior comunicação com o serviço de cálculo de IRS. Na Figura 25, pode-se analisar a implementação do método dessa classe responsável por obter a lista das categorias de imposto. Na linha 4 é formatada a informação que o método recebe. Na linha 9, o método *inicializacao*, carrega a estratégia de cálculo a ser usada para o cálculo, processando o cálculo na linha 19.

```
1. public class CalculoIRSPanel extends I2SPanelBase {
2. ….
3. public SSMPGetCategoriasParmsOut msgGETCATEG(I2SMP100 msgIN100) {
4.      SSMPGetCategoriasParmsIn msgIN = new SSMPGetCategoriasParmsIn(msgIN100);
5.         SSMPGetCategoriasParmsOut resultado = new SSMPGetCategoriasParmsOut();
6.     CalculoIrsStrategy calculadorIrsAu = null;
7.         //dado que agora os valores são obtidos via Gestor de serviço, têm que ser tratadas as suas exceptions
8.         try {
9.             calculadorIrsAux = inicializacao(msgIN);
10.            }
11.            catch (ServiceShuttingDownException e1) {
12.                getLog().error(
13.                    "Serviço GestorCalculoIRS está a ser desligado, impossível fazer cálculo: ", e1);
14.                resultado.setTamCategorias(1);
15.                resultado.setErro("*ERR");
16.                return resultado;
17.            }
18.            /* fazer o cálculo: até este momento só foram usados os valores acumulados, não foi feito o cálculo */
19.            valores = calculadorIrsAux.calcula();
20.            ...
21.            return resultado;
22.        }
```

Figura 25 - CalculoIRSPainel

O *CalculoIRSPainel* que corresponde ao componente *Panel* (#3) depois de receber o pedido tem que invocar o cálculo de imposto, usando o serviço *GestorCalculoIRS* (#4) (ver Figura 21). Analisando a Figura 26, linha 27, verifica-se a invocação do método *getCalculadorAuxiliar* da classe de serviço *GestorCalculoIRS* que retorna a estratégia de cálculo, que posteriormente será usada para o cálculo.

```
1.  protected CalculoIrsStrategy inicializacao(
2.      int modalidade, int numApolice, int adesao,
3.      int nord, int cobertura,
4.      char tipoProvisao, int dataResgate,
5.      double valorResgate, double valorIndMax,
6.      char tipoResgate, int tipoInternoSinistro)
7.      throws
8.          UnableToLoadException,
9.          ServiceShuttingDownException,
10.             RemoteException,
11.             UnAbleToCreateClassException {
12.
13.         //o valor de resgate que veio via YASP tem que ser decimalizado
14.         valorResgate =
15.             getSession()
16.                 .getAmbiente()
17.                 .applyFactorDecimal(BigDecimal.valueOf(valorResgate))
18.                 .doubleValue();
19.
20.         valorIndMax =
21.             getSession()
22.                 .getAmbiente()
23.                 .applyFactorDecimal(BigDecimal.valueOf(valorIndMax))
24.                 .doubleValue();
25.         Data dataRes = new Data(dataResgate);
26.
27.         return getGestorCalculoIRS().getCalculadorAuxiliar(
28.             modalidade, numApolice, adesao,
29.             nord, cobertura, tipoProvisao,
30.             valorResgate, valorIndMax,
31.             dataRes, tipoResgate,
32.             tipoInternoSinistro);
33.     }
```

Figura 26 - CalculoIrsStrategy.inicializacao

A estratégia de cálculo é obtida pelo serviço de cálculo de IRS. O provedor de serviços devolve uma instância do serviço de IRS activo (arrancando o serviço caso ainda não esteja activo), a partir da qual de irá obter a estratégia de cálculo apropriada para executar o pedido de cálculo recebido pelo *CalculoIrsPanel*.

Camada de Domínio

Na camada de domínio existem diversos tipos de intervenção a realizar: (1) refabricar o componente de sinistros para incluir como método alternativo de cálculo o novo sistema de cálculo para PPR/E; (2) é necessário desenvolver o novo componente de cálculo com a implementação das novas regras de fiscalidade; (3) refabricar o sistema actual de cálculo de provisões para permitir obter os saldos das apólices de acordo com as necessidades do novo cálculo.

Começando por intervir sobre o componente de sinistros e, sendo esta uma intervenção de refabricação sobre a base de código em RPG, esta é de difícil execução por não existirem ferramentas que suportem estas modificações nesta base de código. A refabricação, neste caso, surge da necessidade de incluir um novo método de cálculo de imposto sem que isso afecte o sistema actual de sinistros. A decisão sobre qual o método de cálculo a usar, de acordo com a configuração, é retirada do processo de criação do recibo de indemnização. Assim, o processo apenas pede o valor de imposto para um dado resgate, que será calculado de acordo com as regras configuradas e seguidamente retornado.

```
1.  C                   SELECT
2.  C                   WHEN      WTCAL = 'E'
3.   * Invocar programa de cálculo de imposto
4.  C                   CALL      'CALCIMP'
5.  C                   PARM                    CI_$MOD
6.  C                   PARM                    CI_NVER
7.   *Verificar se a variável de ambiente indica se usa os serviços java
8.  C                   WHEN      FLGCALIRS <> 'A'
9.  C                             AND FLGCALIRS <> *BLANKS
10. C                             AND (IS_PPR(CI_$MOD:CI_NVER)=*ON)
11. C                  EXSR      EXE_NEWCLC
12. C                  WHEN      WTCAL = 'P'
13. C                             AND (FLGCALIRS = 'A'
14. C                             OR FLGCALIRS = *BLANKS)
15. C                   EXSR      EXE_PGM
16. C                   OTHER
17. C                   SELECT
18. C                   WHEN      CI_VAL_LIQ>*ZEROS AND
19. C                                       CI_VAL_TOT=*ZEROS
20. C                             AND FLGCALIRS <> *BLANKS
21. C                             AND (IS_PPR(CI_$MOD: CI_NVER)=*ON)
22. C                   EXSR      EXE_NEWCLC
23. C                   WHEN      WTCAL = 'P'
24. C                             AND (FLGCALIRS = 'A'
25. C                             OR FLGCALIRS = *BLANKS)
26. C                   EXSR      EXE_PGM
27. C                   OTHER
28. C                   EVAL      CI_VAL_IRS = *ZEROS
29. C                   ENDSL
30. C                   ENDSL
```

Figura 27 – Selecção do cálculo de imposto - *façade*

A Figura 27 contém, em código RPG, os critérios de selecção do método apropriado de cálculo de imposto a ser usado num processo de sinistro. O objectivo deste código é esconder a complexidade da lógica de selecção do método de cálculo de imposto no processo de criação de resgate. O processo de criação de resgate não necessita de saber qual é o método de cálculo usado para calcular o imposto de um determinado produto. Esta intervenção surge, então, no âmbito da aplicação do padrão *façade*.

A implementação do novo cálculo surge também, no contexto da camada de domínio. Podendo o cálculo de IRS divergir de acordo com o país ou mesmo com o tipo de produto, interessa que o calculador possa ser definido em tempo de execução, de acordo com a informação disponível no momento do pedido de cálculo. A camada de serviço é responsável por esse isolamento e pela identificação da estratégia de cálculo a ser utilizada. O decreto de lei define as regras de cálculo do imposto para os produtos PPR/E. No entanto, há a necessidade de prever a implementação de outras regras de cálculo no futuro. De acordo com os padrões estudados, o padrão estratégia é o que melhor se ajusta a esta necessidade. Na Figura 28, é apresentado o diagrama de classes onde está esquematizada a implementação desse padrão.

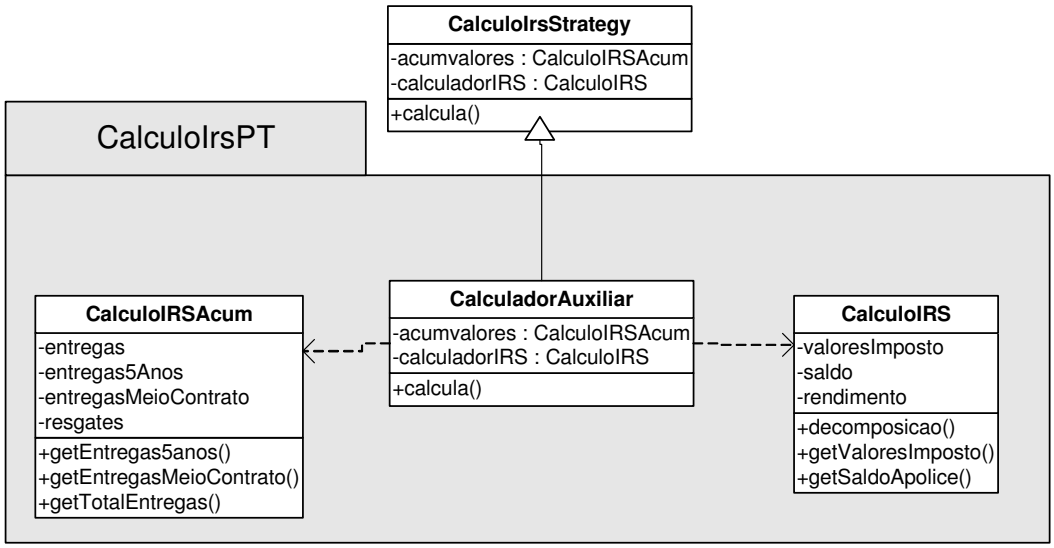


Figura 28 - Padrão estratégia para o calculador do imposto

É possível observar que o calculador está a utilizar a classe abstracta *CalculoIrsStrategy* para implementar os seus métodos sob a estrutura desse interface. A estratégia de cálculo deve ser obtida pelo serviço de cálculo, através da classe *GestorCalculoIRSImpl* (classe que implementa o interface *GestorCalculoIrs*). É, portanto, no momento em que é recebido o pedido de cálculo que é definida a estratégia adequada de cálculo para apuramento do valor do imposto.

```
1. public class GestorCalculoIRSImpl extends PanelServer
2. implements
3.     GestorCalculoIRS, GestorCalculoIRSInternal, GestorCalculoIRSClient,
4.   ServiceProvider, Remote {
5.     private static final String PropertyName_StrategyCalcuImpPPRE ="com.i2s.giv.irs.imp.ppr";
6.     public CalculoIrsStrategy getCalculoIrsStrategy() throws UnableToLoadException {
7.         // Construção de property
8.         String name_prop = PropertyName_StrategyCalcuImpPPRE +I2SSystem.CLASS_EXT;
9.         CalculoIrsStrategy obj = null;
10.            try {
11.                String property = getAmbiente().getProperties().getProperty(name_prop);
12.                if (property == null) {
13.                    throw new UnableToLoadException("Não encontrou propriedade" + name_prop);
14.                }
15.                obj = (CalculoIrsStrategy) ambiente.createInterfaceImplementation(property);
16.                if (!(obj instanceof CalculoIrsStrategy)) {
17.                    return null;
18.                }
19.            }
20.            catch (UnAbleToCreateClassException e) {
21.                log().error("Unable to create class " + name_prop, e);
22.                return null;
23.            }
24.            return obj;
25.        }
```

Figura 29 - Leitura de estratégia a usar

Na Figura 29 pode-se verificar que o contexto de cálculo é obtido através do conteúdo de uma propriedade de ambiente na linha 11 (*"com.i2s.giv.irs.imp.ppr"*). Esse contexto identifica qual a estratégia de cálculo a usar. É criado o objecto de cálculo de acordo com a estratégia identificada na linha 15. Na criação do objecto que implementa a estratégia de cálculo identifica-se uma nova oportunidade para a utilização de padrões. Esta implementação de pode ser auxiliada pela utilização do padrão *abstract factory*, que pertence à classe dos padrões concepcionais.

```
1.          public CalculoIrsStrategy getCalculadorAuxiliar(
2.          int modalidade int numApolice, int adesao, int nord,
3.          int cobertura, char tipoProvisao, double valorResgate, double valorIndMax, Data dataRes,
4.          char tipoResgate,  int tipoInternoSinistro) {
5.          CalculoIrsStrategy calculador = null;
6.          try {
7.          calculador = getCalculoIrsStrategy();
8.              calculador.calculadorprepare(
9.                  ambiente,
10.                     modalidade, numApolice, nord, cobertura,
11.                     dataRes, tipoResgate, tipoProvisao,
12.                     tipoInternoSinistro,
13.                     valorResgate, valorIndMax);
14.
15.             }
16.             catch (UnableToLoadException e) {
17.         log().error("UnableToLoadException ");
18.     }
19.    return calculador;
20.   }
```

Figura 30 - GestorCalculoIrsImpl.getCalculadorAuxiliar

Depois de encontrada a estratégia de cálculo apropriada, é necessário preparar o objecto para o cálculo. Na Figura 30, o método *getCalculadorAuxiliar* obtém a estratégia a ser usada para o cálculo na linha 7 e invoca o método *calculadorprepare* na linha 8 do tipo classe abstracta *CalculoIrsStrategy*, para inicializar a classe com os valores necessários ao cálculo.

Ainda na camada de domínio, surge a necessidade de obter o saldo da apólice considerando entregas e resgates realizados até uma determinada data, mas com capitalização até à data do sinistro. Só assim, é possível taxar o rendimento, a diferentes taxas de imposto, considerado as datas de entregas (ver Anexo A).

Esta é uma necessidade que o sistema actual de cálculo de provisões não consegue dar resposta, sendo por isso necessário refabricar para permitir este comportamento. A refabricação por si só, não altera (nem pode alterar, por definição) o sistema de cálculo de provisão, a intenção é parametrizar as datas de *input* numa primeira fase, sem alterar o comportamento do cálculo de provisões (Figura 31). Numa segunda fase, serão acrescentadas as regras alternativas para selecção dos montantes de entregas e resgates por data.

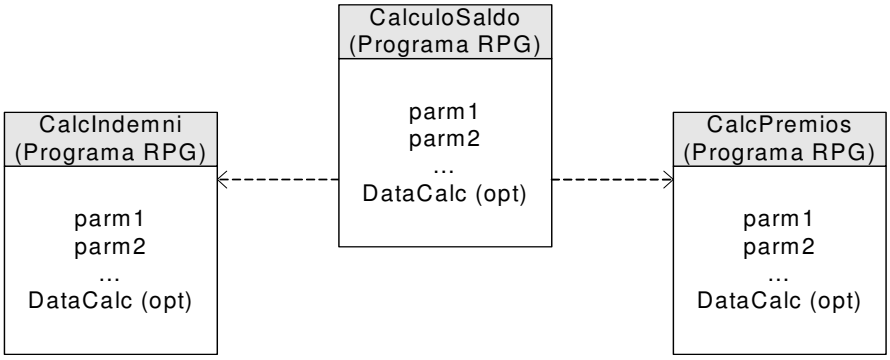


Figura 31 - Refabricação do cálculo de provisão

Esta figura representa as alterações necessárias ao cálculo de provisão para apurar o saldo de acordo com as necessidades das regras do novo cálculo. Os três processos RPG necessários ao cálculo da provisão são refabricados para incluírem um parâmetro

opcional *DataCalc*. O sistema continua a funcionar sem a passagem desta informação, no entanto, quando esta é passada, os processos de selecção de prémios e indemnizações serão condicionados por essas datas.

O cálculo de imposto necessita obter o valor de provisão de uma apólice para apurar o cálculo de imposto. Para isso necessita aceder a um componente já existente no GIS, o componente de cálculo de provisões. O padrão *command* padroniza este tipo de necessidade, permitindo tratar o pedido com a invocação de um objecto, onde é parametrizado o tipo de pedido Figura 32.

```
1.  /**
2.   * Faz o cálculo dos saldos para as várias datas e atribui às variáveis locais
3.   * @param buffer
4.   */
5.  protected void calcAS400Sld(String buffer) {
6.      AS400Slave as400slave = getAmbiente().getAs400SlavePool().getAS400Slave();
7.      try {
8.          AS400MsgReturn msg = as400slave.request(SLV_SLD_IRS_NAME, buffer);
9.          //liberta o escravo
10.             getAmbiente().getAs400SlavePool().freeSlave(as400slave);
11.             //
12.             if (msg.getI2SAs400Message() == null) {
13.                 //vamos ter de inserir o separador decimal
14.                 SlvSaldoIRSRtnMsg rtnmsg =
15.                     new SlvSaldoIRSRtnMsg(msg.getI2sMP100Message(), getAmbiente());
16.                 setSaldoApoliceDataResg(
17.                     (getAmbiente().applyFactorDecimal(rtnmsg.getSaldoDataResgate()))
18.                         .doubleValue());
19.             }
20.             else {
21.                 getLog().error(msg.getI2SAs400Message().getMsgData());
22.             }
23.         }
24.         catch (AS400SlaveRequestException e) {
25.             getLog().error(
26.                 "AS400SlaveRequestException: Retrieving information from Slave",
27.                 e);
28.         }
29.         catch (RuntimeException e) {
30.             getLog().error("RuntimeException: Retrieving information from Slave", e);
31.             getAmbiente().getAs400SlavePool().freeSlave(as400slave);
32.             throw e;
33.         }
34.     }
```

Figura 32 - Método calcAS400Sld

Por outro lado, é necessário existir um programa em RPG que receba o pedido de Java e processe o cálculo de provisão. Este programa, baptizado de *slave*, não deve ter lógica embutida. A sua tarefa limita-se em receber o pedido, converter os parâmetros recebidos para o formato conhecido pelo programa de cálculo de provisões, executar o cálculo e devolver o resultado obtido (Figura 33).

```
1.  ******************************************************************************
2.  C     *ENTRY        PLIST
3.  C                   PARM                    P#BUFFIN
4.  C                   PARM                    P#BUFFIN_SZ
5.  C                   PARM                    P#BUFFOUT
6.  C                   PARM                    P#BUFFOUT_SZ
7.  C                   PARM                    P#BUFFOUT_MXSZ
8.  C                   PARM                    P#MSG
9.  ******************************************************************************
10. C                   CALLP     SetMsg(P#MSG:GEN@OK:N@MSGF)
11. C                   EVAL      P#BUFFOUT_SZ = %SIZE(DSRTN)
12. C                   EVAL      PRTDADOS = %ADDR(P#BUFFIN)
13. C                   EVAL      SLDBUF = %ADDR(P#BUFFOUT)
14. C                   MOVE      $MOD          w$MOD
15. C                   MOVE      NVER          wNVER
16. C                   MOVE      NAPO          wNAPO
17. C                   MOVE      $CBT          w$CBT
18. C                   MOVE      DRSG          wDRSG
19. C                   MOVE      DTMC          wDTMC
20. C                   MOVE      DT5A          wDT5A
21. C                   MOVE      $ESQ          w$ESQ
22. C                   CALL      'CALCSLDIRS'
23. C                   PARM                    w$MOD
24. C                   PARM                    wNVER
25. C                   PARM                    wNAPO
26. C                   PARM                    w$CBT
27. C                   PARM                    wDRSG
28. C                   PARM                    wDTMC
29. C                   PARM                    wDT5A
30. C                   PARM                    wDT8A
31. C                   PARM                    w$ESQ
32. C                   PARM                    TESQ
33. C                   PARM                    YSLDDRSG
34. C                   PARM                    YSLDDT5A
35. C                   PARM                    YSLD2002
36. C                   PARM                    YSLDDTMC
37. C                   PARM                    P#MSG
38. C                   EVAL      SLDDRSG = YSLDDRSG
39. C                   EVAL      SLDDT5A = YSLDDT5A
40. C                   EVAL      SLD2002 = YSLD2002
41. C                   EVAL      SLDDTMC = YSLDDTMC
42. C                   IF        GetMsgID(P#MSG) ='GIV000' OR GetMsgID(P#MSG) ='STS000'
43. C                             GetMsgID(P#MSG) ='STS000'
44. C                   CALLP     SetMsg(P#MSG:GEN@OK:N@MSGF)
45. C                   ENDIF
```

Figura 33 - Recepção de um pedido Java (RPG)

Após receber o pedido de obtenção de saldos, o processo RPG, *CALCSLDIRS*, executado na linha 22 da Figura 33, invocará repetidamente o cálculo de provisões (processo *CalculoSaldo*, ver Figura 31) para cada uma das datas necessárias ao cálculo do imposto (data de resgate, data à cinco anos, data a meio do contrato, e data lei).

Camada de Base de Dados

O papel desta camada é tratar da comunicação entre os objectos e as tabelas relacionais. É necessário construir classes que farão a ponte entre as classes que implementam lógica de domínio e a base de dados. Encontra-se o padrão *Data Mapper* [Fowler 2002] que sugere como deve ser efectuada a gestão em memória dos dados de uma tabela e a sua movimentação (escrita e leitura), mantendo a independência entre a

lógica de domínio e a base de dados. Este padrão tem enúmeras aplicações, tornando-se, muitas vezes, um processo repetitivo dado o número de tabelas que o sistema possui. Por essa razão, é conveniente a utilização (ou desenvolvimento) de utilitários que auxiliem na geração automática do código. O uso de utilitários que permitam a geração automática de código facilita a implementação de futuras alterações que venham a ser necessárias efectuar.

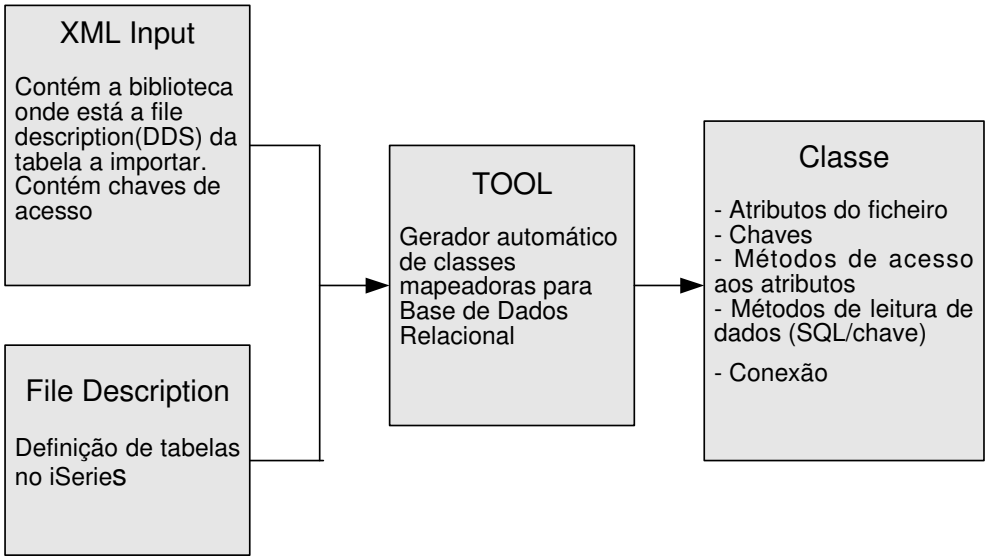


Figura 34 - Geração automática de código

O gerador utilizado para o mapeamento das tabelas foi o Castor (http://castor.exolab.org) que tem como entradas a definição da tabela DDS (*Data Description Specifications*) residente no iSeries, com a descrição dos campos que a constituem e um XML [McLaughlin 2002] que contém a localização da biblioteca onde existe a DDS, e as chaves que se pretendem usar para o acesso ao seu conteúdo (Figura 34). Como resultado, obtém-se a classe de mapeamento para a tabela relacional, com os \métodos necessários à manipulação da informação, assim como métodos que permitem obter colecções de dados para as chaves definidas no XML de entrada. A Figura 35 contém o XML relativo à tabela de valores da apólice (APOLVB), que guarda valores acumulados por apólice. Esta tabela armazena, entre outros, os valores das entregas relevantes para o cálculo de imposto (ver Anexo B), acumulados sob a forma de valores de balanço.

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by (I2S) -->
3. <TableClass xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="d:\i2sbusinessobj\i2stable.xsd" name="Apolvbl" package="com.i2s.giv.dados.tables400">
4. <imports>
5. <import>java.math.BigDecimal</import>
6. <import>java.sql.Date</import>
7. </imports>
8. <classDefinition/>
9. <connection>
10.   <driver>com.ibm.as400.access.AS400JDBCDriver</driver>
11.   <url>jdbc:as400://I2SSEG2;libraries=GIV008R;transaction isolation=none;date format=iso;big decimal=false</url>
12.   </connection>
13.   <keylist>
14.   <key unique="true">
15.   <fields>
16.   <field>ab$mod</field>
17.   <field>abnapo</field>
18.   <field>abnord</field>
19.   <field>ab$cbt</field>
20.   <field>abtipo</field>
21.   <field>ab$val</field>
22.   </fields>
23.   </key>
24.   </keylist>
25.   </TableClass>
```

Figura 35 - XML para geração da tabela APOLVB

A utilização deste tipo de ferramentas para o auxílio do processo de codificação representa três vantagens:

(1) Facilita o processo de codificação, permitindo que pedaços de código idênticos sejam gerados automaticamente, diminuído a probabilidade de inserção de erros de codificação e uniformizando o código escrito.

(2) Permite que, caso surja a necessidade de alguma modificação estrutural no código gerado automaticamente, seja possível a introdução dessas alterações em grande escala e sem um elevado esforço de desenvolvimento. Para isso, só é necessário definir as novas regras no gerador e voltar a gerar o código com base nas novas regras. Nestas intervenções é importante garantir a existência de um bom conjunto de casos de teste que garantam a imutabilidade de comportamento do sistema após a reconstrução do código.

(3) Desenvolvimento de classes procurando adoptar padrões que permitam uma melhor e efectiva reutilização de código.

No diagrama de classes da Figura 36 estão representadas as classes geradas de uma forma automática que permitem o acesso à informação residente na base de dados para obter os agrupamentos necessários à aplicação das taxas do imposto.

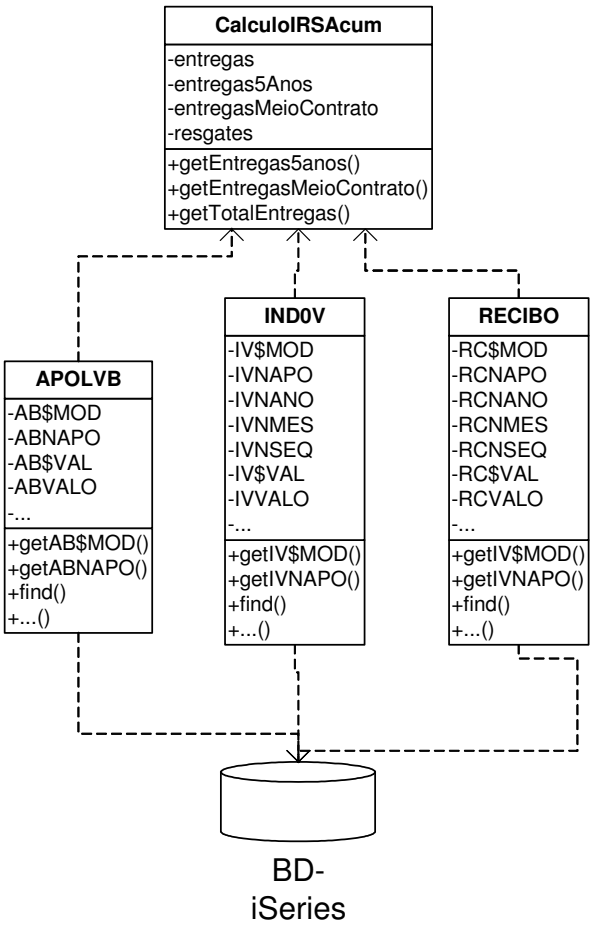


Figura 36 - Classes de acesso à base de dados

No cálculo do imposto é necessário conhecer os prémios pagos agrupados por diferentes categorias e a informação relativamente ao contrato de seguro que está a ser alvo de resgate. Segundo o decreto de lei que regulamenta o cálculo de imposto, é necessário agrupar os prémios pagos nas seguintes categorias: (1) entregas realizadas há mais de cinco anos em relação à data de resgate; (2) entregas realizadas até ao meio de contrato, em relação à vigência de contrato no momento do resgate; (3) entregas até à

publicação do decreto de lei no Diário da República. O acesso às tabelas relacionais através destas classes de mapeamento permite obter essa informação da base de dados.

Na intervenção sobre o componente *DBHelper* (#5) pode-se utilizar um padrão para o auxílio na melhoria do desempenho do cálculo. O padrão *lazy load* tipifica o carregamento da informação da base de dados concretizando a leitura no momento em que a informação seja necessária ao cálculo. Assim, permite que no caso de não haver necessidade de carregar informação, a conexão à base de dados e a leitura não sejam realizados, representado uma considerável melhoria no desempenho (Figura 37).

```
1. public double getTotalEntregas() {
2.   if (isNullTotakEntregas) {
3.     return loadEntregas();
4.   }
5.  return totalEntregas;
6. }
```

Figura 37 - Implementação do lazy load

Resumo das Refabricações e Padrões Usados

A percepção dos padrões relevantes para a resolução de um determinado problema é importante, para que, desde o início, a aplicação adquira uma boa organização de código. Sem a percepção dos padrões apropriados, haverá a necessidade de efectuar múltiplas iterações de código no sentido de o organizar posteriormente. Assumindo que o grande objectivo num novo desenvolvimento, advogado pelas metodologias ágeis, é assegurar a disponibilização imediata da funcionalidade ao cliente, não deve ser, no entanto, descartada a possibilidade de desenvolver bom código sempre que se tenha a percepção de quais as estruturas correctas a adoptar.

Neste *case study* procurou-se adoptar de início, nos novos desenvolvimentos, as estruturas correctas para o código através da utilização de padrões de desenho. Por sua vez, as refabricações tiveram como alvo o código existente em RPG que não estava preparado para acolher os novos requisitos impostos pelas novas regras de fiscalidade.

Assim, conclui-se que o rejuvenescimento de aplicações deve ser analisado numa perspectiva de refabricação do sistema existente, adaptando-o para a absorção dos novos requisitos emergentes, em que a implementação desses requisitos deve ser enquadrada, sempre que possível, em padrões conhecidos, mas nunca esquecendo o principal propósito de disponibilizar o mais cedo possível a nova funcionalidade ao cliente.

Para a implementação da nova fiscalidade foram utilizados um conjunto de padrões que se adequaram ao problema em análise, e foram realizadas refabricações no código existente, considerando sempre a possibilidade de adoptar padrões conhecidos (Tabela 3).

Tabela 3 - Quadro resumo de refabricações e padrões utilizados

Camada	Componentes envolvidos	Padrões Aplicados	Refabricação
Serviço	#3. Painel	*Command*	Não
	#4. GestorCalculoIrs	*Façade*	Não
Domínio	#1. Sinistros	*Façade*	Sim
	#6. CalculoIRS	Estratégia *Abstract Factory*	Não
	#7. Provisão	(nenhum)	Sim
Base de Dados	#5. DBHelper	*Data Mapper* *Lazy load*	Sim

A desvantagem da adopção de padrões é que, muitas vezes, a necessidade da aplicação destes só se torna perceptível no momento em que é necessário incluir novas alterações ao código, verificando-se, então, que este não está estruturado convenientemente de modo a suportar essas alterações. Daí surge a necessidade de se efectuarem várias iterações (do ciclo de desenvolvimento) para restruturar convenientemente o código. Esta situação ultrapassa-se à medida que a equipa de desenvolvimento vai ganhando experiência, conseguindo, assim, associar mentalmente os problemas que surgem, a modelos padronizados conhecidos, que são a solução.

Por outro lado, as vantagens identificadas com aplicação das técnicas da refabricação e aplicação de padrões com a implementação do módulo fiscalidade foram: (1) desenvolvimento de código estruturado logo apartir dos primeiros ciclos de desenvolvimento; (2) código organizado e simples; (3) agilização do sistema, conferindo-

lhe escalabilidade; (4) possibilidade de reutilização de componentes antigos no novo módulo de fiscalidade.

4.3. Testes Implementados

Uma boa especificação de testes permite garantir a qualidade do software. Recorrendo a um processo sistematizado de definição de testes, é possível construir um conjunto de testes que abranjam todas as variações possíveis numa regra de negócio. A especificação de testes é também uma forma de descrever o sistema, documentando as suas funcionalidades. No desenvolvimento do componente de cálculo, as especificações aqui definidas servem de especificações de desenvolvimento, com a descrição do comportamento esperado para o cálculo de imposto. Nesta secção é usada a abordagem descrita no capítulo anterior desta dissertação para a especificação e implementação de testes unitários.

A construção e realização de testes são de extrema importância qualquer que seja a metodologia de desenvolvimento adoptada. No entanto, quando se aplica a metodologias ágeis, como o *eXtreme Programming*, esta importância fica redobrada devido às técnicas que são aplicadas neste tipo de metodologias. Técnicas como a refabricação pressupõem que o código existente não modifica o seu comportamento após a sua aplicação. Para isso, é necessário que o código desenvolvido tenha mecanismos de testes que o garantam. O *Junit* é um utilitário de testes que foi desenvolvido com a finalidade de facilitar a codificação de testes unitários. No entanto, além da ferramenta é necessária uma estratégia que permita desenvolver os testes de uma forma sistemática e que assegurem a fiabilidade do código desenvolvido. Com este objectivo e aplicando os princípios já referidos no capítulo anterior desta dissertação, aplica-se neste *case study* uma metodologia de especificação de testes baseada em [Labiche 2001].

O primeiro passo neste processo de especificação de testes é a representação das sequências ou actividades envolvidas no componente que se pretende validar. As possíveis derivações obtidas com essa análise permitem definir quais os testes a

implementar. Assim, na Figura 38 é feita a representação do diagrama de actividades para o cálculo de IRS que permitam o resgate dentro das condições previstas de resgates, isto é com taxa a 4%.

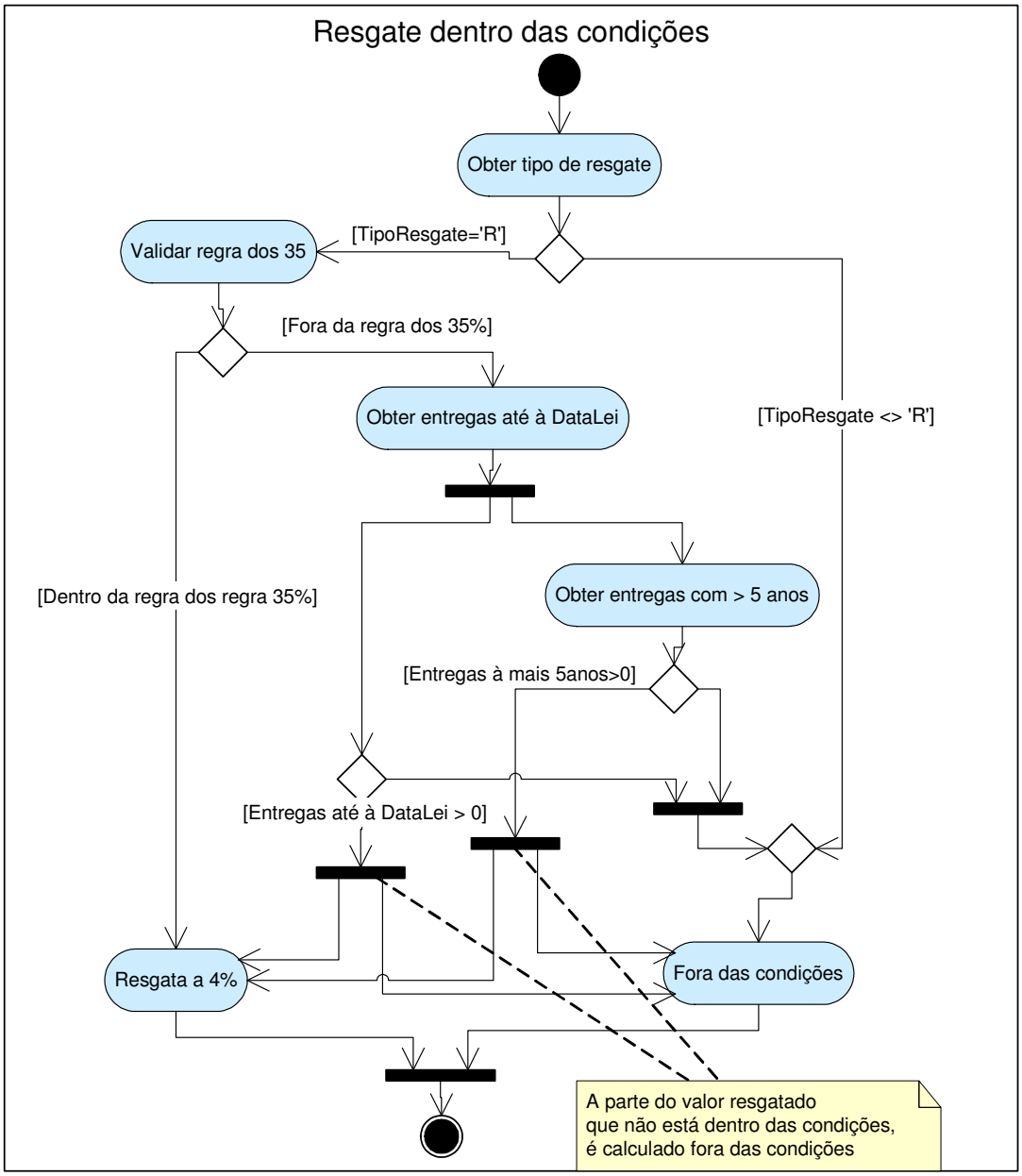


Figura 38 - Diagrama de actividades para resgate dentro das condições

Não sendo possível efectuar o resgate dentro das condições previstas, o resgate fora das condições pode ter uma taxa variada dependendo da idade do contrato de seguro e do histórico dos prémios pagos nesse contrato. Assim, é possível derivar o resgate a realizar fora das condições de acordo com as actividades definidas na Figura 39.

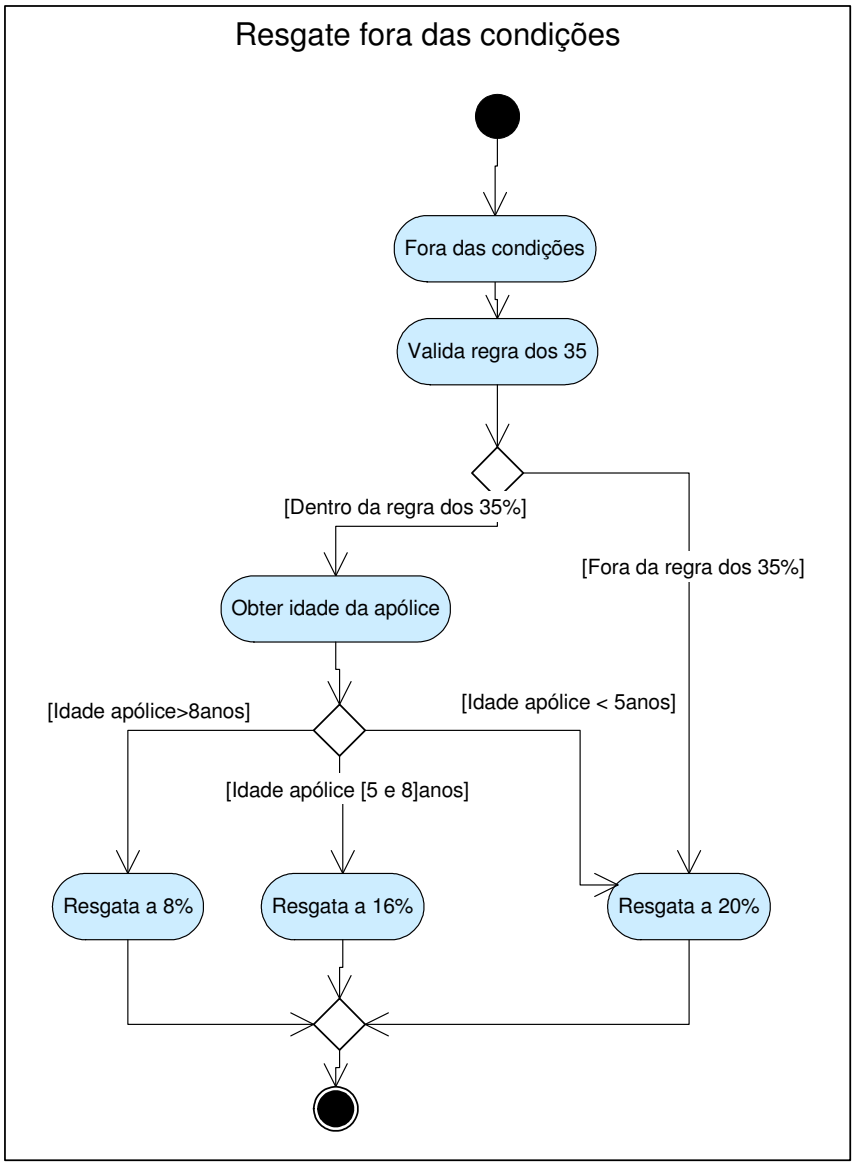


Figura 39 - Diagrama de actividades para resgate fora das condições

Com base nos diagramas de actividades da Figura 38 e da Figura 39, é possível identificar todas as possibilidades de cálculo da taxa de imposto para a definição dos testes unitários a serem implementados, prevendo-se todas as variações de cálculo.

O uso de uma notação formal permite que esta seja compreendida pela equipa de desenvolvimento, mas também pelo cliente (companhia de seguros). Numa metodologia de desenvolvimento em que é privilegiado o diálogo com o cliente, esta característica adquire maior relevância. Desta forma, o cliente pode também participar, na definição dos testes necessários para a conformidade do componente. Apresenta-se na Figura 40 a especificação formal das actividades definidas previamente na Figura 38 para o cálculo do imposto dentro das condições.

S0: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{CalculaForaCond}$
S1: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$
S2: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$.
(CalculadorIrs $_{GetEntregas2002}$ | | CalculadorIrs $_{CalculaForaCond}$)
S2.1: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$. CalculadorIrs $_{GetEntregas2002}$
S2.2: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$. CalculadorIrs $_{CalculaForaCond}$
S3: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$. CalculadorIrs $_{GetEntregas2002}$.
(CalculadorIrs $_{GetEntregas5anos}$ | | CalculadorIrs $_{CalculaForaCond}$)
S3.1: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$. CalculadorIrs $_{GetEntregas2002}$.
CalculadorIrs $_{GetEntregas5anos}$
S3.2: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$. CalculadorIrs $_{GetEntregas2002}$.
CalculadorIrs $_{GetEntregas5anos}$. CalculadorIrs $_{CalculaForaCond}$

Notação usada:
CalculadorIrs$_{GetTipoResgate}$:
Calculador - objecto
$_{GetTipoResgate}$ - método
. - Ligação lógica dependente entre objectos/métodos
| | - Ligação lógica opcional entre objectos/métodos
[ ] - Referência a regra

Figura 40 - Sequências de cálculo dentro das condições

De acordo com o diagrama da Figura 38, o cálculo de imposto pode sair fora das condições para as sequências: S0, S2 e S3. Representando essas sequências possíveis por [ForaCondidições], pode-se ainda derivar as actividades definidas previamente na Figura 39, da seguinte forma (Figura 41):

S4: [ForaCondidições] . CalculadorIrs $_{DentroRegra35}$
S5: [ForaCondidições] . CalculadorIrs $_{DentroRegra35}$. Apolice $_{GetDataInicioApolice}$

Notação usada:
CalculadorIrs$_{GetTipoResgate}$:
Calculador - objecto
$_{GetTipoResgate}$ - método
. - Ligação lógica dependente entre objectos/métodos
[ForaCondições]- inclui as possíveis sequências resgate fora das condições

Figura 41 - Sequências de cálculo dentro das condições

Identificadas todas as sequências possíveis para o apuramento das diferentes taxas de imposto, a que um resgate está sujeito num produto PPRE, a próxima fase consiste na definição dos testes, em que essas sequências serão aplicadas. Assim, na Tabela 4 são apresentadas as condições previstas no cálculo de imposto.

Tabela 4 - Condições de testes

Condição	#Código
Resgata dentro das condições, quando dentro 35%, e motivo válido	#C1
Resgata dentro das condições ,quando fora 35%, entregas são anteriores à data lei e motivo válido	#C2
Resgata dentro das condições quando entregas são têm mais de cinco anos à data de resgate e motivo válido	#C3
Resgata fora das condições quando motivo inválido	#C4
Resgata fora das condições quando motivo válido mas fora dos 35%, e entregas posteriores à data lei e sem entregas à menos de cinco anos	#C5
Resgata fora das condições quando motivo válido mas fora dos 35% e entregas posteriores à data lei	#C6
Resgata fora das condições a 8% se dentro dos 35% e idade contrato > 8 anos	#C7
Resgata fora das condições a 16% se dentro dos 35% e idade contrato entre 5 e 8 anos	#C8
Resgata a 20 % se fora das condições e idade contrato < cinco anos	#C9
Resgata a 20 % se fora das condições e fora dos 35%	#C10
Verificando-se as condições em [#C2], parte do valor resgatado excede o montante das entregas com mais de cinco anos	#C11
Verificando-se as condições em [#C3], parte do valor resgatado excede o montante das entregas à data lei	#C12

Estas condições esperadas são definidas em conjunto com o cliente, e representam o comportamento esperado para o sistema de cálculo, com todas as situações previstas. Com as condições aqui descritas, é possível estabelecer a relação com as sequências possíveis de cálculo já identificadas (Figura 40 e Figura 41), e assim definir os testes unitários necessários para o cálculo de IRS. Na Tabela 5, identificam-se quais as sequências que permitem a validação de cada uma das condições.

Tabela 5 - Associação de condições e sequências de testes

Condições		Sequências que permitem validar a condição	*#Test case*
#C1	Resgata a 4 %	S1	TEST_0001
#C2	Resgata a 4 %	S1, S2.1	TEST_0002
#C3	Resgata a 4 %	S1, S3.1	TEST_0003
#C4	Resgata fora condições	S0	TEST_0004
#C5	Resgata fora condições	S1, S3.2	TEST_0005
#C6	Resgata fora condições	S1, S2.2	TEST_0006
#C7	Resgata fora cond. a 8%	[ForaCond], S4, S5	TEST_0007
#C8	Resgata fora cond. a 16%	[ForaCond], S4, S5	TEST_0008
#C9	Resgata fora cond. a 20%	[ForaCond], S4, S5	TEST_0009
#C10	Resgata fora cond. a 20%	[ForaCond], S4	TEST_0010
#C11	Resgata a 4 % e fora cond.	S1, S2.1, S4	TEST_0011
#C12	Resgata a 4 % e fora cond.	S1, S3.1, S4	TEST_0012

Nesta tabela, a coluna *#Teste case*, contém o identificativo para o respectivo *Junit* que o vai implementar. Todos estes *Junit* são apresentados no Anexo C. No entanto, para melhor se compreender a implementação do *Junit*, é explicado a seguir a implementação do teste unitário *TEST_0001*.

Como já referido, o *Junit* é uma ferramenta de auxílio ao desenvolvimento de testes unitários. A execução do teste consiste em: (1) inicializar as variáveis necessárias à execução de cada teste unitário através do método *setUp()*; (2) executar o teste unitário; (3) remover todos os objectos e variáveis que foram necessários à execução do teste unitário.

O método *setUp()* deve inicializar as variáveis que são necessárias ao cálculo de imposto. Para isso, é criado um XML com o mesmo nome do teste unitário que será carregado antes da sua execução, inicializando o objecto de acordo com o objectivo do teste a realizar. Por exemplo, se o conjunto de testes que se está a desenvolver se chamar *TestCaseCalculoIrs*, e os testes unitários se chamarem *test_0001(), test_0002(), ..., test_000n(),* os XML que serão carregados no *setUp()* devem ser nomeados com *TestCaseCalculoIrs_test_000n.xml*. Na Figura 42, é apresentado o XML com a informação necessária à execução do *TEST_0001*. Neste XML são construídas duas *tags*

com objectivos distintos, em que: a *tag* <*inputs*> é onde está a informação de *input* para o teste; a *tag* <*results*> é onde constam os resultados esperados após a execução do teste.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by Nuno Silva (I2S, SA) -->
<TestData>
  <inputs ambiente="GIV450" modalidade="00" versao="00" proposta="00000"
      apolice="00000" nord="00000" cobertura="00" data_inicio="19941216"
       tipo_provisao="C"
     data_resgate="20050528" tipo_interno_sin="01"
      tipo_resgate="R"
      valor_resgate="1800"
      valor_indmax="5700.00"

      total_entregas_8a="0"
      total_entregas_5a="0"
      total_entregas_2002="0"
      total_entregas_MC="3500"
      total_entregas="5000"
      saldo_8a="0"
      saldo_5a="0"
      saldo_2002="0"
      saldo="5760.03"
      />
  <results
      taxa_out="CAT04"
      />
</TestData>
```

Figura 42 - TestCaseCalculoIrs_test_0001.xml

De acordo com a informação constante no XML, através da qual, se inicializaram os objectos necessários ao cálculo do imposto, é possível executar as sequências de acção identificadas para cada teste de uma forma controlada. Ou seja, é conhecido o comportamento esperado dos objectos no contexto das inicializações efectuadas. Assumindo estes pressupostos, pode-se validar o resultado da execução de um teste, conhecendo-se previamente o resultado esperado.

Conforme apresentado na Tabela 5, o *TEST_0001*, implementa a sequência S1 (S1: CalculadorIrs $_{GetTipoResgate}$. CalculadorIrs $_{Regra35}$), pelo que, o teste unitário deve implementar esta sequência de acções. A Figura 43 apresenta a respectiva implementação.

```
    //test_0001
    //Resgata dentro das condições, quando dentro 35%, e motivo válido
        public void test_0001()
          throws
            FileNotFoundException,
            Xml2ObjProcessingException,
            UnableToLoadException {

       if(calculadorIrs.getTipoResgate()=='R'){
         if(regra35()){
           // Resultados esperados
            String taxaResultado=getResultAsString("taxa_out");
            assertTrue("A taxa de output não está dentro das condições previstas de resgate.
         ",hasTaxa(taxaResultado));
         }
       }
}
```

Figura 43 - TestCaseCalculoIrs.test_0001()

Neste código é possível verificar que o teste só é válido se o tipo de resgate for válido (tipo de resgate = 'R'), se a apólice respeita a regra dos 35%, e se o resultado do cálculo estiver de acordo com o resultado esperado, definido no XML (Figura 42).

Os restantes testes seguem todos esta ordem de processos, e a sua implementação pode ser consultada no Anexo C. Em caso de falha na execução dos testes, são indicados os testes que falharam e os respectivos motivos. A Figura 44 mostra o écran com o resultado da execução dos testes aqui especificados sobre o cálculo de imposto.

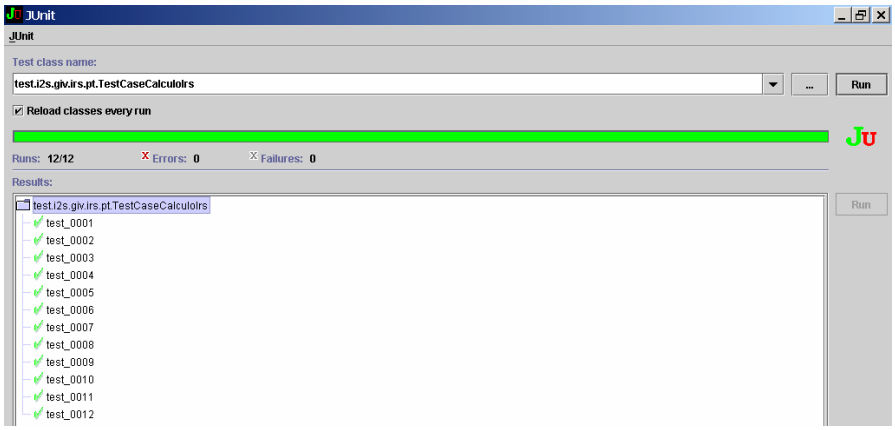


Figura 44 - Resultado da execução dos testes

4.4. O Novo Módulo de Fiscalidade

O desenvolvimento deste módulo, que resultou de um novo requisito originado por novo decreto de lei, teve como principais objectivos a rápida disponibilização da funcionalidade ao cliente e não provocar modificações indesejadas ao comportamento do sistema. Os novos desenvolvimentos, no âmbito do módulo fiscalidade, foram realizados em código Java, sendo que as intervenções realizadas sobre o código RPG tiveram como único objectivo a refabricação do código para permitir a integração do novo componente. Este módulo foi desenvolvido, na sua essência, em Java, uma linguagem diferente daquela em que os componentes do GIS estão ainda na sua maioria implementados (a maioria dos processos estão ainda implementados em RPG, mas com tendência para a redução do seu peso em favor do novo código desenvolvido em Java).

Não sendo o objectivo desta dissertação argumentar qual a razão da adopção tecnológica feita pela I2S em relação ao Java, fica no entanto o princípio base que sustenta essa orientação assumida pela I2S: é uma linguagem OO, fácil de codificar, é multi-plataforma, é suportada pelos parceiros da I2S, e actualmente existem enúmeras ferramentas de suporte à linguagem ao nível de codificação, refabricação, testes, documentação e instalação.

A implementação do módulo fiscalidade foi realizada segundo as fases identificadas no capítulo três, como: recolha de histórias do utilizador, desenho para validação, escrita de testes (como modo de documentação), codificação com padrões e refabricação. As histórias do utilizador resultaram da análise e interpretação do respectivo decreto de lei, mas também houve a necessidade de realizar reuniões com os clientes, no sentido de clarificar diferentes interpretações decorrentes da análise desse mesmo decreto.

A primeira vantagem desta metodologia de desenvolvimento é que, quando a distribuição deste componente se torna oficial e se instala no cliente, há uma elevada

percentagem de confiança em que o produto instalado satisfaz efectivamente os propósitos do cliente.

A segunda vantagem está relacionada com a possibilidade de ter presente as especificidades de negócio de cada cliente resultante da sua participação no processo de desenvolvimento. Neste *case study* as especificidades encontradas estiveram principalmente relacionadas com a forma como são tratadas as transferências de contratos entre companhias. Nas transferências de contratos, as companhias têm modos diferentes de tratar a informação associada ao contrato oriundo de uma congénere. E entre essas informações estão os prémios pagos. Verifica-se que os clientes usam diferentes estruturas do GIS para armazenar essa informação, havendo a necessidade de flexibilizar o sistema, de forma a prepará-lo para essas circunstâncias.

Uma terceira vantagem foi a rapidez com que foi disponibilizado ao cliente o novo sistema de cálculo. A redução do tempo que seria gasto na fase de inicial do projecto com a análise do sistema, permite atingir o objectivo de disponibilizar a funcionalidade em tempo útil ao cliente. A filosofia ágil substitui essa fase prévia de análise do sistema por reuniões com o cliente, permitindo ajustar o projecto às suas reais necessidades, ainda mesmo durante a fase de desenvolvimento.

Evolução do sistema

Depois de concluído este projecto com a sua instalação em vários clientes da I2S, o componente de cálculo de imposto ficou dotado com características que permitem a fácil inclusão de novas regras de cálculo de imposto para qualquer produto. O trabalho realizado permite agora uma fácil migração de outras regras de cálculo implementadas em RPG para Java. Os padrões adoptados são facilitadores da evolutibilidade do sistema, permitindo a inclusão de novas regras de cálculo de IRS com o mínimo esforço, e com pouco ou nenhum impacto para o GIS. Por outro lado, é potencializada a reutilização dos componentes já desenvolvidos. A estrutura de testes pode também ser reutilizada, à

excepção dos testes especificados sobre a camada de domínio, uma vez que a lógica de negócio, em princípio, será diferente.

4.5. Conclusão

No contexto de uma abordagem de desenvolvimento ágil a primeira fase foi especificar as regras contidas no decreto de lei que regulamenta os resgates de PPR/E. Depois de recolhidas as histórias, foi analisado o sistema de sinistros e modelou-se os requisitos da nova funcionalidade. Especificaram-se depois os testes necessários à conformidade das regras de negócio, seguindo-se depois a codificação do novo componente com recurso a padrões de desenho, e com a aplicação de refabricações para permitir a reutilização dos componentes já existentes.

O uso das técnicas e metodologia propostas mostraram-se apropriadas na implementação dos *case study* em análise porque:

(1) A refabricação permitiu a reutilização de componentes existentes.

(2) A especificação prévia dos testes mostrou-se útil para validação prévia do sistema junto do cliente, documentando-se assim (sob a forma de especificação de testes) os requisitos para a implementação do novo cálculo.

(3) A utilização de padrões de desenho quer no desenvolvimento de novos componentes, quer na refabricação dos que já existiam, permitiu desde logo garantir a escalabilidade do sistema para futuras alterações.

(4) Garantiu-se a disponibilização do módulo fiscalidade, em tempo útil para o cliente.

5. Conclusões

5.1. Síntese do trabalho realizado

A elevada agressividade do mundo empresarial obriga as empresas de software a reverem as suas estratégias de desenvolvimento para se adaptarem às reais necessidades dos seus clientes. Essa adaptação tem como objectivo conferir agilidade às aplicações quer para a fácil absorção de novos requisitos que surjam, quer para o rápido desenvolvimento de novas funcionalidades. Esta evolução ganha maior complexidade quando o objecto de evolutibilidade são sistemas legados.

Nesse sentido, os objectivos assumidos desta dissertação tiveram o intuito de estabelecer um fio condutor na refabricação de software para o rejuvenescimento de aplicações. Tendo por base esse princípio, o autor desta dissertação tem a percepção de que o resultado deste trabalho é um contributo válido na tentativa de explorar um problema muito complexo e vasto que atravessa longitudinalmente diferentes áreas da engenharia de software.

É importante considerar que a necessidade da refabricação analisada nesta dissertação incidiu na modificação da estrutura do código existente em RPG como fase preparatória para a evolutibilidade no suporte à integração de novos componentes. No entanto, verifica-se que os estudos realizados e publicados são, na sua esmagadora

maioria, orientados à refabricação aplicada sobre linguagens *Object Oriented*. A importância que a implementação em RPG ainda representa hoje em dia é inquestionável, principalmente devido à sua forte presença nos sistemas financeiros.

Nesse sentido, esta dissertação propõe um processo para a agilização dos sistemas de informação, preparando a base para a sua futura evolutibilidade e rejuvenescimento.

5.2. Trabalho futuro

Os propósitos assumidos desta dissertação foram na sua generalidade alcançados. Fica, no entanto a ideia que dada a abrangência das questões que envolvem o rejuvenescimento de aplicações e que foram sendo aprofundadas com o decorrer desta dissertação, a "fórmula" para o rejuvenescimento de aplicações não se esgota neste trabalho, antes pelo contrário, fica bem perceptível a necessidade de estudar outras metodologias com o propósito de rejuvenescer aplicações, adequando-as à infinita variedade de problemas que se constatam no mundo real dos sistemas de informação.

Nesta dissertação foram dados os primeiros passos na necessidade de associar o conceito de refabricação de software a código legado como o RPG. No entanto, ficam alguns temas em aberto e que merecem um estudo mais aprofundado, dos quais se podem propor alguns tópicos como trabalho futuro:

(1) É assumido neste trabalho que se deve ter a percepção dos padrões existentes e a capacidade de escolher o padrão ajustado para um determinado problema. A base deste conhecimento baseia-se na experiência do indivíduo que está a resolver o problema. Caso esse conhecimento falhe, as iterações de desenvolvimento vão-se arrastar em sucessivas refabricações, para organizar o que foi mal concebido de início. As metodologias ágeis prevêem e assumem esta falha, no entanto seria interessante estudar a possibilidade de padronizar também os problemas de desenvolvimento, e relacioná-los com os padrões conhecidos (soluções), estabelecendo o par problema/solução.

(2) No *case study* desta dissertação é usado o conceito de refabricação no contexto de uma linguagem procedimental como o RPG. Por não ser muito comum esta utilização de conceito (a refabricação está mais ligada ao contexto OO), também seria interessante estudar a possibilidade de redefinir o conceito refabricação no contexto das linguagens procedimentais (como o RPG).

(3) A "fórmula" de rejuvenescimento de aplicações sugerida neste trabalho vai no sentido de preparar um sistema existente para a evolutibilidade, dotando-o de características que o permitam acolher novos requisitos que sejam impostos. No entanto, não é abordada a possibilidade do sistema de informação, ou um determinado módulo desse sistema, estar num nível caótico tão elevado, que seja totalmente inadequada qualquer acção no sentido de o preparar com tais características de evolutibilidade. Assim, é também pertinente investigar um método que permita analisar e avaliar o nível de desorganização de um determinado módulo e, considerando a sua criticidade no sistema, decidir como intervir sobre esse componente.

Bibliografia

[Abrahamsson 2002] Abrahamsson, Pekka, Salo*, et al.*, Agile software development methods. 2002, Julkaisija - Utgivare.

[Alur 2002] Alur, D., Crupi, J. and Malks, D., core J2EE Patterns: Best Practices and Design Strategies. 2002, Sun.

[Astels 2002] Astels, D., Refactoring With UML. 3rd Conference on eXtreme Programming and Agile Processes in Software Engineering, Alghero, Sardinia, Italy,2002.

[Beck 1999] Beck, K., Extreme Programming Explained. 1999, Addison-Wesley Pub Co; 1st edition.

[Beck 2002] Beck, K., Beedle, M., Bennekum, A. and Cockburn, A. (2002). http://agilemanifesto.org/

[Beck 2000] Beck, K. and Fowler, M., Planning Extreme Programming. 2000, Addison Wesley.

[Beck 1998] Beck, K. and Gamma, E. (1998). Test Infected: Programmers Love Writting Tests. http://members.pingnet.ch/gamma/junit.htm

[Benbasat 1999] Benbasat, I. and Zmud, R. W. (1999). "Empirical Research in Information Systems: The practice of Relevance." MIS Quarterly **23**: 3-16.

[Bhabuta 1988] Bhabuta, L., Sustaining productivity and competitive advantage by marshalling IT. IFIP Conference on Information Technology Management for Produtivity and Strategic Advantage, Singapure,1988.

[Boehm 1988] Boehm, B., A Spiral Model of Software Development and Enhancement. IEEE Computer,1988.

[Boehm 2002] Boehm, B. (2002). "Get Ready for Agile Methods, with Care." IEEE Computer **35**(1): 64-69.

[Choudrie 2005] Choudrie, J. and Dwivedi, Y. K. (2005). "Investigating the Research Approaches for Examining Technology Adoption Issues." Journal of Research Practice **1**(1): 12.

[Cinnéide 2000] Cinnéide, M. Ó. (2000). Automated Application of Design Patterns: A Refactoring Approach. Computer Science. Dublin, University of Dublin**:** 242.

[Coulthard 2003] Coulthard, P. and Farr, G. (2003). An RPG-to-J2EE Road Map. iSeries NEWS**:** 3.

[Crispin 2002] Crispin, L., House, T. and Wade, C. (2002). The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment. UPGRADE - The European Magazine for the IT Professional. **III:** 11-17.

[Crystal] Crystal. http://alistair.cockburn.us/crystal/crystal.html

[Davis 1988] Davis, A., Bersoff, E. and Comer, E. (1988). "A Strategy for Comparing Alternative Software Development Life Cycle Models." IEEE Transactions on Software Engineering **14**(10): 1453-1461.

[Deursen 2002] Deursen, A. v., Program Comprehension Risk and Opportunities in Extreme Programming. IEEE-Proceedings of the Eighth Working Conference On Reverse Engineering,2002.

[Elssamadisy 2002] Elssamadisy, A., Recognizing and responding to "bad smells" in extreme programming. 24th International Conference on Software Engineering, Chicago, IL,2002 ACM.

[Evans 2003] Evans, E., Domain-Driven Design - Tackling Complexity in the Heart of Software. 2003, Addison Wesley.

[FDD] FDD. http://www.featuredrivendevelopment.com/

[Foote 1994] Foote, B. and Opdyke, W. F., Life Cycle and Refactoring Patterns that Support Evolution and Reuse. First Conference on Patterns Languages of Programs (PLoP '94), Monticello, Illinois,1994.

[Fowler 1999] Fowler, M., Beck, K., Brant, J., Opdyke, W. F. and don Roberts, Refactoring: Improving the Design of Existing Code. 1999, Addison-Wesley Pub Co; 1st edition.

[Fowler 2003] Fowler, M. and Foemmel, M. (2003). Continuous Integration. http://www.martinfowler.com/articles/continuousIntegration.html

[Fowler 2002] Fowler, M., Rice, D., Foemmel, M.*, et al.*, Patterns of Enterprise Application Architecture. 2002, Addison Wesley.

[Fraikin 2002] Fraikin, F. and Leonhardt, T., SeDiTeC - Testing Based on Sequence Diagrams. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Edingburgh,2002.

[Galliers 1991] Galliers, R. D. and Sutherland, A. R. (1991). "Information systems strategy formularion: The stages of grouth model revisited." Journal of Information Systems: 89-114.

[Gamma 1993] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 93,1993.

[Gamma 1994] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. 1994, Addison-Wesley.

[Grunbacher 2002] Grunbacher, P. and Hofer, C., Complementing XP with Requirements Negotiation. 3rd Conference on eXtreme Programming and Agile Processes in Software Engineering, Alghero, Sardinia, Italy,2002.

[Hall 2001] Hall, M., core Servlets and JavaServer Pages. 2001, SunMicrosystems.

[Harrold 2001] Harrold, M. J., Jones, J. A., Li, T. and Liang, D., Regression Test Selection for Java Software. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001), Tampa, FL, USA,2001.

[Heinecke 2002] Heinecke, H., Noack, C. and Schweizer, D., Constructing Agile Software Processes. 3rd Conference on eXtreme Programming and Agile Processes in Software Engineering, Alghero, Sardinia, Italy,2002.

[HighSmith 2000] HighSmith, H., Adaptative Software Development: A Collaborative Approach to Managing Complex Systems. 2000, Dorset House.

[IBM 1999] IBM, ILE COBOL for AS/400 Programmer's Guide, second edition. 1999.

[IBM 2001] IBM, WebSphere® Development Studio: ILE RPG Programmer’s Guide. 2001, IBM.

[IBM 2002] IBM, WebSphere® Development Studio: ILE RPG Reference. 2002, IBM.

[Johnson 2002] Johnson, M., Designing Enterprise Applications with the J2EETM Platform, Second Edition. 2002, Addison-Wesley.

[Junit 2003] Junit (2003). "http://www.junit.org."

[Kerievsky 2004] Kerievsky, J., Refactoring to Patterns. 2004, Addison-Wesley.

[Labiche 2001] Labiche, Y. and Briand, L., A UML - Based Approach to System Testing. Fourth International Conference on the Unified Modeling Language, Toronto, Canada,2001 IEEE.

[Labiche 2000] Labiche, Y., Thevenod-Fosse, P., Waeselynck, H. and Durand, M., Testing Levels for Object-Oriented Software. 22nd IEEE International Conference on Software Engineering (ICSE), Limerick (Ireland),2000.

[Leffingwell 1999] Leffingwell, D. and Widrig, D., Managing Software Requirements. 1999, Addison Wesley.

[Manzoni 2003] Manzoni, L. V. and Price, R. T. (2003). "Identifying Extensions Required bu RUP to Comply with CMM Levels 2 and 3." IEEE-ITransactions On Software Engineering **29**(2): 181-192.

[Marinescu 2002] Marinescu, F., EJB Design Patterns. 2002, Wiley Computer Publishing.

[McLaughlin 2002] McLaughlin, B., Java and XML Data Binding. 2002, O'Reilly.

[Myers 1997] Myers, M. D. (1997). "Qualitative Research in Information Systems." MIS Quarterly **21**.

[Nolan 1979] Nolan, R. (1979). "Managing the crises in data processing." Harvard Business Review **57**(2): 115-126.

[Opdyke 1992] Opdyke, W. F. (1992). Refactoring Object-Oriented Frameworks. Computer Science, Illinois.

[Parrish 2001] Parrish, A., Jones, J. and Dixon, B., Extreme Unit Testing: Ordering Test Cases to Maximize Early Testing. XP Agile Universe, Raleigh, NC,2001.

[Rational 1997] Rational, UML - Object Constraint Language Specification. 1997.

[Rational 1998] Rational (1998). Rational Unified Process: Best Practices for software Development Teams (www.rational.com), Rational.

[Roberts 1999] Roberts, D. (1999). Practical Analysis for Refactorinng. Computer Science. Illinois, University of Illinois**:** 137.

[Rumbaugh 1998] Rumbaugh, J., Jacobson, I. and Booch, G., The Unified Modeling Language Reference Manual. 1998, Addison Wesley.

[Sharma 2001] Sharma, R. and Stearns, B., J2EE Connector Architecture and Enterprise Application Integration. 2001, Addison Wesley.

[Stotts 2002] Stotts, D., Lindsey, M. and Antley, A., An Informal Formal Method for Systematic JUnit Test Case Generation. Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods,2002.

[Sundft 2002] Sundft, D., Devroy, R., Manfred, E. and Karen, W., i2 Solutions on the IBM. 2002, IBM.

[Tokuda 1999] Tokuda, L. A. (1999). Evolving Object-Oriented Designs with Refactorings. Faculty of the Graduate School. Texas, University of Texas at Austin**:** 144.

[Tsai 1999] Tsai, W.-T., Tu, Y., Shao, W. and Ebner, E., Testing Extensible Design Patterns in Object-Oriented Frameworks through Scenario Templates. 23rd International Computer Software and Applications Conference (COMPSAC '99), IEEE Computer Society, Phoenix, AZ, USA,1999 IEEE Computer Society.

[Wang 1997] Wang, Y., King, G., Fayad, M.*, et al.*, On Built-in Test Reuse in Object-Oriented Framework Design. ESEC/FSE'97 Workshop on Object-Oriented Reengineering, Zurich,1997.

[Wang 1999] Wang, Y., King, G. and Wickburg, H. (1999). "A Method for Built-in Tests in Component-based Software Maintenance." IEEE-International Conference on Software Maintenance and Reengineering (CSMR'99): 186-189.

[Zhu 1997] Zhu, H., Patrick, A. V. and John, H. R. (1997). "Software Unit Test Coverage and Adequacy." ACM Computing Surveys **29**(4).

Anexo A. Regras fiscais para PPR/E

<table>
<tr><td colspan="4">REGRAS DE ENQUADRAMENTO DO REGIME FISCAL</td></tr>
<tr><td colspan="1">Dentro das Condições</td><td colspan="3">Fora das Condições</td></tr>
<tr><td>4%</td><td>8%</td><td>16%</td><td>20%</td></tr>
<tr>
<td><u>Condição Base:</u>
Se motivo de resgate válido:
Reforma/velhice/educação/
desemprego longa dur./
incapacidade perm./
doença grave
ou
sinistro morte

R0:
Se sinistro morte
➔ Resgata tudo a 4%
R1:
Se Cumpre regra 35%
(Entr. até meio contrato é superior ou igual a35% do total das entregas)
➔ Resgata tudo a 4%
(se não cumpre 35%)
R2:
Entregas até 2002/7/2
➔ Resgata a totalidade das entregas anteriores a 2002 a 4%
R3:
Entregas posteriores a 2002/7/2 e com mais de 5 anos
➔ Resgata entregas que satisfaçam esta condição a 4%</td>
<td><u>Condição Base:</u>
R1:
Se Cumpre regra 35%
(Entr. até meio contrato é superior ou igual a35% do total das entregas)
e
R4:
Idade do Contrato Superior a 8 anos

Resgate da totalidade a 8%</td>
<td><u>Condição Base:</u>
R1:
Se Cumpre regra 35%
(Entr. até meio contrato é superior ou igual a35% do total das entregas)
e
R5:
Idade do Contrato entre 5 e 8 anos

Resgate da totalidade a 16%</td>
<td>Tudo o resto é resgatado
A 20 %</td>
</tr>
<tr>
<td><u>R2:
Variáveis para valid dos entregas até 2002:</u>
Ea2002– Entregas até 2002
<u>R3:
Variáveis para valid dos entregas com mais de 5anos:</u>
Ent5A - Entregas com mais de 5 anos
Ea2002– Entregas até 2002
IdadCon– Idade Contrato
Regra:
Ent5A > 0
E IdadeCon > 5</td>
<td><u>R4:
Variáveis para valid da idade do Contrato:</u>
IdadCon– Idade Contrato
Regra:
E IdadeCon > 8</td>
<td><u>R5:
Variáveis para valid da idade do Contrato:</u>
IdadCon– Idade Contrato
Regra:
IdadeCon >= 5 e
IdadeCon <= 8</td>
<td></td>
</tr>
<tr>
<td colspan="3"><u>R1; Variáveis para validação dos 35%:</u>
EMeioC – Entregas Meio Contrato
EntTo – Total de Entregas
Regra: EMeioc >= EntTo * 35%</td>
<td></td>
</tr>
</table>

Anexo B. Valores de balanço

APOLBV - VALORES DE BALANÇO	
Tipo Valor	**Regra**
Entregas > 5 anos (ENT5A) $val=706	Todos os recibos cobrados entre a data início da apólice e a data de resgate com : Data_Resgate – Data_Referencia_Recibo >= 5 anos e Data_Referencia_Recibo > dataLei:2002/07/02 Afectado negativamente com os recibos de estorno do mesmo período
Entregas > 8 anos (ENT8A)	Todos os recibos cobrados entre a data início da apólice e a data de resgate com: Data_Resgate – Data_Referencia_Recibo > = 8 anos Afectado negativamente com os recibos de estorno do mesmo período
Entregas até 2002 (EA2002) $val=709	Todos os recibos cobrados entre a data início da apólice e a data de resgate com : Data_Referencia_Recibo <= 2002/07/02 Afectado negativamente com os recibos de estorno do mesmo período
Entregas até meio contrato (EMEIOC) $val=707	Todos os recibos cobrados entre a data início da apólice e a data de resgate com : Data_Referencia_Recibo < Data_Meio_Contrato Afectado negativamente com os recibos de estorno do mesmo período

Anexo C. Código de testes da fiscalidade

Listagem do código de testes Implementados com Junits:

```
package test.i2s.giv.irs.pt;

/**
  *
 * @author I2S - Nuno Silva
 *
 */
public class TestCaseCalculoIrs extends TestCasesXmlComparesBase {

        CalculoIrs calculadorIrs = null;
        HashMap resultados = null;

        /**
         * @see TestCase#tearDown()
         */
        protected void setUp() throws Exception {
        loadXML();

          Ambiente amb = null;
          Moeda moeda = null;

          int modalidade = getInputAsInt("modalidade");
          int versao = getInputAsInt("versao");
          int apolice = getInputAsInt("apolice");
          int proposta = getInputAsInt("proposta");
          int nord = getInputAsInt("nord");
          int cobertura = getInputAsInt("cobertura");
          Data data_inico_apl = new Data(Integer.parseInt(getInputAsString("data_inicio")));

          char tipoProvisao = getInputAsString("tipo_provisao").charAt(0);
          double valorResgate = Double.parseDouble(getInputAsString("valor_resgate"));

          double valorIndMax = Double.parseDouble(getInputAsString("valor_indmax"));
          Data data_resgate = new Data(Integer.parseInt(getInputAsString("data_resgate")));
          int tipo_interno_sinistro = getInputAsInt("tipo_interno_sin");
          char tipo_resgate = getInputAsString("tipo_resgate").charAt(0);

          double total_entregas = Double.parseDouble(getInputAsString("total_entregas"));
          double total_entregas_5a =
            Double.parseDouble(getInputAsString("total_entregas_5a"));
          double total_entregas_2002 =
            Double.parseDouble(getInputAsString("total_entregas_2002"));
          double total_entregas_MC =
            Double.parseDouble(getInputAsString("total_entregas_MC"));

          double saldo_5a = Double.parseDouble(getInputAsString("saldo_5a"));
          double saldo_2002 = Double.parseDouble(getInputAsString("saldo_2002"));
          double saldo = Double.parseDouble(getInputAsString("saldo"));
```

```
            calculadorIrs =
              new CalculoIrs(
                amb, modalidade, versao, apolice, proposta, nord, cobertura,
                data_inico_apl, moeda, tipoProvisao,
                valorResgate, valorIndMax, data_resgate, tipo_resgate, tipo_interno_sinistro,
                total_entregas,total_entregas_5a, total_entregas_MC, total_entregas_2002,
                saldo, saldo_5a, saldo_2002);
            resultados = calculadorIrs.decomposicaoIRS();
          }



          /**
           * @see TestCase#tearDown()
           */
          protected void tearDown() throws Exception {
            super.tearDown();
          }
          /**
           * @see main(String[] args)
           */

          public static void main(String[] args) {
            SetupI2SSystem4Tests.setup(args);
            junit.textui.TestRunner.run(TestCaseCalculoIrs.class);
            SetupI2SSystem4Tests.closeDown();
          }
          /**
           * Constructor for TestCaseCalculoIrs.
           * @param arg0
           */
          public TestCaseCalculoIrs(String arg0) {
            super(arg0);
          }
          private static final char TIPO_RESGATE_CONST_R = 'R';

        //test_0001
        //Resgata dentro das condições, quando dentro 35%, e motivo válido
          public void test_0001()
            throws
              FileNotFoundException,
              Xml2ObjProcessingException,
              UnableToLoadException {

         if(calculadorIrs.getTipoResgate()=='R'){
          if(regra35()){
            // Resultados esperados
             String taxaResultado=getResultAsString("taxa_out");
             assertTrue("A taxa de output não está dentro das condições previstas de resgate.
          ",hasTaxa(taxaResultado));
          }
         }
  }

        //test_0002
```

```
//Resgata dentro das condições ,quando fora 35%,  entregas são anteriores à data lei e
    motivo válido
public void test_0002()
      throws
            FileNotFoundException,
            Xml2ObjProcessingException,
            UnableToLoadException{
 if(calculadorIrs.getTipoResgate()=='R'){
  if(!regra35()){
   if(calculadorIrs.getEntregas2002()>0){
    String taxaResultado=getResultAsString("taxa_out");
    assertTrue("A taxa de output não está dentro das condições previstas de resgate.
   ",hasTaxa(taxaResultado));
   }
  }
 }
}

//test_0003
//Resgata dentro das condições quando entregas são têm mais de 5 anos à data de resgate
    e motivo válido
public void test_0003()
      throws
            FileNotFoundException,
            Xml2ObjProcessingException,
            UnableToLoadException {
 if(calculadorIrs.getTipoResgate()=='R')
 {
  if(!regra35()){
   if(calculadorIrs.getEntregas5Anos()>0){
    String taxaResultado=getResultAsString("taxa_out");
    assertTrue("A taxa de output não está dentro das condições previstas de resgate.
   ",hasTaxa(taxaResultado));
   }
  }
 }
}

//test_0004
//Resgata fora das condições quando motivo inválido
public void test_0004()
      throws
            FileNotFoundException,
            Xml2ObjProcessingException,
            UnableToLoadException {
 if(calculadorIrs.getTipoResgate()!='R')
 {
  String taxaResultado=getResultAsString("taxa_out");
  assertTrue("Resgatou indevidamente dentro das condições previstas de resgate!
   ",hasTaxa(taxaResultado));
 }
}

//test_0005
//Resgata fora das condições quando motivo válido mas fora dos 35%,
// e entregas posteriores à data lei e sem entregas à menos de 5 anos
```

```
public void test_0005()
        throws
                FileNotFoundException,
                Xml2ObjProcessingException,
                UnableToLoadException {
  if(calculadorIrs.getTipoResgate()=='R')
  {
    if(!regra35()){
      if(calculadorIrs.getEntregas2002()<=0 &&
         calculadorIrs.getEntregas5Anos()<=0){
            assertFalse("Resgatou indevidamente dentro das condições previstas de resgate.
     ",hasTaxa(CalculoIrs.CONST_ID_CAT4));
      }
    }
  }
}

//test_0006
//Resgata fora das condições quando motivo válido mas fora dos 35%
// e entregas posteriores à data lei
//Nota: Este não sobrepões ao anterior para o sistema deduz às entregas com > 5anos
//as entregas realizadas até à data lei (2002)
public void test_0006()
        throws
                FileNotFoundException,
                Xml2ObjProcessingException,
                UnableToLoadException{
  if(calculadorIrs.getTipoResgate()=='R'){
    if(!regra35()){
      if(calculadorIrs.getEntregas2002()<=0){
           assertFalse("Resgatou indevidamente dentro das condições previstas de
     resgate.",hasTaxa(CalculoIrs.CONST_ID_CAT4));
      }
    }
  }
}

//test_0007
//Resgata fora das condições a 8% se dentro dos 35% e idade contrato > 8 anos
public void test_0007()
        throws
                FileNotFoundException,
                Xml2ObjProcessingException,
                UnableToLoadException{
  if(isForaCondicoes())
  {
    if(regra35()){
      if(calculadorIrs.getDataResgate().
        subDurYear(calculadorIrs.getDataInicioContrato())>8)
      {
           String taxaResultado=getResultAsString("taxa_out");
           assertTrue("Não resgatou à taxa prevista. ",hasTaxa(taxaResultado));
      }
    }
  }
}
```

```
//test_0008
//Resgata fora das condições a 16% se dentro dos 35% e idade contrato entre 5 e 8 anos
public void test_0008()
       throws
              FileNotFoundException,
              Xml2ObjProcessingException,
              UnableToLoadException{
  if(isForaCondicoes())
  {
    if(regra35()){
      int idade = calculadorIrs.getDataResgate().
                subDurYear(calculadorIrs.getDataInicioContrato());
      if(idade >=5 && idade <=8 ){
        String taxaResultado=getResultAsString("taxa_out");
        assertTrue("Não resgatou à taxa prevista. ",hasTaxa(taxaResultado));
      }
    }
  }
}

//test_0009
//Resgata a 20 % se fora das condições e idade contrato < 5 anos
public void test_0009()
       throws
              FileNotFoundException,
              Xml2ObjProcessingException,
              UnableToLoadException {
  if(isForaCondicoes())
  {
    if(regra35()){
      int idade = calculadorIrs.getDataResgate().
                subDurYear(calculadorIrs.getDataInicioContrato());
      if(idade < 5){
        String taxaResultado=getResultAsString("taxa_out");
        assertTrue("Não resgatou à taxa prevista. ",hasTaxa(taxaResultado));
      }
    }
  }
}

//test_0010
//Resgata a 20 % se fora das condições e fora dos 35%
public void test_0010()
       throws
              FileNotFoundException,
              Xml2ObjProcessingException,
              UnableToLoadException {
  if(isForaCondicoes()){
    if(!regra35()){
          assertTrue(hasTaxa(CalculoIrs.CONST_ID_CAT20));
    }
  }
}

//test_0011
```

```
//Verificando-se as condições em  [#C2], parte do valor resgatado excede
// o montante das entregas com mais de 5 anos
public void test_0011()
        throws
                FileNotFoundException,
                Xml2ObjProcessingException,
                UnableToLoadException{
  if(calculadorIrs.getTipoResgate()=='R'){
    if(!regra35()){
      if(calculadorIrs.getEntregas2002()>0){
        String taxaResultado=getResultAsString("taxa_out1");
        assertTrue("A taxa de output não está dentro das condições previstas de resgate.
     ",hasTaxa(taxaResultado));
        String taxaResultado2=getResultAsString("taxa_out2");
        assertTrue("A taxa de output não está dentro das condições previstas de resgate.
     ",hasTaxa(taxaResultado2));
      }
    }
  }
}

//test_0012
//Verificando-se as condições em  [#C3], parte do valor resgatado excede
// o montante das entregas à data lei
public void test_0012()
        throws
                FileNotFoundException,
                Xml2ObjProcessingException,
                UnableToLoadException {
  if(calculadorIrs.getTipoResgate()=='R'){
    if(!regra35()){
      if(calculadorIrs.getEntregas5Anos()>0){
        String taxaResultado=getResultAsString("taxa_out2");
        assertTrue("A taxa de output não está dentro das condições previstas de resgate.
     ",hasTaxa(taxaResultado));
        String taxaResultado2=getResultAsString("taxa_out1");
        assertTrue("A taxa de output não está dentro das condições previstas de resgate.
     ",hasTaxa(taxaResultado2));
      }
   }
  }
}

// hasTaxa(String taxa)
// Verifica so resultado um cálculo de devolveu a taxa passada por parm
    private boolean hasTaxa(String taxa) {
        new StringBuffer("\n has taxa:" + taxa);*/
      Iterator iter = resultados.keySet().iterator();
      while (iter.hasNext()) {
        // categoria
        String categoria = (String) iter.next();
        ArrayList valoresCat = (ArrayList) resultados.get(categoria);

        Double valorCATCAP = (Double) valoresCat.get(CalculoIrs.CONST_ID_CAPITAL);
        Double valorCATIMP = (Double) valoresCat.get(CalculoIrs.CONST_ID_IMPOSTO);
        if (categoria.compareTo(taxa) == 0 && valorCATIMP.doubleValue() > 0) {
```

```
                    return true;
                }
              }
              return false;
            }
        // regra35()
        // verifica regra dos 35% -> total de entregas até meio contrato > total entregas
        private boolean regra35(){
          return (calculadorIrs.getEntregasMeioContrato()>=
              calculadorIrs.getTotalEntregas()*0.35);
    }

        // isForaCondicoes()
        // verifica se resgate sai for a das condições
        private boolean isForaCondicoes(){
          return(
            calculadorIrs.getTipoResgate()!='R' ||
            (!regra35() && calculadorIrs.getEntregas2002()<=0 &&
               calculadorIrs.getEntregas5Anos()<=0));
        }
}
```