Filipe Jorge da Silva Araújo

[Licenciatura em Engenharia Informática Bolonha]

Reconfiguração Dinâmica Estruturada de *Workflows* de Serviços Web

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadora: Prof[a]. Doutora Maria Cecília Gomes
Co-orientador: Prof. Doutor Hervé Paulino

Júri:

Presidente: Prof. Doutor António Ravara
Arguentes: Prof[a]. Doutora Dulce Domingos
Vogais: Prof[a]. Doutora Maria Cecília Gomes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2012

Reconfiguração Dinâmica Estruturada de *Workflows* de Serviços Web

Agradecimentos

Esta tese não representa só o culminar de uma fase da minha vida académica, representa também o acreditar, o querer e a vontade em atingir patamares mais altos. Foi um grande desafio que sem as pessoas que a seguir vou enumerar não seria possível.

Em primeiro lugar gostaria de agradecer aos meus orientadores, *Professora Maria Cecília Gomes* e *Professor Hervé Paulino*, pela ajuda que me deram em muitas fases complicadas desta tese, em especial a *Professora Maria Cecília Gomes* pelo esforço incansável e por ter sempre acreditado em mim.

Agradeço à empresa *Quidgest*, por me ter disponibilizado tempo e espaço para terminar esta tese, mas também por me ter ajudado a crescer profissionalmente. Em particular, ao *João Paulo*, a *Cristina* e a todos os colaboradores que passaram pela grande equipa de gestão documental (*Beatriz*, *Joana*, *Tiago V.*, *Sérgio F.* e *João H.*). Um obrigado a todos.

Um agradecimento ao *Miguel M.*, por me ter acompanhado numa parte importante do meu percurso académico, mas também pela amizade e momentos de distracção criados.

A Departamento de Informática da FCT que me concedeu uma bolsa de investigação para me focar unicamente neste objectivo. A comunidade *Activiti* que me ajudou a resolver muitos problemas durante a fase de desenvolvimento.

Não queria deixar de agradecer a minha *mãe* por me ter sempre apoiado, por me ter dado forças em muitos momentos difíceis e por ter me ajudado a ser a pessoa que sou hoje. Ao meu *irmão João* por ter sido aquele amigo que precisava em muitas fases da minha vida. Obrigado pelas maluquices e por seres o meu *"Soul brother"*. Um obrigado ao meu *pai*, *irmão António*, e *avôs António* e *José* pela sua contribuição neste processo tão moroso.

Um agradecimento muito especial a *Cláudia*, pelo companheirismo, amor e ternura que me nunca faltaram. Muito obrigado.

Obrigado a todos a quem não mencionei mas que também fizeram parte desta aventura.

Por último, mas não menos importantes, um agradecimento a todos os meus amigos de quatro patas que passaram na minha vida: A *Esmiga*, *Garfi*, *Xina*, *Xinkas*, *Bolinhas*, *pérola(s)* e *Mikas*.

Resumo

A complexidade crescente dos sistemas distribuídos baseados em serviços requer, cada vez mais, suporte para dinamismo e automatismo, permitindo assim construir sistemas evolutivos mais versáteis e perduráveis. Alguns exemplos são, adição dinâmica de serviços, modificação dinâmica de dependências entre serviços ou substituição dinâmica de serviços de forma a satisfazer novos requisitos da aplicação. Adicionalmente, com aumento da complexidade dos sistemas distribuídos, por exemplo aplicações em larga de escala (caso particular de *Cloud computing*), a necessidade de existir coordenação e estruturação entre diferentes serviços web, tornou a composição de serviços essencial. Para esse efeito são necessários *standards* para expressar e coordenar essa composição, dos quais os sistemas de gestão de *workflows* (*WfMs*) são um exemplo. Neste seguimento surgem os *workflows* de serviços, que permitem criar fluxos de trabalho que automaticamente escolhem, compõem e interoperabilizam vários serviços para atingir um objectivo. Consequentemente, as referidas adaptações necessárias demonstram-se também necessárias para a adaptação/modificação destes tipos de *workflows*. A existência de flexibilidade, controlada/não controlada pelo utilizador, é necessária quer na interacção com os serviços, quer na sua agregação/composição.

Nesta medida, o objectivo desta dissertação é disponibilizar, através de uma ferramenta *WfMs*, mecanismos de reconfiguração dinâmica num ambiente de composição de serviços. Para atingir este objectivo a solução teve como base padrões que capturam estruturas e comportamentos recorrentes no processo de composição e interacção entre serviços, assim como da sua reconfiguração dinâmica. O protótipo implementado instancia estes conceitos no âmbito de uma ferramenta de modelação e execução de processos em notação *BPMN*, o Activiti. A validação da proposta foi efectuada através de 3 casos de uso nas áreas de negócio e científica.

Palavras-chave: Coordenação, Composição, Serviço, *Workflow*, Padrões, Reconfiguração Dinâmica ...

Abstract

The importance of distribuited applications which interact with multiple services, mutually dependent, has showed that computational systems should ensure dynamism and automatism in their execution. In this sense, the demand for novel support tools and environments providing dynamic reconfiguration features is intensifying.

The emergence of new requirements or new computing paradigms requires systems to be prepared to adapt, thereby constructing evolutionary systems more versatile and enduring. For instance, dynamic inclusion of novel services, dynamic modification of dependencies among services or dynamic service replacement in order to satisfy new application requirements. The increasing complexity of distributed systems, such as large scale applications (for example, Grid e Cloud computing), and the need of coordination and structuring between different web services showed that the composition of services is essential. For this purpose, standards are needed to express and coordinate composition, for example, workflows managment systems (WfMs). Following that standard, the concept of workflows of services was purpose. This supports automatic workflows that chooses and composes services with interoperability, to achieve a goal. In addition, adaptation/modification in WfMs is also needed, such as flexibility in both interaction and composition of services.

Therefore, the goal of this thesis is to provide, with a WfMs tool, dynamic reconfiguration mechanisms on a domain of services composition. To achieve such goal, the solution are based in the pattern concept as a way of capturing recurrent structures and behaviours which specify the evolution of the adaptable system. The implemented prototype instantiates these concepts within a modeling and execution tool of processes in *BPMN* notation, the Activiti. The validation of the proposal was made on three use cases in the areas of business and science.

Keywords: Coordination, Composition, Service, State, Workflow, Patterns, Dynamic Reconfiguration ...

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

1.1 Contexto e Motivação

Nos dias de hoje, a partilha de informação tornou-se fulcral para o desenvolvimento de sistemas, tanto para empresas como para a área de investigação. A necessidade de integração e interoperabilidade entre sistemas diferentes levou assim, à definição de métodos *standard* e directivas a seguir no seu processo de criação. Neste contexto, surgiu o conceito de computação orientada a serviços que, através da abstracção de serviço e mecanismos necessários à sua disponibilização, permite suportar a interacção e interoperabilidade referidas. A arquitectura SOA é um dos exemplos *standard* visto que fornece normas para a construção de serviços, os quais representam o acesso a uma (ou mais) funcionalidades de um sistema externo, encapsulando-o de forma a esconder todos os detalhes aos clientes do serviço. Adicionalmente, cada serviço tem associadas políticas de acesso que têm de ser cumpridas pelos sistemas cliente.

Actualmente, os serviços *Web* (*Web Services*) são a aproximação ao SOA mais comum e conhecida, sendo o meio empresarial um caso de sucesso do seu uso. Contudo, um dos requisitos que tem vindo a aumentar em importância é a necessidade de compor vários serviços para atingir um resultado. Com o aumento de serviços disponíveis a custo acessível, onde se incluem os serviços na *Cloud*, torna-se rentável desenvolver novas aplicações como resultado da reutilização e composição de serviços existentes.

De facto, dada a simplicidade do conceito de serviço e dos mecanismos *standard* disponíveis, pode observar-se uma tendência crescente de disponibilizar *tudo como um serviço* (*XaaS – everything as a service*. Os exemplos vão desde a) a área de *redes de sensores sem fios* (*Wireless Sensor Networks – WSNs*), onde o conceito de serviço permite disponibilizar uma interface de mais alto nível simplificando a recolha de dados e parametrização deste

tipo de redes [PS11]; b) o domínio da *Internet das Coisas* (*Internet of Things – IoT*) [AIM10], onde o conceito de serviço permite integrar diferentes tipos de entidades e objectos desse domínio, quer reais quer virtuais, em contextos inteligentes; até c) serviços no contexto de computação em *Cloud* [MG09], nas suas diferentes dimensões, e.g. *Infrastructure as Service*, *Platform as a Service*, ou *Software as a Service*).

Por outro lado, a larga aceitação do modelo de serviços resulta também do modo usual de interacção pedido/resposta sem estado, e.g. presente nos serviços Web na sua forma primitiva, que permite um desacoplamento entre o serviço e o seu acesso por parte de vários clientes. No entanto, a noção de *serviços com estado*, no sentido que representa o acesso/obtenção de informação sobre actividades continuadas no tempo e recursos com estado, tem merecido uma atenção acrescida, sendo inclusivamente usada nos exemplos referidos.

Por exemplo, a submissão de tarefas de longa duração, como é o caso de alguns serviços *Cloud computing* e *Grid computing*, requer que os clientes possam interrogar o serviço sobre o estado dessas tarefas. De igual modo, aplicações distribuídas de longa duração, e.g. aplicações *web* cujo o prestígio depende da sua continua disponibilidade, requerem mecanismos de suporte à manutenção do seu estado. É também comum existirem entidades na IoT representadas através de serviços com estado, dado que muitas entidades pertencem a um único dono (e.g. dispositivos electrónicos em casa de particulares) não sendo necessário contemplar a invocação do serviço por parte de diferentes utilizadores. Já acesso a WSNs requer serviços que permitam a obtenção dos dados recolhidos pelas redes, sem que os utilizadores tenham de estar continuamente a interrogar o serviço para a obtenção de informação.

Dadas as estas características, os exemplos de serviços acima requerem a realização de um estado dinâmico/variável que tem de se manter consistente ao longo da troca de mensagens entre o serviço e cada um dos clientes [Ibm04]. A interacção com estes serviços implica, consequentemente, modelos de interacção mais complexos entre um serviço e os seus utilizadores, para além do tradicional acesso pedido/resposta. Como exemplo temos a subscrição de eventos ou a disseminação de dados baseada em *stream*. São por isso necessários novos modelos e soluções para interacção com este tipo de serviços.

O paradigma de serviços em geral, permite assim uniformizar quer o acesso a, quer a integração de, entidades com características diferentes, e que se encontram a diferentes níveis das infra-estruturas computacionais complexas dos dias de hoje. Por exemplo, o conceito de serviço permite combinar o acesso a dados recolhidos por *WSNs* com a sua disponibilização a aplicações em execução no contexto de *cloud computing*.

Existem diversas formas de realizar a inter-ligação de serviços, como seja a composição de serviços em *mashups* que, por exemplo, usam e combinam informação de várias fontes para criar um novo serviço ou aplicação *Web*. Outro exemplo é através de sistemas de *fluxo de trabalho* (*workflow*), como a seguir se descreve.

1.1.1 Sistemas de Gestão de Workflows

Os sistemas de gestão de *workflows* (WfMS) [vdAvH02] facilitam a especificação, execução, registo, e controlo dinâmico de tarefas, envolvendo múltiplas pessoas e/ou sistemas externos. Os *Business Process Management System (BPMS)*, em particular, não só gerem *workflows* como os integram com outros sistemas e controlam o processo envolvido [KLL09]. Estas ferramentas, específicas da área administrativa/de negócio, possibilitam a representação do fluxo de controlo de várias tarefas num processo, a sua automatização e optimização, com o fim de representar um processo de negócio particular. Uma tarefa pode ser, por exemplo, a invocação de uma aplicação, de um *web service*, mas podem também representar a interacção com pessoas.

As dependências entre tarefas são definidas através de uma da linguagem de especificação, como seja, por exemplo, a notação *Business Process Modeling Notation (BPMN) [OMG]*, e estão estruturadas no próprio *workflow* definindo possíveis caminhos para o fluxo de execução em tempo real. Acontece que vários tipos de dependências são recorrentes em processos em diferentes áreas de negócio, pelo que foram identificados e capturados como *Padrões de Workflow* [dAea]. Os motores de execução para estas ferramentas são, por sua vez, responsáveis por garantir a execução das tarefas de acordo com a especificação do *workflow*, após a sua instanciação no contexto de um processo em particular.

Por seu lado, a especificidade das aplicações nas áreas de ciências e engenharia, requerendo o suporte e manutenção de aplicações complexas e de larga escala, e que apresentem tolerância a falhas, levou ao surgimento de ferramentas de *workflow*, específicas dessas áreas. Tipicamente, a maioria dos processos científicos envolve um número reduzido de pessoas, e um *workflow* é usado para capturar a execução de um conjunto de ferramentas e modelos científicos, havendo transformação e análise dos dados produzidos e sua visualização. Um cientista/engenheiro pode analisar a execução dos vários componentes que constituem um processo, alterar a sua parametrização, reproduzir experimentações usando as mesmas parametrizações em execuções consecutivas, etc. Nestes *workflows*, os processos capturam geralmente a execução e composição de aplicações que geram ou processam grandes volumes de dados, e requerem grandes capacidades computacionais (e.g. simulações na área de meteorologia da astrofísica, etc). O fluxo de execução está assim geralmente relacionado com o fluxo de dados entre as várias tarefas que constituem um processo. Um exemplo característico é a aplicação de uma sequência de filtros a um largo conjunto de dados que constituem uma imagem, para posterior visualização.

Se antigamente as ferramentas de *workflow* nos domínios da ciência e engenharia, não permitiam representar mecanismos de fluxos de controlo mais sofisticados, nem actividades manuais (e.g. acções realizadas por peritos conhecedores do domínio), a complexidade das aplicações de hoje em dia implica a existência de ferramentas com estas características. É assim crescente a necessidade de representar a integração de diversos

sistemas e ferramentas, e a cooperação entre peritos, de diferentes áreas (e.g. matemática, biologia, química, biofísica, etc.). Tal é facilitado pelo uso de linguagens *standard* de especificação de *workflows*, e suas ferramentas, sendo crescente o uso de ferramentas da área de negócio para representar processos nas áreas de ciência e engenharia [YGN09].

Inversamente, muitos processos da área de negócio requerem também capacidades computacionais de larga escala e/ou produzem um grande volume de dados. Por exemplo, as simulações na área do mercado de acções são bastante complexas, e têm de produzir resultados em tempo real para um elevado número de utilizadores. As áreas emergentes relacionadas com a *web* e redes sociais têm também de processar um elevado número de dados.

Assim, verifica-se uma necessidade, em ambas as áreas, de estender as ferramentas de cada domínio, ou desenvolver novas soluções [SKD10] , de modo a dar resposta aos novos requisitos tanto da área de negócio, como das áreas de ciência e engenharia. Um destes requisitos é a necessidade de desenvolver sistemas de *workflow* com mecanismos de adaptação dinâmica mais complexos.

1.1.2 Adaptação Dinâmica de *Workflows* de Serviços

Tanto para os serviços em geral, como no caso particular de sistemas de *workflow* de serviços web, a existência de mecanismos de adaptação dinâmica é uma característica fundamental que tem de ser garantida pelos sistemas de suporte à inter-ligação/integração serviços e sua coordenação. Por exemplo, o uso e importância crescentes dos *web services* implica que da parte dos *"providers"* (i.e. de quem disponibiliza o serviço) exista a garantia de acesso constante, independentemente da localização geográfica, bem como tolerância a falhas. Os problemas associados a essa garantia aumentam se o serviço depender de outros e/ou se existir composição de serviços. Uma das dimensões na solução deste tipo de questões é suportar a reconfiguração dinâmica do sistema, por exemplo, substituir dinamicamente (de modo transparente/automático) um serviço indisponível por outro, ou recorrer a uma sua réplica, reduzindo o impacto de possíveis falhas ou de má qualidade de serviço. Por outro lado, é necessário responder a novos requisitos dos utilizadores em tempo de execução, e.g. com a inclusão ou substituição dinâmica de serviços, ou alteração das dependências entre serviços.

Adicionalmente, a interacção com serviços que representam recursos com estado ou actividades de longa duração, requer não só modelos de interacção mais complexos, e.g. baseados em eventos e fluxos de dados como sejam publicador/subscritor e *streaming*, como também mecanismos de suporte à sua adaptação dinâmica. Por exemplo, pode ser necessário reduzir a cadência da entrega de dados num *stream*, caso o receptor não tenha capacidade de realizar o processamento desses dados à mesma velocidade. Do mesmo modo, para que as aplicações clientes se possam adaptar a alterações no contexto desses serviços ou da interacção com eles, pode ser benéfico mudar o modelo de interacção

usado, tal como descrito em [BGP12]. Por exemplo, caso se verifique a detecção de valores elevados de temperatura em WSNs numa determinada zona, pode ser necessário mudar o modelo de interacção com o serviço web que representa essas WSNs, e.g. de publicador/subscritor, para *streaming*.

Para além disso, caso o acesso e coordenação deste tipo de serviços esteja a ser orquestrada a partir de uma ferramenta de *workflow*, seria benéfico que alterações no contexto desses serviços pudessem desencadear a reconfiguração dinâmica de um processo em execução. Assim, a reconfiguração dinâmica de *workflows* de serviços poderia ser desencadeada não apenas do lado da aplicação, como também como resultado dos serviços com os quais as aplicações interagem.

Como a seguir se descreve, o trabalho proposto nesta tese tem precisamente como objectivo disponibilizar mecanismos de reconfiguração dinâmica que permitam oferecer esses tipos de adaptabilidade, bem como contribuir para o suporte à definição estruturada de *workflows* tanto da área de negócio como da área de engenharia.

1.2 Objectivo do Trabalho e Solução Proposta

O trabalho apresentado nesta tese teve como objectivo disponibilizar mecanismos estruturados de desenho e reconfiguração dinâmica de *workflows*, cujas as tarefas podem representar ou aceder a *web services*, sejam serviços sem estado, ou serviços representando recursos com estado ou actividades continuadas no tempo. Os mecanismos estruturados de reconfiguração dinâmica baseiam-se no conceito de padrão, sendo estes disponibilizados nas dimensões de estrutura e comportamento. Os padrões de coordenação implementados são orientados aos dados, e.g. produtor/consumidor, *streaming*, e publicador/subscritor.

Para mais, este trabalho teve como objectivo possibilitar o desenvolvimento de aplicações quer da área de negócio, quer da área científica, que possam tirar partido dos mecanismos de reconfiguração dinâmica oferecidos. Para tal foi escolhida uma ferramenta de suporte à especificação e execução de *workflows* da área de negócio, que permite a especificação de processos em BPMN e oferece um motor de execução para BPMN. Esta ferramenta foi estendida com as seguintes funcionalidades:

- As tarefas podem ser agregadas com base em padrões característicos da área científica, i.e. orientados ao fluxo de dados, para além do *BPMN* permitir especificar padrões característicos da área de negócio, i.e. orientados ao controlo de fluxo. O interesse na extensão de ferramentas de *workflow* da área de negócio para uso na área científica, ou no desenvolvimento de abordagens integradoras, é uma área de investigação que tem merecido um interesse crescente, tal como descrito em [SKD10].

- Os *workflows* são reconfiguráveis, quer em tempo estático, quer em tempo de execução, permitindo assim acomodar novos requisitos definidos pelo utilizador, bem

como modificações relacionadas com os *web services* acedidos.

Embora não tendo sido possível realizar no tempo útil desta tese, este trabalho tinha também como objectivo a integração com o trabalho desenvolvido no contexto de outra tese de mestrado [Bap10], de modo a permitir o acesso a serviços "com estado" usando modelos de interacção dinâmicos, oferecidos no contexto de uma sessão. No entanto, a descrição conceptual dessa integração é descrita na secção 5.3.

Segue-se a descrição mais detalhada da solução proposta por este trabalho e de dois exemplos de aplicação.

1.2.1 Solução proposta

A solução proposta nesta tese baseia-se no conceito de padrão, visto que os padrões são aplicados como formas estruturadas de interacção entre tarefas num *workflow*, existindo padrões de estrutura e comportamento, que servem de base aos mecanismos de reconfiguração dinâmica suportados. A justificação para utilização de padrões prende-se com a observação de que, nos sistemas modernos, a forma como os utilizadores de um serviço comunicam com ele, e também a interacção entre serviços, segue um conjunto de padrões recorrentes, tais como, cliente-servidor, publicador/subscritor, *streaming*, produtor/consumidor, *Paremeter/Sweep*, etc. Adicionalmente, os mecanismos de reconfiguração oferecidos são estruturados, no sentido que se baseiam no conceito de padrão, com separação das dimensões de estrutura e comportamento. Cada padrão pode ser alterado individualmente, ou ser substituído por outro. Tal como para o desenvolvimento de sistemas, a utilização de padrões no suporte à reconfiguração dinâmica é importante como forma de capturar e reutilizar situações recorrentes, e, recentemente, estratégias comuns de reconfiguração dinâmica. Alguns padrões apresentam inclusivamente uma estrutura reconfigurável, permitindo o seu uso como base da definição de arquitecturas adaptáveis. Por exemplo, padrões arquitecturais [BMR+96] tais como Publicador/Subscritor, Cliente/Servidor, Master/Slave, etc., permitem flexibilidade na alteração do número de alguns dos seus componentes (e.g. número de subscritores, clientes, escravos), disponibilizando assim formas de suporte à escabilidade nos sistemas que neles se baseiam. Sendo objectivo deste trabalho a disponibilização de um protótipo com formas de suporte à reconfiguração dinâmica de *workflows* de serviços com base em padrões, no contexto desta tese foram avaliados os seguintes tipos de padrões no âmbito de composição e interacção entre serviços embora apenas um subconjunto tenha sido implementado:

- padrões arquitecturais, tais como, cliente-servidor, publicador-subscritor, *streaming*, etc;
- padrões de desenho, tais como topologias comuns (estrela, *pipeline*, etc), *façade*, *proxy*, etc;
- padrões de Workflow, tais como, alguns padrões de controlo de fluxo e de controlo de dados, etc;

- padrões de interacção/conversação no contexto dos serviços, tais como, *Atomic multicast notification*, *Multi-responses*, etc.

A ferramenta escolhida para a implementação deste trabalho designa-se *Activiti* [Act] e disponibiliza a *Business Process Modeling Notation (BPMN)* para a especificação de *workflows*, bem como o suporte à execução directa desses *workflows BPMN*. Usando a notação *BPMN* é possível definir diversos padrões de *workflow* como especificado em [WvdAD+05]. A ferramenta *Activiti* foi estendida com a inclusão de alguns padrões típicos de fluxo de dados: *Streaming*, Produtor/Consumidor e Publicador/Subscritor. Para mais, foram disponibilizados *templates* de padrões estruturais representando topologias típicas: *Pipeline*, estrela e anel. Estes *templates* permitem agregar tarefas genéricas definindo *workflows* abstractos que podem ser reutilizados em diferentes aplicações. Basta para isso que eles sejam parametrizados com tarefas específicas ou outros padrões. A secção 3.1 descreve em detalhe o modelo subjacente a solução.

De seguida serão apresentados dois exemplos de aplicação para ajudar a clarificar o objectivo deste trabalho, e a solução implementada.

1.2.2 Exemplo da aplicação na área empresarial

Um exemplo típico de *workflows* na área de negócio é a especificação de processos em gestão de projectos. As tarefas associadas à gestão de novos projectos vão desde o registo de informação do projecto; contacto com colaboradores responsáveis pela elaboração da proposta a apresentar ao cliente; possível correcção dessa proposta; implementação da proposta caso seja aceite; etc.

Em sistemas de *workflow* característicos da área de negócio é possível especificar o fluxo de execução com base em regras de controlo: escolher um de dois caminhos no *workflow*; continuar a execução por os caminhos possíveis; esperar que um conjunto de tarefas em execução termine para prosseguir para uma nova tarefa; etc. No entanto, não é do nosso conhecimento que existam sistemas de *workflow* que combinem esses tipos de controlo de fluxo de execução, com padrões de coordenação baseados em fluxo de dados, típicos da área científica (e.g. o produtor/consumidor, *streaming*, etc), que possam ser reconfigurados em tempo de execução. O exemplo de aplicação descrito na secção 5.1 pretende assim ilustrar a vantagem de combinar estes dois tipos de padrões baseados em fluxo de controlo e, em fluxo de dados com reconfiguração dinâmica.

1.2.3 Exemplo de aplicação no domínio *DDDAS*

A aplicabilidade do trabalho proposto nesta tese ao contexto de *workflows* de *web services* em ciência e engenharia, é ilustrada nas secções 5.2 e 5.3.3. O exemplo escolhido pertence ao domínio dos sistemas *Domain Data-Driven Application System (DDDAS))* [Dar04], e representa um cenário de emergência como resultado da ocorrência de inundações.

Tal como descrito em [Dar04], o paradigma *DDDAS* permite representar a integração dos aspectos computacionais e de recolha de dados, necessários a aplicações de simulação de eventos complexos. As sinergias entre aplicações de simulação e as infraestruturas de medição/monitorização, são representados num "ciclo fechado de controlo" (*feedback/control-loop*), onde os dados recolhidos permitem melhorar a precisão e desempenho das simulações, e onde estas, inversamente, podem parametrizar esses mecanismos de recolha/acesso aos dados. O exemplo apresentado na secção 5.2 ilustra como o trabalho proposto nesta tese, poderia ser usado para implementar uma simulação típica deste domínio, tal que a aplicação poderia escolher e parametrizar as fontes de dados existentes e, inversamente, como essas fontes poderiam implicar alterações nessa aplicação de simulação. Em geral, nos exemplos apresentados nas secções 5.2 e 5.3.3, os *workflows* construídos representam diversos cenários no contexto da aplicação de simulação, no domínio da análise e monitorização de inundações, sendo realizadas acções de reconfiguração dinâmica de acordo com o evoluir da situação. Estes exemplos ilustram assim a vantagem de se poder especificar uma aplicação através de um *workflow* que permite o acesso a *web services* e, de combinar modelos de controlo de execução nas dimensões de fluxo de controlo e fluxo de dados, e com mecanismos de reconfiguração dinâmica também nos modelos de interacção dinâmica com os serviços.

1.3 Contribuições da Tese

As contribuições desta tese incluem:

- A extensão de uma ferramenta de modelação de *workflows* da área de negócio (*BPMN 2.0*) com padrões de comportamento característicos da área de ciência e engenharia (i.e. padrões orientados ao fluxo de dados).
- O uso de padrões como um meio para representar interacções entre tarefas num *workflow*, podendo estas tarefas representar o acesso a *web services*, bem como servindo de base aos mecanismos de reconfiguração dinâmica implementados.
- A definição de uma arquitectura e implementação de um protótipo *middleware* que incorpora:
  - A implementação de padrões identificados em 1, nas dimensões de estrutura e comportamento.
  - O suporte à construção de *Workflows* de *Web Services* permitindo uma estruturação e orquestração baseada em padrões.
  - Mecanismos de reconfiguração dinâmica com base em padrões.

1.4 Estrutura do Documento

O documento está estruturado na seguinte forma:

- O capítulo 2, faz um levantantamento do estado da arte, nas áreas de análise e linguagens de composição de *workflows*, tecnologias orientadas a serviços (*SOA*, *web services*), reconfiguração dinâmica e por fim a padrões a vários níveis: *workflows*, desenho, interacção e comunicação entre serviços.
- O capítulo 3, demonstra o modelo conceptual por detrás do desenvolvimento desta tese, e um levantamento tecnológico efectuado para encontrar a melhor ferramenta de forma a entendê-la para os objectivos pretendidos.
- O capítulo 4, descreve a implementação efectuada na ferramenta escolhida.
- O capítulo 5, ilustra exemplos de avaliação do nosso protótipo.
- O capítulo 6, descreve as conclusões obtidas.

2

Estado da arte

Neste capítulo são apresentados os temas relacionados com o domínio desta tese, tendo em vista a construção de um sistema de composição de serviços suportando alguns mecanismos de reconfiguração dinâmica. A figura 2.1 descreve, em particular, como esses temas estão relacionados, bem como quais os métodos/tecnologias escolhidos.

A secção 2.1 começa por descrever a noção de *workflow*, a forma mais comum de composição e interacção de serviços, bem como sistemas de suporte à sua gestão. A secção 2.2 descreve os conceitos associados à noção de serviço, em particular. O modelo SOA é uma referência de base na definição das directivas necessárias ao desenvolvimento de métodos de partilha de serviços, e por isso é descrito com algum detalhe nessa secção. Os *web services*, em particular, são um instanciação do modelo SOA, possibilitando interoperabilidade entre sistemas.A secção 2.3, por sua vez, descreve o que são *workflows* de *web services*, bem como quais as linguagens que podem ser usadas na sua composição/-coordenação.

Outras formas de coordenação, para além de orquestração e coreografia, são através de padrões de interacção/comunicação, descritos nas secções 2.4.4 e 2.4.3. Padrões que ajudam a suportar interacções estruturadas entre serviços.

A secção 2.4 descreve exemplos particulares de padrões que capturam esquemas de interacção recorrentes, quer no contexto da modelação do *workflow*, quer do desenho de aplicações distribuídas em geral, e baseadas em serviços, em particular. Este trabalho, procura tirar partido da reutilização destes modelos típicos de interacção, também como forma de suporte a reconfiguração dinâmica.

Os padrões para além de suportar interacções, poderão ser usados juntamente com

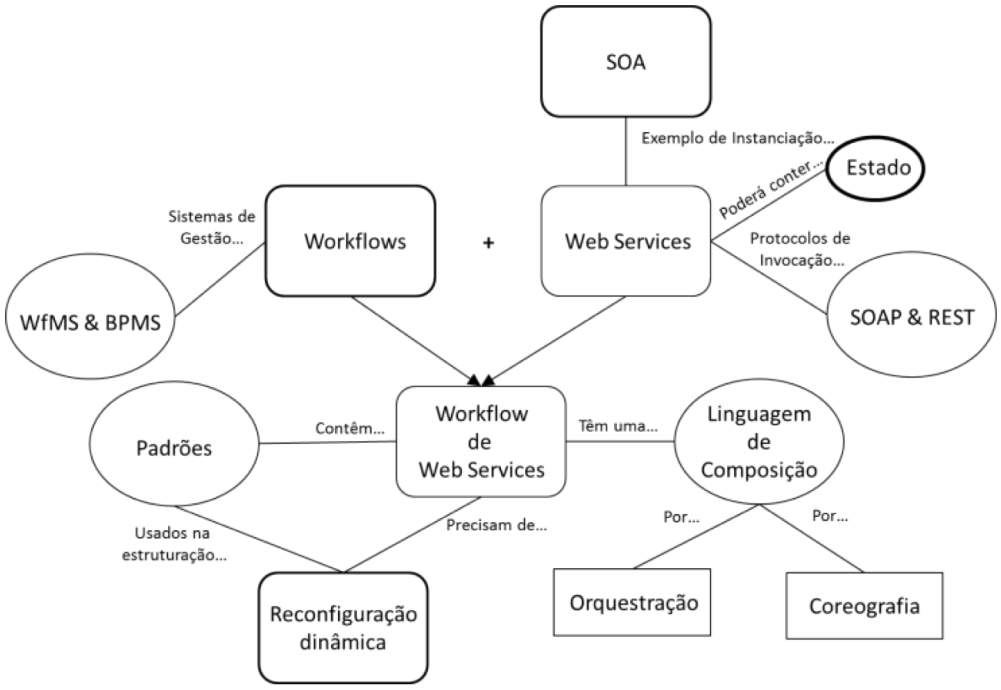


Figura 2.1: Áreas do estado da arte

mecanismos de reconfiguração dinâmica, apresentados na secção 2.5, para garantir adaptabilidade as mudanças externas. A secção 2.6 exemplifica essas reconfiguração no domínio dos *workflows*.

2.1 *Workflows*

Nos sistemas modernos e com a evolução da tecnologia, a cooperação tem-se mostrado essencial para o desenvolvimento tecnológico. Neste seguimento, surge a noção de coordenação, que é necessária para o funcionamento de um sistema como para as comunicações e interacções entre sistemas.

O artigo [MC90] define coordenação como o "acto de trabalhar em conjunto em harmonia", isto é, o trabalho a produzir por vários intervenientes, para a realização de múltiplas actividades dependentes entre si que têm um fim em comum. A noção de dependência entre actividades (i.e. actividades inter-dependentes) significa que as actividades estão dependentes do desfecho de outras para se poderem realizar.

Os *workflows* são um dos possíveis exemplos da especificação de coordenação entre actividades. Sendo visível que, para certos trabalhos, existem tarefas com dependências claras e a sua execução contém passos possíveis de automatizar, a ideia de criar um fluxo de trabalho coordenado foi a ideal. Para se verificar sucesso, as várias tarefas a realizar terão que ser executadas e coordenadas entre si, para garantir que o objectivo final do processo (i.e. qual o procedimento que leva à criação de um produto ou serviço numa área de negócio) é atingido e, com a eficácia necessária. Os *sistemas de workflow* tentam

simplificar essa execução coordenada.

No processo de definição de um sistema de *workflows* é necessário distinguir que componentes existem[GHS95]:

1. ***Workflow*** captura um processo, definindo qual a sequência de tarefas a seguir e as condições/regras a realizar, de modo a realizar o objectivo final;

2. **Tarefas** representam as operações necessárias, podendo ser realizadas por humanos e/ou máquinas [HTZD08];

3. **Objectos manipuláveis (ou transmissíveis) entre tarefas** documentos, imagens, etc;

4. ***Roles*** Quais as *skills* humanas necessárias ou um serviço de um sistema de informação necessário, para realizar uma tarefa em particular;

5. **Agentes** são os humanos ou sistemas de informação que cumprem papéis (*roles*), realizam tarefas, e interagem durante a execução do *workflow*.

Os sistemas de *workflow* podem ser vistos como um mecanismo específico para garantir a coordenação entre um conjunto de tarefas que constituem um processo, podendo ainda estar embebidos numa aplicação ou então serem autónomos. Tal implica diferentes formas de os implementar tal como referido em [Ple02].

Outra dimensão importante é a linguagem de especificação já que define a estruturação do *workflow*, a duração e condições das tarefas e permite definir como são manipuladas as excepções geradas. Existem dois tipos de especificações, nomeadamente modelo abstracto e modelo concreto (ou de execução) [YB05]. No modelo abstracto, um *workflow* é descrito numa forma abstracta, isto é, o *workflow* é especificado sem referir que recursos vão ser usados na execução das tarefas. Este modelo, fornece uma forma flexível aos utilizadores de definirem *workflows* sem se preocuparem com detalhes de implementação de baixo-nível. Usando mecanismo de descoberta e mapeamento, as tarefas são transportáveis e podem ser mapeadas a um serviço em tempo de execução. Para além disso este modelo facilita a reutilização e a partilha de *workflows*. Ao invés, no modelo concreto os recursos são especificados e associados às tarefas no processo de modelação, o que não garante as qualidades mencionadas anteriormente. Estando os sistemas distruíbuidos num meio dinâmico, é mais adequado os utilizadores definirem os *workflows* num modelo abstracto. Dependendo então dos estados dos recursos, o modelo concreto pode ser gerado, totalmente ou parcialmente, antes ou durante a execução de um *workflow*.

A dificuldade em implementar e, garantir interoperabilidade, dos processos de negócio num sistema de *workflows* levou a criação de diferentes sistemas de gestão, referidas a seguir.

2.1.1 *WfMS e BPMS*

Sendo essencial para as empresas a construção de *workflows* adequados, surgiram os sistemas de gestão de *workflow* (WfMS) que facilitam a especificação, execução, registo, e controlo dinâmico de tarefas, envolvendo múltiplas pessoas e/ou sistemas externos (*heterogeneous, autonomous and / or distributed , HAD*).

O acesso a sistemas externos (*HAD*) é particularmente importante para a evolução dos sistemas de gestão de workflows, devido a constante necessidade de partilha de informação entre intervenientes, pelo que é necessário garantir que os *workflows* se consigam integrar e interoperabilizar. Os sistemas de gestão de processos de negócio surgiram para facilitar essa integração (*Business Process Management System (BPMS)*).

Nos *BPMS* são utilizados métodos, técnicas e ferramentas para analisar, modelar, publicar, optimizar e controlar processos envolvendo recursos humanos, aplicações, documentos e outras fontes de informação. Os sistemas *WfMS* são focalizados em gerir *workflows* e os *BPMS* não só gerem *workflows* como os integram com outros sistemas e controlam o processo envolvido [KLL09].

Para atingir integração e interoperabilidade, os sistemas *BPMS* contém, no seu ciclo de vida, quatro *standards* básicos: *standards* gráficos, de execução, de intercâmbio e de diagnóstico. Todos eles, têm a particularidade de usar a linguagem *XML* como *standard* de comunicação [KLL09]: a) *Standards* gráficos, são direccionados para permitir aos utilizadores manipular e visualizar o fluxo de informação, os pontos de decisão e os papéis dos processos de negócio numa maneira visualmente estruturada (por exemplo, com diagramas); b) *Standards* de intercâmbio, têm o papel de traduzir os *standards* gráficos em *standards* de execução; c) *Standards* de diagnóstico, ajudam a entender que tipo de tecnologias servirão para fins de *reporting* e monitorização do sistema; d) *Standards* de execução, focados nos procedimentos necessários para a realização do fluxo de trabalho.

No âmbito desta tese os *standards* de execução e de modelação são os mais importantes, visto usarem tecnologias que garantem interoperabilidade e integração entre sistemas. Um bom exemplo será a junção entre *Web Services Flow Language (WSFL) + XLANG* [KLL09], que usando a capacidade da *WSFL* de ofercer serviços para entidades externas e a forma verbal de representar os dados na *XLANG*, originou a linguagem *Web Services - Business Process Execution Language (WS-BPEL)*, anteriormente denominada *Business Process Execution Language for Web Services (BPEL4WS)*. Esta linguagem fornece a possibilidade de comunicação com processos externos, o que traz as vantagens necessárias para implementação e integração de *web services* específicos com o sistema. Na secção 2.3.1 esta linguagem será apresentada em maior pormenor.

Os sistemas de *workflow* garantem uma forma de compor tarefas, podendo essas ser serviços. No entanto, conceito o "partilha de serviços" tornou-se fulcral para construção de arquitecturas de sistemas distribuídos, sendo fundamental no contexto de uma arquitectura orientada a serviços, que a seguir se descreve.

2.2 *SOA*

O termo serviço é usado para variadíssimos contextos e, o dicionário de língua Portuguesa [Pri] diz-nos que é o acto de servir ou de fornecer algo, num domínio específico. Em sistemas informáticos, o termo serviço pode significar "funcionalidades de um qualquer *software*" ou até mesmo "o uso de um qualquer requisito *hardware*". Um bom exemplo será, num sistema de gestão de *workflows*, a disponibilização para entidades externas de um serviço de funcionalidades base de monitorização, seja para acesso a dados ou para fins de integração.

Um serviço tem essencialmente de estar bem-definido e ser independente do contexto ou estado de outros serviços. Ao ser bem-definido, significa que os utilizadores sabem qual o seu objectivo e como com ele vai interagir. Para isso, terá de estar bem identificado e terá de garantir que o pedido seja atendido. Ao serem independentes e criados num domínio particular, os serviços têm a vantagem de poderem ser utilizados para partilha de informação e, ainda, oferecer funcionalidades específicas do sistema. Ambas as particularidades facilitam a integração entre sistemas. No entanto, o desafio para partilha de serviços está em garantir vários pontos importantes, tais como, escalabilidade (e.g. acrescentar novos serviços), disponibilidade (e.g. mecanismos que lidam com excepções que possam ocorrer) , desempenho (e.g. rapidez de execução), robustez (e.g. tolerância a falhas) e qualidade de serviço (e.g. o conjunto de todas mencionadas mas o respeitar o acordado com o cliente).

Com esta premissa, surgiu a vertente *SOC (Service Oriented Computing)* , em que as aplicações se baseiam num conjunto de serviços disponibilizando funcionalidades bem definidas. De forma a estruturar um conjunto de interacções entre serviços num sistema, SOC instância as arquitecturas orientadas a serviços *(Service-oriented architecture (SOA))* que fundamentalmente são as directivas (*guidelines*) a seguir na construção de um sistema informático com partilha de serviços, utilizadas durante as fases de desenvolvimento e integração de sistemas. As arquitecturas *SOA* têm então como objectivo encontrar metodologias que garantam o desenvolvimento dos processos de negócio, adicionando valor aos serviços e possibilitando uma maior eficiência nos processos. Para isso, a garantia de partilha de serviços é fundamental entre múltiplos sistemas de diferentes domínios, e o termo serviço é encarado como um meio para responder a um pedido/objectivo [Arc].

No processo de implementação de um sistema, uma das características principais deste modelo de arquitectura é a separação da implementação dos serviços da sua definição, bem como serem independentes entre si, excepto possivelmente em funcionalidades bem identificadas. Tal garante melhor e mais fácil interacção entre consumidores e serviços. Com serviços independentes, a sua disponibilidade têm de ser garantida com mecanismos de auto-correcção, isto é, no seu desenvolvimento terão que ser criadas ferramentas para ajudar à sua manutenção, para serem posteriormente fornecidas aos gestores do sistema.

No início da implementação de um sistema, e usando as directivas (*guidelines*) de uma

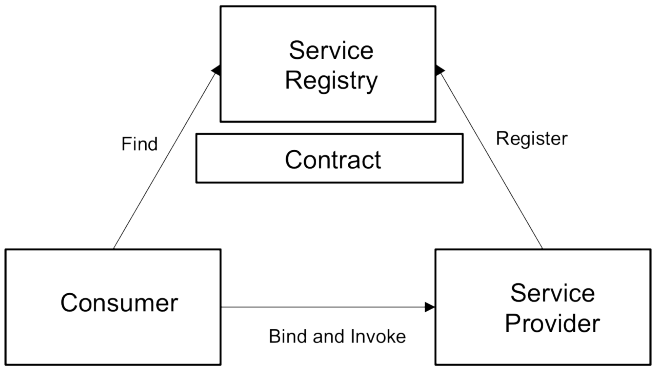


Figura 2.2: Paradigma *"Find-Bind-Execute"*

arquitectura *SOA*, é necessário primeiro distinguir quais os intervenientes (entidades) principais [SOA]:

Service Contract define a forma específica de comunicação e ligação entre o consumidor e o registo, utilizando tecnologias conhecidas por ambos;

Service Registry regista os serviços existentes e os consumidores interessados num dado serviço, mantêm os contratos entre o consumidor e o fornecedor de serviços

Service Provider fornece o serviço. Regista os serviços que implementa no *registry*.

Consumer requer/procura um serviço. Começa por procurar um serviço no *registry* e quando o encontra, cria um contrato entre ambos que têm de ser respeitado na execução do pedido.

Service Proxy por questões de performance é o responsável em manter em *cache* referências para os serviços mais usados.

Service Lease talvez das entidades menos utilizadas em instâncias do modelo *SOA*: dá a conhecer aos serviços a informação dos consumidores descrita nos contratos; para isso, é dado um período de tempo ao consumidor em que o contrato é válido, passado esse tempo terá de realizar de novo o pedido.

Como demonstrado na figura 2.2, é necessário que os três conceitos básicos de partilha de serviços coexistam: *"Find, Binding e execution"*, isto é, procura, ligação e invocação de um serviço. Este paradigma é a base para a existência de partilha e acordo entre as partes envolvidas, e permite que os serviços sejam dinamicamente descobertos e ligados, com uma interface de ligação em rede, de independente localização, o que garante interoperabilidade e fácil distribuição de partilha. Essencialmente é criada uma camada de *middleware* comum entre diferentes sistemas.

2.2.1 *Web Services*

O modelo arquitectural *SOA* pode ser concretizado sobre várias tecnologias, sendo a mais generalizada designada por *Web Services* [Ser].

Web services referem-se a tecnologias, ou metodologias, centradas em sistemas informáticos, que permitem criar ligações a serviços disponibilizados na *web*, sendo serviço o endpoint da ligação. Tipicamente, os serviços estão hospedados num servidor e usando o protocolo *HTTP*, os *web services* fornecem uma *interface* já *standard* para que os sistemas remotos os acedam e os executem. A sua interoperabilidade e o seu *deployment* na internet aumentou consideravelmente a popularidade dos *web services*.

Actualmente existem algumas normas bem definidas para o seu uso, tais como: de segurança (WS-Security), de mensagens (WS-ReliableMessaging), de transacção (WS-Transaction), de gestão (WS-Management) e de composição (WS-BPEL e WS-CDL) [W3S]. Normas que posteriormente serão referenciadas na secção 2.3.

A construção e desenvolvimento deste tipo de arquitecturas realizam-se através de dois possíveis protocolos de invocação, *SOAP* e *REST*, explicados respectivamente nas secções 2.2.1.2 e 2.2.1.3.

Contudo os tradicionais *Stateless web services* não têm sido capazes de responder as novas exigências tecnológicas, o que originou o conceito de noção de estado (*Stateful web services*). Na secção seguinte será apresentado em maior detalhe o porquê da sua importância.

2.2.1.1 *Stateful Web Services*

Com a evolução da tecnologia de informação, novas exigências têm surgido, por exemplo, muitas das interacções que existem são mais sofisticadas que a tradicional comunicação pedido-resposta dos *Stateless web services*. Sendo que, em alguns casos, a invocação de serviços cria ligações de longo termo necessárias para o funcionamento, a disponibilidade tem de ser mantida de alguma forma. Por exemplo, num sistema interactivo que necessite de uma ligação a um sistema para a realização de uma tarefa e num dado instante seja necessário saber em que estado está o processo de atendimento a esse pedido, esse estado é uma característica do *web service*. Assim, surge a noção de estado que basicamente representa cada pedaço de informação que necessita de se ter em consideração na formulação da resposta, mas que não está relacionada com o conteúdo do pedido [OAS].

A noção de estado incide em dois pontos: estado da Infra-estrutura que suporta o serviço e estado da aplicação que representa o serviço. No exemplo acima, a informação da aplicação contem alguns aspectos interessantes para os clientes, incluindo: como os recursos são identificados e referenciados por outros componentes do sistema; como as mensagens podem ser enviadas de maneira a que execute e retorne um dado recurso; a forma como são criados esses recursos; Como são terminados e como podem ser alterados [Ibm04].

A seguir serão apresentados os métodos de invocação tradicionais aos *web services*.

2.2.1.2 *SOAP*

Nos *web services SOAP*, a troca de mensagens é realizada exclusivamente usando o protocolo *Simple Object Access Protocol (SOAP)* que, no sentido figurado, representa um envelope que contem a informação a ser enviada e que posteriormente será processada. Basicamente o protocolo SOAP é uma norma de comunicação baseada em *eXtensible Markup Language (XML)* e que utiliza o protocolo HTTP para o envio de mensagens.

Nos *web services SOAP*, a linguagem *XML* serve como base de definição da linguagem *Web Services Desciption Language (WSDL)*, usada na definição do ponto de ligação nos *web services SOAP*, isto é,definição e parametrização sobre, onde e como se pode aceder aos serviços.

Para fins de registo e gestão de serviços existe uma plataforma que suporta este papel, denominada *Universal Description, Discovery and Integration (UDDI)*.

Para os *Web Services SOAP* existem três tipos de intervenientes, o *consumer*, o *publisher service* e o *service directory*. De seguida, com apoio na figura 2.3, será explicado o funcionamento, numa forma muito sucinta, do processo de troca de mensagens entre esses intervenientes (é usado o protocolo *SOAP* para especificar a troca de mensagens sendo utilizada a linguagem *WSDL* para descrever as operações existentes nos serviços disponíveis): a) O *service directory/registry* é onde estão registados todos os serviços acessíveis e onde ocorre a comunicação com o UDDI; b) O *publisher service*, tem a responsabilidade de descrever e responder onde e como se comunica com um serviço, usa a linguagem *WSDL* para descrever as operações e publica os serviços no *service directory*; c) O *consumer* é aquele que procura o serviço.

1. O primeiro passo será o *consumer service* enviar para o *service directory*, utilizando a *API* fornecida, um pedido para localizar e saber como se irá conectar a um serviço, isto é, que requisitos são necessários para existir uma ligação;

2. O *consumer service* ao receber e analisar a resposta do *service directory* envia de seguida um pedido ao *publisher service* para obter a resposta desejada;

3. Neste ponto, o *publisher service* encaminha o *consumer service* para o serviço desejado.

Os reais benefícios, neste momento, dos *web services SOAP* centram-se na facilidade de integração entre sistemas dado que o seu uso é generalizado no meio empresarial sendo considerado a forma padrão de interacção.

2.2.1.3 *RESTful*

O paradigma *REST (Representational State Transfer)* foi introduzido e definido em 2000 por Roy Fielding na sua tese de doutoramento [Fie00]. O objectivo deste paradigma é oferecer normas que simplifiquem a interacção entre um cliente e um recurso. As arquitecturas

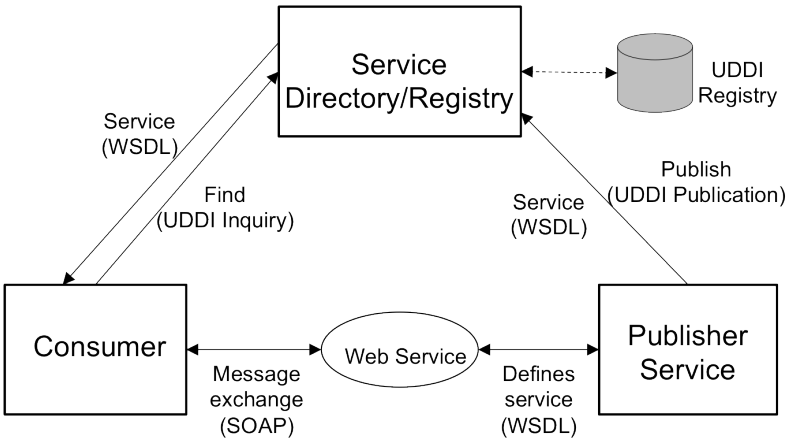


Figura 2.3: Web Services SOAP

baseadas neste paradigma consistem numa interacção semelhante a que existe entre clientes e servidores. Um perfeito exemplo desta arquitectura poderá ser a *Web*, em que os recursos são páginas web e os pedidos são realizados através de pedidos *POST, PUT, DELETE, etc.* Com esta definição surgiu um protocolo de invocação para representar um *web service*, designado *Web Service RESTful*.

Uma arquitectura *RESTful* poderá ser referida como uma arquitectura *ROA (Resource Oriented Arquitecture)* [RR07], isto porque, pegando do que foi mencionado anteriormente, uma arquitectura *RESTful* fornece recursos, sendo esses acedidos por um identificador único através de pedidos por HTTP.

Um recurso é algo que seja suficientemente importante para ser referido e fornecido como um serviço. Habitualmente, neste tipo de arquitectura, um recurso poderá ser algo gravado num computador, como por exemplo um *stream* de *bits*, isto é: um ficheiro, uma linha da base dados, um resultado de um algoritmo, etc. Normalmente, os resultados vêm representados nos formatos *HTML*, *XML* ou *Json*.

Para a construção de *Web Service RESTful*, devem existir quatro princípios básicos [RR07]:

Endereçamento Um *Web Service* é endereçado, isto é, os aspectos interessantes da aplicação devem ser expostos como recursos endereçáveis por um *URI*.

Estado e *Statelessness* Existem dois tipos de estado num serviço *RESTful*. Existe o estado do recurso, que representa a informação dos recursos, e o estado da aplicação, que é a informação do caminho tomado pelo cliente através da aplicação (e.g. na web, o caminho *exemplo.pt\novoexemplo*). O estado do recurso é mantido no servidor e é enviado ao cliente em diferentes formas de representação (*XML*, *HTML*, etc). O estado da aplicação mantêm-se no cliente até ser usado para criar, apagar ou alterar um recurso. Um serviço *RESTful* é *stateless* quando o servidor não guarda o estado da aplicação.

Connectedness O servidor pode guiar o cliente para um estado do sistema enviando-lhe

links nas suas representações.

A *Interface* Uniforme Os princípios a seguir na construção da interface entre cliente e servidor simplificam e isolam a arquitectura, o que permite que ambas as partes evoluam independentemente. Todas as interacções entre clientes e recursos são realizadas através de métodos HTTP básicos. Os princípios são: identificação dos recursos normalmente por URIs; manipulação de recursos através das representações; mensagens auto-descritivas (as mensagens contém informação sobre elas próprias, isto é, os recursos são independentes da sua representação para que o seu conteúdo seja acedido em diferentes formatos); e as interacções com estado são realizadas através de *hyperlinks*, isto é, existe um *Hypermedia* que serve como motor do estado do sistema.

Esta arquitectura maximiza o uso de interfaces já bem-definidas e outros protocolos existentes, por exemplo, *HTTP*, *XML* e *URI*. Contudo, minimiza a adição de novas funcionalidades nas aplicações porque sendo uma padrão de invocação com uma definição estática essas adições são dificultadas.

2.2.1.4 Comparação

As vantagens, partilhadas em ambas as abordagens, focalizam-se principalmente no acesso remoto a serviços e integração de sistemas e ambas oferecem as ferramentas necessárias para se poder criar um sistema integrável e que garante interoperabilidade. No entanto, em comparação com os *web services* tradicionais, a abordagem *REST* tende a ser mais leve (*Lightweight*) devido ao seu estilo mais simplista (pedido directo entre cliente e servidor). Consequentemente esta abordagem é mais fácil de construir e gerir, também porque, tanto do lado do cliente, como do lado do servidor a complexidade envolvida é mais leve neste paradigma (um pedido *HTTP GET* é mais fácil de interpretar que um pedido *SOAP GET*, visto que neste caso existe um *parser* para interpretar a mensagem recebida enquanto que no outro caso a própria mensagem representa o pedido).

Nos WS SOAP, as mensagens estão bem definidas uma vez que são construídas através da linguagem WSDL, permitindo maior flexibilidade para tipificagem de dados e construção de mensagens. Assim, é mais acessível construir uma mensagem com maior informação. Ao invés, na arquitectura *REST*, é mais complicado construir uma mensagem complexa por não se basear numa linguagem expressiva. Em questões de segurança, devido à rigidez do sistema, os *web services SOAP* garantem maior segurança, basta pensar que, na web não é possível enviar dados confidenciais contidos num URI.

2.3 *Workflow* de *Webservices*

O recurso a módulos externos de *software* na forma de serviço tem sido fundamental no processo de desenvolvimento de aplicações. O aumento da complexidade dos sistemas

distribuídos, juntamente com a necessidade de existir coordenação e composição entre serviços externos, tornou a estruturação de diferentes *web services* em *workflow* essencial.

Os *workflows* de *web services* permitem criar um fluxo de trabalho que automaticamente escolhe, compõe e interoperabiliza os *web services* para atingir um objectivo. Os *web services*, e suas características, são usados para partilha de serviços, e os *workflows*, são usados na representação e criação de um fluxo de trabalho de um processo que se rege por um conjunto de tarefas, automatizadas ou manuais, invocando aplicações ou serviços externos. Com características de ambos os mundos é garantida a execução de um fluxo de trabalho de modo eficiente, de acordo com dependências pré-definidas [LF07] e com os melhores executantes, sendo esses o acesso a serviços externos.

Num sistema de *workflow* de *web wervices* é necessário criar uma arquitectura que suporte a fusão de ambos os conceitos. Para atingir esse objectivo existem requisitos mínimos que são necessários para garantir que um sistema é suficientemente robusto. Por exemplo, a garantia de fácil criação, *deployment*, customização e manutenção de serviços e *workflows*. Portanto, é essencial que de início que se defina a linguagem a usar, como se irá realizar a comunicação e a gestão das acções sobre os serviços e intervenientes.

Comunicação A comunicação entre serviços é realizada utilizando um dos protocolos de invocação de *web services*: protocolo *SOAP* (que já se encontra suficientemente normalizado). ou *Web Services RESTful*, que foram mencionados na secção 2.2;

Linguagem Existem dois tipos de linguagens: linguagens de execução e de modelação. Dentro das linguagens de execução, existem duas abordagens com o objectivo de compor e coordenar um *workflow* de *web services*: orquestração e coreografia entre serviços. As diferenças entre elas residem essencialmente na forma de como é executado e controlado o sistema.

Uma linguagem de orquestração específica a execução de um processo entre vários intervenientes com um ponto de controlo, isto é, a troca de mensagens é controlada pelo coordenador (*designer*) central de orquestração. Nenhum dos intervenientes tem noção do estado de outros, apenas o coordenador. A figura 2.4 ilustra esse conceito. Uma das linguagens mais conhecidas nesta abordagem, é a linguagem *BPEL (Business Process Execution Language)* (descrita na secção 2.3.1) que tem como propósito orquestrar o acesso e a execução de um processo de negócio montado sobre um *workflow* de *web services*. Outra linguagem que segue esta abordagem, é a linguagem ORC descrita na secção 2.3.4, que usando quatro primitivas de concorrência disponibilizadas é possível orquestrar acessos a serviços computacionais.

Uma coreografia, por seu lado, caracteriza o protocolo de interacções entre serviços, isto é, define as sequências de mensagens a serem trocadas entre serviços, com o propósito de garantir a interoperabilidade. No entanto é descentralizado porque não é necessário indicar para cada interveniente, que mensagens deve trocar e com quem. A figura 2.5 ilustra o conceito de coreografia entre serviços. Neste caso

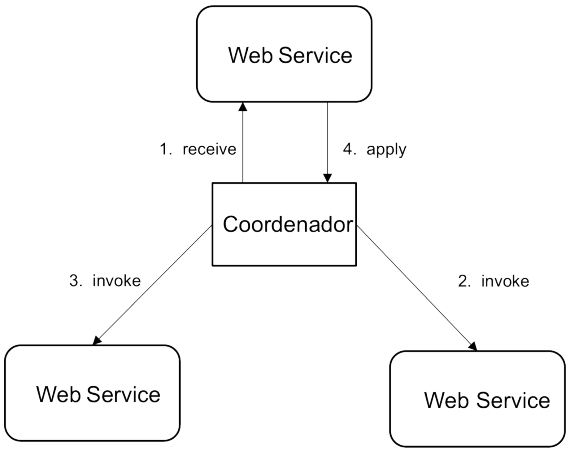


Figura 2.4: Orquestração

a linguagem mais generalizada é *WS-CDL (Web Services Choreography Description Language)* (secção 2.3.2), esta linguagem garante interoperabilidade entre serviços, em que troca de mensagens entre intervenientes é feita numa forma coreografada, isto é, com uma sequência ordenada ou não ordenada ao estilo de uma arquitectura *peer-to-peer*. Cada *peer* conhece as acções que deve realizar no contexto da coreografia

No contexto das linguagens de modelação de *workflows* de *web services*, a notação BPMN é a referência. Consiste essencialmente numa notação gráfica para modelação de processos de negócio, tendo como característica disponibilizar vários tipos de tarefas. Nessas tarefas e de interesse para o nosso trabalho existe a *service task*. Na secção 2.3.3 é descrita em maior detalhe essas características.

Gestão de Acções Para a gestão de acções é frequente existir um *engine*, que poderá ser, em teoria [GL02], acedido como um *Web Service*. Este *engine* tem como intuito controlar e gerir a adição (*deployment*) de novos serviços e de *workflows*, possíveis de serem executados.

As secções seguintes descrevem, respectivamente, as linguagens de execução (orquestração e coreografia) e a linguagem de modelação (*BPMN*) mais conhecidas.

2.3.1 *BPEL*

A linguagem *BPEL* [KL08] descreve um processo como um conjunto de actividades, isto é, um processo é composto por serviços em que esses interagem por troca de mensagens. Tipicamente é usada em *web services*, baseada em XML e é desenhada para partilha de tarefas em ambientes distribuídos.

A BPEL, como mencionado na secção 2.3, é uma linguagem de orquestração.

Descreve o controlo lógico necessário para existir coordenação entre *web services* num

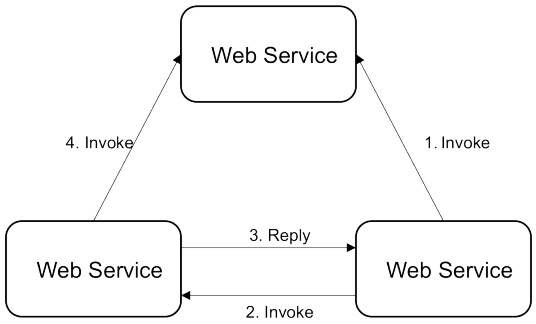


Figura 2.5: Coreografia

fluxo. Contem a definição de um processo, isto é, as actividades, os *links* com as parcerias necessárias, as variáveis e os controladores de eventos. Um dos pontos fortes desta linguagem é, permitir orquestrações complexas entre múltiplos serviços, através de um único serviço de controlo.

As *frameworks WS-coordination* e *WS-transaction* complementam esta linguagem, fornecendo mecanismos que definem os protocolos norma a serem usados nos eventos de transacção, tanto em sistemas base, como em sistemas de *workflow* que necessitam de coordenação entre múltiplos serviços.

2.3.2 *WS-CDL*

WS-CDL [WSC04], que também é codificada em *XML*, descreve as colaborações dos participantes, definindo o comportamento de cada um deles, isto é, a que serviços se devem ligar e de que serviços recebem a informação. Tudo isto para atingir um objectivo comum, um processo de negócio. Não existe coordenador e os serviços têm noção da composição existente, sabem quando devem interagir e que operações é necessário executar.

WS-CDL não define simplesmente a coreografia das mensagens, define também, se relevante, o tipo de mensagens a serem trocadas. Numa coreografia é necessário que os intervenientes estejam conscientes do conteúdo das mensagens pelos que esta contêm *tokens* que ajudam ao reconhecimento da informação.

Adicionalmente, a possibilidade de estabelecer responsabilidade e papéis dentro do fluxo torna-se crucial. Isto para garantir que quando se define a informação a ser trocada, cada um dos participantes saiba o que terá de realizar para atingir o objectivo pretendido. Portanto, sendo essencial definir um participante *(um Web Service)* é importante esclarecer: o seu papel *(roleType)*, isto é, que operações e comportamento terá no sistema; as relações *(relationshipType)*, ou seja, ao estar ligado a outros serviços, que responsabilidades terá de cumprir; e por fim o seu tipo *(participantType)*.

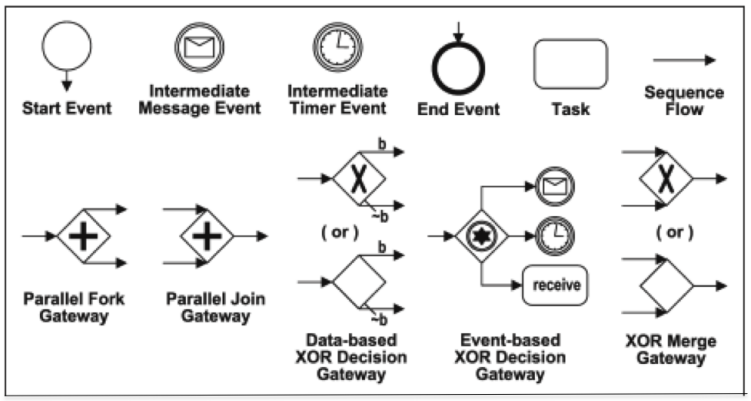


Figura 2.6: Um subconjunto de elementos nucleares *BPMN 2.0* [ODtHvdA07]

2.3.3 BPMN

Business Process Model and Notation (BPMN) oferece uma notação gráfica para modelação de processos de negócio, baseada na linguagem *XML* seguindo o metamodelo *Business Process Definition Metamodel* (pertencente à organização OMG [1]). A notação *BPMN* específica processos de negócio em *Business Process Diagram (BPD)*[WC99] (ou *BPMN-DI, Business Process Model and Notation - Diagram*), baseado na técnica de fluxo gráficos semelhante aos diagramas de actividade da linguagem *Unified Modeling Language (UML)*[Sim05]. Esta notação tem vindo a ter uma crescente aceitação na área de negócio.

Esta notação surgiu com a necessidade de modelar processos de negócio, com uma notação complexa mas ao mesmo tempo intuitiva para utilizadores técnicos e utilizadores nas área de negócio. A especificação da *BPMN* prevê o mapeamento entre os gráficos da notação e a estrutura inerente as linguagens de execução, particularmente a *BPEL*. Normalmente as ferramentas de suporte a esta notação têm um motor de execução para testar os processos construídos.

Com as crescentes evoluções, esta notação encontra-se na versão 2.0 com a particularidade de garantir a construção de modelos coreografados ou orquestrados e de oferecer novos elementos, tais como, eventos assíncronos e/ou temporizados e sub-processos (Figura 2.6).

Os elementos nesta notação estão divididos em cinco categorias [2]:

1. Objectos de fluxo: Eventos (início do processo, fim do processo e eventos temporizados), tarefas de actividade (*activities*) e *gateways*.

2. Objectos de dados: objectos de dados, *input* de dados, *output* de dados e persistência de dados.

3. Objectos de conexão: Linhas de fluxo, linhas de mensagens, associações e associação de dados.

[1] Object Management Group. http://www.omg.org/, específica *standards* na área de *BPM*
[2] http://training-course-material.com/training/BPMN_Presentation

4. Objectos *Swimlanes*: *Pools* e *Lanes*.

5. Objectos artefactos: Grupos e anotações de texto.

Dentro dos objectos de fluxo encontram-se as tarefas de actividade que são fulcrais para a realização de um processo de negócio. Na tabela 2.3.3 encontra-se a descrição das tarefas base de um sistema *BPMN 2.0*.

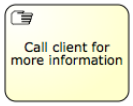
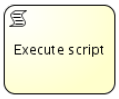
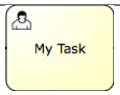
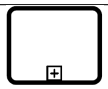
Notação gráfica	Descrição
	Uma tarefa que no fluxo de execução é executada sem qualquer intervenção do motor de execução sendo representativa de uma intervenção humana. (e.g. realizar um telefonema)
	Espera pela chegada de uma mensagem de um participante externo para retomar a execução no ponto em que se encontra no processo.
	Oferece a possibilidade durante a execução de um processo de adicionar novas regras de negócio.
	Executa um conjunto de acções que o motor de execução possa executar.
	É uma tarefa que usa um serviço externo ao processo, por exemplo, como apresentado na figura, executar código *java*.
	O fluxo de execução do processo espera que o utilizador realize uma acção.
	É uma actividade em que os seus detalhes internos são modelados por tarefas, *gateways*, eventos e linhas de fluxo.
	Invoca a execução de um outro processo de negócio.

Tabela 2.1: Elementos básicos da notação *BPMN 2.0*

Existem outras tarefas, por exemplo *email task*, que são específicas de certas ferramentas mas no entanto não são consideradas tarefas básicas de uma ferramenta *BPMN 2.0*. Com estes elementos é possível compor *workflows* de diferentes formas, inclusive construir orquestrações de tarefas.

2.3.4 Orc

A linguagem *Orc* [oTaA] é uma linguagem direccionada à programação distribuída e concorrente. Através da noção de *sites*, oferece um acesso uniforme a serviços computacionais, incluindo comunicação distribuída e manipulação de dados. Esta noção de *site* é a unidade de computação fundamental de um programa *Orc*. Os *sites* são similares a funções ou procedimentos de outras linguagens, representando neste caso o acesso a um serviço que pode ser remoto (criado por outros utilizadores) logo, pode ser não confiável. A linguagem *Orc* foi desenhada para resolver um padrão computacional comum em diversas aplicações de diferentes áreas: obter dados de um ou mais serviços remotos, realizar alguns cálculos com os dados obtidos, e invocar outro serviços com os resultados. Esta obtenção de dados e invocação de outros serviços é designada por *orquestração*. Neste caso, trata-se de uma orquestração de *sites*. Embora nas orquestrações deste domínio a execução deva ser eficiente, existem no entanto condicionantes várias que limitam o seu desempenho, tais como, atrasos na comunicação, serviços não confiáveis, ou excesso de ligações a um serviço diminuindo o seu tempo de resposta. Assim, o objectivo base por detrás desta linguagem é oferecer quatro primitivas simples de forma a facilitar orquestrações: computação paralela, sequência, *selective pruning* e detecção de terminação. Segundo os criadores desta linguagem, a combinação destas quatro primitivas mostra um forma poderosa de expressar padrões de comunicação. Esta linguagem pode ser utilizada como uma linguagem de script *web* de forma a construir *mashups* de *web services*. Para além disso é possível programar padrões de *workflow* usuais tal como apresentado no artigo [CPM06].

2.3.5 *Conclusão*

A linguagem *BPEL* fornece suporte tanto a processos de negócios executáveis como processos de negócio abstractos. Um processo de negócio executável é aquele que modela o comportamento dos participantes numa específica interacção de negócio, enquanto que o processo abstracto (também denominado protocolo de negócio) específica as mensagens trocadas com composições externas. Isto significa a linguagem *BPEL* suporta não só orquestração, com os processos executáveis, como suporta coreografia com os processos abstractos [KL08].

Apesar de ter um controlo central, a linguagem *BPEL*, no entanto, é mais flexível que WS-CDL, porque, sabendo qual é o controlo central e o facto de não existir necessidade de estado, mais facilmente se adicionam novos *web services*.

No entanto, a linguagem *Orc*, sendo também uma linguagem de orquestração, oferece um ambiente poderoso de composição por orquestração de serviços. Contudo não oferece uma forma simples de disponibilizar os mecanismos de reconfiguração desejáveis para esta tese. Ao ser uma linguagem baseada em *scripts* torna difícil durante a execução realizar dinamismo não programado.

A notação *BPMN* oferece suporte a processos de negócios executáveis mas não a abstractos. É uma notação mais geral, isto é, não compõe unicamente *workflows* de *web services*. No entanto, o seu número variado de elementos garante a composições complexas de *workflows*, com a tarefa *service task* a garantir a invocação de *web services*.

2.4 Programação e Computação baseada em Padrões

O conceito de padrão surge como uma forma de capturar conhecimento experiente ou úteis eventos recorrentes, foram sistematizados em primeiro lugar pelo famoso "Gang of Four"[GHJV95] que criaram os padrões de desenho (*design patterns*). A partir desse ponto têm sido extensivamente usados no processo de desenvolvimento de *software*.

Durante algum tempo os padrões eram definidos como descrições textuais/não-formais e eram simples directivas no processo de desenvolvimento de *software*. No entanto, emergiram como uma primeira forma de abstracção, isto é, tornaram-se entidades computacionais com o seu próprio ciclo de vida. Como consequência, tornaram-se manipuláveis, persistentes e podem ser encarados como unidades de execução ou de reconfiguração.

Na fase de modulação de *workflow*, em particular, é frequente verificar-se recorrência na forma como são realizadas as interdependências entre tarefas, como a seguir se descreve.

2.4.1 Padrões de *Workflows*

O conceito de padrões de *workflow*, definido por Van der Aalst et al [dAea], surgiu no seguimento da premissa referida de se observar recorrências na modelação de *workflows*. Tal levou a uma identificação de padrões de *workflow* seguindo uma estratégia que simplifica a manutenção e reduz o trabalho de modulação de um *workflow*. Os padrões propostos foram analisados e divididos por várias dimensões existentes num sistema orientado a processo[VDATHKB03]:

Perspectiva de controlo de fluxo descreve as tarefas e a sua ordem de execução entre diferentes componentes de um *workflow*, isto é, captura os aspectos relacionados com as dependências entre as várias tarefas (e.g. sequência, paralelismo, escolha, sincronismo, etc). Os padrões focam-se nesse aspecto e estão divididos em várias categorias (apresentadas na tabela 2.2 e com exemplos na tabela 2.3).

Perspectiva de recursos Representa os recursos que vão ser reservados (*allocate*) ou delegados a uma tarefa ou a um caminho de execução,. Os recursos podem ser humanos (e.g. um trabalhador) ou não humano (e.g., equipamento) [RHE04]. Os padrões de recursos de um *workflow* têm o objectivo de capturar as várias maneiras de como esses recursos são representados e como utilizam os *workflows*. Algumas das categorias existentes são: Padrões de criação (como o nome indica, padrões que existem na fase de *design* do *workflow*); Padrões de repartição (*allocation*) (retira ou fornece

Categoria	Descrição	Exemplos
Padrões básicos de controlo de fluxo	Categoria que captura os aspectos elementares do processo de controlo e são similares as definições dos conceitos inicialmente propostos pela *Workflow Management Coalition*[Hol95]	Sequência, Divisão paralela, Sincronismo, Escolha exclusiva (tabela 2.2, número: 1,2,3,4) e Junção simples
Padrões de iteração	Padrões capturados pelos comportamentos repetitivos de um *workflow*	Ciclos arbitrários, *Loop* Estruturado (tabela 2.2, número: 7), Recursividade
Padrões de sincronização e ramificação avançada	Padrões que caracterizam conceitos para complexas ramificações e junções que surjam nos processos de negócio.	Junção sincronizada geral (tabela 2.2, número: 5), Multi-junção, Discriminador bloqueante etc.
Padrões de múltiplas instâncias	Os padrões de múltiplas instâncias descrevem situações onde existem múltiplos fio de execução activos num modelo de processo relacionado com a mesma tarefa	Múltiplas instâncias sem sincronização, Junção dinâmica de múltiplas instancias, etc
Padrões baseados no estado	Padrões baseados no estado, reflectem situações em que a solução mais eficaz de se realizar, na linguagem dos processos, é suportado pela noção de estado (estado neste contexto representa os dados inerentes a uma tarefa)	Secção crítica, *Milestone* (tabela 2.2, número: 6), Escolha diferida, etc
Padrões de cancelamento e de forçar a conclusão	Padrões comuns no processo de cancelamento de uma tarefa/*workflow* e no processo de forçar a conclusão de uma tarefa/*workflow*	Cancelar tarefa, Cancelar múltiplas instâncias activas, etc
Padrões de terminação	Representa os padrões existente no momento em que o *workflow* é considerado completo	Terminação implícita e explicita.
Padrões *trigger*	Representam os padrões criados por acções externas que possam ser necessárias para iniciar certas tarefas	*Trigger* transitório e persistente (tabela 2.2, número: 8)

Tabela 2.2: Categorias padrões de controlo de fluxo

recursos); Padrões de desvio (quando falha um recurso existem vários padrões de comportamento possíveis de serem executados); Padrões de auto-arranque (devido a alguma falha certos recursos são automaticamente alocados); etc [3] .

Perspectiva de dados Representa a gestão da informação, escopo das variáveis, etc. Existem uma série de características que ocorrem repetidamente nas diferentes formas de modelação e que podem ser divididas em 4 categorias distintas: Visibilidade de dados, relacionado como os elementos dos dados conseguem ser visíveis nos vários componentes de um *workflow*; Interacção de dados, foca a maneira como os dados comunicam entre os vários elementos activos no *workflow*; Transferência de dados, descreve os mecanismos necessários para a interacção entre elementos numa interface de um *workflow*; Encaminhamentos (*Routing*) baseados em dados, caracteriza a

[3]Workflow Patterns Home page: http://www.workflowpatterns.com/patterns/resource/

maneira de como cada elemento de dados pode influenciar outras operações, particularmente a perspectiva de controlo de fluxo.

Perspectiva de manipulação de excepções lida com as várias causas das excepções e com as várias acções necessárias de correcção.

Os padrões, nestas diferentes perspectivas, ajudam a construir e modular um *workflow* em qualquer domínio, mostrando as fraquezas e forças das várias aproximações de especificação de processos, e podem servir como base para uma linguagem ou ferramenta de desenvolvimento. Um dos exemplos é a linguagem YAWL (Yet Another Workflow Language) [4], que é baseada numa rigorosa análise dos sistemas de gestão de *workflows* e das linguagens *workflow*. A linguagem baseia-se nas *petri nets* [vdAH03] e consegue capturar e identificar alguns padrões de controlo de fluxo, excepto padrões de múltiplas instâncias, padrões de cancelamento e de junções-Ou generalizadas.

Existem vários estudos sobre as diferentes dimensões de padrões existentes, como por exemplo no artigo [MGRtH11] onde é realizada uma análise desses padrões sobre ferramentas como *Kepler*, *Taverna* e *Triana*.

A dimensão padrões de fluxo de controlo foi analisada com a notação *BPMN 2.0*, sendo o responsável da ferramenta *YAWL* o responsável por essa análise [WvdAD+05]. Outro exemplo é o artigo [WC99] que mostra a versatilidade desta notação.

Os padrões de *workflow* são considerados um caso paralelo dos padrões de desenho que são descritos na secção seguinte.

2.4.2 Padrões de desenho

O ciclo de vida do software pode ser dividido em: análise, desenho, implementação, testes e manutenção. Na fase de desenho, é importante focar os problemas possíveis e, numa análise profunda, identificar os padrões de desenho *(design patterns)* que são possíveis de utilizar. Um sistema de *software* surge assim com a análise e a utilização de princípios de *design* no *Software*. Estes princípios representam as directivas (*guidelines*) que ajudam a evitar maus desenhos de um sistema, definindo três características importantes a serem evitadas [Mar03]:

Rigidez Um sistema rígido é difícil de alterar porque essas alterações vão afectar outras partes do sistema.

Fragilidade Um sistema com fragilidade resulta em que, ao ser realizada uma alteração, inexplicavelmente o sistema pára.

Imobilidade Um sistema que apresente características de imobilidade tem dificuldade em ser transposto para outros domínios visto que não se consegue adaptar por estar demasiado dependente da aplicação corrente.

[4] YAWL Home page: http://www.yawlfoundation.org/

Com estas características e com a avaliação de vários sistemas, verificaram-se recorrências em várias pontos, identificando-se 3 categorias diferentes de padrões: Padrões de desenho de criação; Padrões de desenho de comportamento; e Padrões de desenho de estruturação. A primeira lida com os mecanismos de criação, a segunda com a identificação de padrões comuns de comunicação entre entidades e a terceira com a estruturação. Estas categorias simplificam o *design* identificando maneiras eficientes de realizar as relações entre as entidades.

Os padrões têm 4 elementos essenciais na sua representação: Nome (tipicamente descreve o problema); Problema (quando o padrão deve ser aplicado); Solução (fornece uma descrição abstracta dos problemas e a maneira como geralmente os resolve); Consequências (os resultados e os *trade-offs* que possam existir ao aplicar o padrão, normalmente em termos de espaço e tempo de execução).

Os padrões de desenho citados no livro [GHJV95], resolvem, de diferentes formas, muitos dos problemas encontrados no desenvolvimento de um programa orientado a objectos: a) encontrar os objectos apropriados para uma dada tarefa b) determinar a granularidade do objecto (isto é, como vai ser construído) c) especificar as interfaces dos objectos d) ajudar a especificar como se vai implementar o objecto em si

Com o objectivo de ilustrar tipos recorrentes no desenvolvimento de sistemas distribuídos, de seguida, serão apresentados alguns exemplos de padrões agrupados por categoria. Cada um é representado pelo nome, solução, problema e consequências:

- **Padrões de criação**

  Singleton - garante que é criada uma instância da classe e fornece um ponto global de acesso ao objecto. Padrão que deve ser usado quando devemos garantir que só uma instância da classe deve ser criada e quando essa deve ser acedida em todo o código.

  Abstract Factory - oferece a interface para criar uma família de objectos relacionados, sem explicitamente especificar as suas classes. Padrão usado quando só as interfaces do produto (objectivo da classe) devem ser reveladas e a implementação mantém-se escondida aos clientes e os produtos da mesma família devem ser usados todos juntos.

  Builder - define uma instância para criar um objecto mas deixa as subclasses decidirem qual a classe que instanciam. Permite um controlo refinado sobre o processo de construção.

  Prototype - como o nome indica, específica o tipo dos objectos a criar usando uma instância de um protótipo. Criam-se novos objectos copiando os protótipos criados através deste método.

- **Padrões de comportamento**

Chain of Responsibility - permite um objecto enviar um comando sem saber que objecto vai interpretar e gerir esse pedido. Existe uma classe que re-encaminha o pedido para o objecto respectivo.

Command - encapsula os pedidos num objecto para mais tarde serem atendidos: permite chamar os métodos em qualquer altura, não sendo necessário saber que objecto contém o método em questão.

Observer - define a dependência de um-para-muitos entre objectos, de maneira a que, quando um objecto muda de estado, todos os seus dependentes são notificados e alterados automaticamente.

- **Padrões na estruturação**

Proxy - O *proxy* serve como intermediário entre a fonte e as entidades. Fornece transparência na localização, tolerância a falhas e balanceamento de carga através de cache dos objectos mais usados.

Facade - Simplifica um sistema com vários subsistemas com uma simples *interface*. Este padrão também é útil quando um sistema está dividido em vários subsistemas, e tanto o acesso à comunicação como o ponto de entrada do sistema têm de ser restritos.

Para além dos padrões de desenho na construção de um sistema, existem estilos de arquitecturas possíveis de aplicar. Estilo, neste contexto, representa a classificação de uma arquitectura de um sistema de acordo com os padrões similares que existam na sua implementação. Muitas destas arquitecturas são usadas em sistemas distribuídos (e.g. cliente/servidor, *Peer2Peer*). Sendo também, designados por padrões de arquitectura[BMR+96], sendo estes compostos por vários padrões de desenho. Alguns exemplos possíveis de padrões são:

Client/Server - Os clientes fazem pedidos a um servidor o qual tem o papel de enviar a resposta.

Peer2Peer - Cada *peer* é visto com um cliente e servidor ao mesmo tempo. Um *peer* fornece e pede serviços a outros *peers*.

Producer/Consumer - O fluxo de dados é uni-direccional, do produtor ao consumidor.

Streaming - Fluxo continuo de dados ente o servidor e o cliente.

Publish/Subscriber - Um ou mais entidades subscrevem um certo interesse num evento. Quando esse evento ocorre, os subscritores são notificados. Não é necessário estar registado no evento para receber notificação.

Parameter-Sweeper - Repetidas invocações de um componente com um único conjunto de parâmetros.

Master-Slave - O mestre distribui tarefas a serem realizadas pelos escravos e agrega os resultados de cada um.

Layers - ajuda a estruturar aplicações em camadas independentes, de maneira que as mudanças numa camada não afecte as outras, ajudando assim a extensibilidade.

Pipes & Filters - Permite composição de filtros. Arquitectura ideal se existem muitas transformações a realizar e é necessário existir flexibilidade. Em tempo real é possível adicionar filtros.

Adicionalmente, a própria interacção entre serviços apresenta características recorrentes também capturadas através do conceito de padrões.

2.4.3 Padrões de interacção entre serviços

No artigo [BDtH05] são exemplificados alguns padrões existentes no campo dos sistemas *BPM (Business process management)*.

Com o avanço das tecnologias de composição de serviços, a importância das interacções de serviços em processos de negócio colaborativos tornou-se relevante. Os padrões identificados, descrevem cenários onde um dado número de serviços com processos internos, necessita de interagir com outros de acordo com regras pré-estabelecidas. Essas interacções ocorrem, primordialmente, na camada de composição de serviços, que no caso dos *workflows* são exemplificadas nas secções 2.3.2 (coreografia) e 2.3.1 (orquestração). A aplicação de padrões neste contexto é útil para a análise das capacidades da linguagem BPEL e para estender especificações da linguagem WS-CDL. No caso da construção de um sistema de raiz, mostram serem essenciais para a criação de uma linguagem de suporte para interacções entre serviços.

De seguida serão apresentados, sucintamente, os 13 padrões de interacção entre serviços formalizados:

- **Padrões de interacção bilateral com transmissão-única**

  Send - Um serviço envia uma mensagem para outro serviço.

  Receive - Um serviço recebe uma mensagem de outro serviço.

  Send/Receive - Um serviço A envia ao serviço B uma mensagem (o pedido) e recebe a resposta.

- **Padrões de interacção multilateral com transmissão-única**

  Racing incoming messages - Um serviço espera receber uma ou mais mensagens em conjunto. As mensagens podem vir estruturadas de forma diferente e podem vir de diferentes serviços. As mensagens são processadas consoante o tipo e/ou categoria do serviço fonte.

One-to-many send - Um serviço envia mensagens para um conjunto de serviços. As mensagens são do mesmo tipo, isto é, têm o mesmo objectivo mas podem ter formatos diferentes, porque os serviços podem ter diferentes formas de interagir.

One-from-many receive - Um serviço recebe um conjunto de mensagens relacionadas enviadas por diferentes serviços. A chegada das mensagens tem de ser cronometrada de maneira a que sejam atendidas como um pedido único. A interacção pode ser completada ou não, dependendo do conjunto de mensagens recebido.

One-to-many send/receive - Um serviço envia um pedido a outros serviços, que podem ser idênticos ou relacionados logicamente. A resposta é esperada num tempo limite. Se não chegarem dentro desse tempo limite a interacção pode não ser bem sucedida.

- **Padrões de interacção com transmissão-múltipla**

Multi-responses - Um serviço A envia um pedido a um conjunto de serviços de forma a receber as respostas pretendidas. Depois de receber as respostas bloqueia o acesso a outros serviços. Existem várias formas de bloquear: o serviço A envia uma notificação ao conjunto de serviços, para não enviarem respostas; ou é determinado um intervalo de tempo máximo, indicado pelo serviço A; etc

Contigent requests - Um serviço A envia um pedido ao serviço B e, caso não receba resposta dentro de um intervalo de tempo, o pedido é enviado para outro serviço, e assim sucessivamente.

Atomic multicast notification - Envio de uma notificação a um grupo de serviços e que tem de ser aceite dentro de um intervalo de tempo. Só existirá sucesso quando o número de respostas aceite estiver entre um valor mínimo e um máximo definidos.

- **Padrões de encaminhamento**

Request with referral - Um serviço A envia um pedido ao serviço B indicando que a resposta deve ser enviada a um outro grupo de serviços dependendo da avaliação das condições.

Relayed request - Semelhante ao anterior mas o serviço B terá um maior papel, isto é, estará como "observador" da interacção até essa terminar.

Dynamic routing - Baseado numa condição de encaminhamento, um pedido é encaminhado para um conjunto de serviços. A ordem de encaminhamento é flexível, podendo mais de um serviço ser activado para receber o pedido. Depois desse conjunto de pedidos ser atendido, o pedido pode ser encaminhado

para outro conjunto de serviços. O encaminhamento pode ser sujeito a condições dinâmicas, baseadas nos dados contidos no pedido original ou nos passos intermédios.

Os padrões apresentados ajudam a exemplificar possíveis formas de os serviços interagirem, todavia, são apresentados numa forma abstracta e pouco muito detalhada. É neste contexto que surgem os padrões de conversação, que exemplificam recorrências numa forma mais descritiva, isto é, em maior detalhe.

2.4.4 Padrões de conversação entre serviços

Os padrões anteriores ilustram a interacção entre consumidor e fornecedor de serviços, mas são exemplificados sem qualquer noção da camada lógica de comunicação dos serviços. Por exemplo, é exemplificada uma interacção de um para muitos, mas têm em conta as características desses? Isto é, é explicado como o serviço está preparado para agregar os vários pedidos? Fornece políticas de conversação?

A base da definição de conversação, explicada no artigo [Hoh07], é construída através da conjugação de vários elementos: os papéis dos participantes, a semântica das mensagens e protocolo de comunicação.

No mesmo artigo, são apresentados os padrões de conversação, que servem como guias para desenhar conversações robustas e efectivas entre serviços. Os padrões encontrados variam entre primitivas de comunicações de baixo nível, até as primitivas de comunicação existentes em negociações de alto-nível (*business-negotation*).

De seguida são apresentados alguns dos padrões, divididos nas diferentes áreas relacionadas com o processo de interacção entre serviços. É dado um exemplo para cada. Para informação mais detalhada o artigo mencionado mostra os vários padrões com imagens associadas:

Discovery - Antes de um serviço comunicar com outro tem de identificar os parceiros de conversação. Numa primeira fase, os serviços têm que se identificar e, numa segunda, os papeís têm de ser identificados. Um exemplo é o padrão "descoberta dinâmica"*(dynamic discovery)*. O cliente envia um pedido à rede, indicando que tipo de serviços pretende. Todos os fornecedores interpretam o pedido e todos os que poderem responder enviam uma resposta. A figura 2.7 apresentada no artigo [Hoh07], descreve o processo de descoberta, que é realizado através dos seguintes passos:

1 - Envio do pedido *(broadcast)*

2 - Os fornecedores interpretam o pedido e consideram se respondem

3 - Fornecedores interessados enviam as respostas

4 - O consumidor escolhe qual a melhor resposta

5 - O consumidor inicia a interacção com o fornecedor escolhido

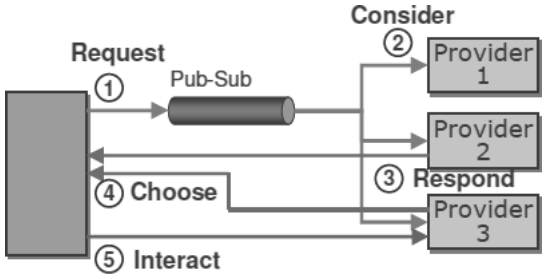


Figura 2.7: Descoberta dinâmica [Hoh07]

Establishing a Conversation - Assim que é conhecido o parceiro de comunicação, a conversação têm de se estabelecer. Um exemplo padrão é o *Three-way Handshake*. Existem três fases: o consumidor inicia a conversação: recebe o *acknowledge* do fornecedor; o consumidor envia de volta outro *acknowledge*.

Basic Conversations - Os padrões básicos estabelecem o vocabulário de base para descrever conversações mais complexas entre dois serviços. Um exemplo de um padrão base é *Request-Response*: um serviço envia uma mensagem de tipo pedido e recebe outra mensagem com a resposta. Outro padrão útil será o padrão *Subscribe-Notify*: o serviço interessado envia um pedido e o fornecedor, que tiver a resposta, envia um conjunto de mensagens de notificação.

Multi-Party Conversations - Enquanto que as *Basic Conversations* envolvem só dois serviços, neste caso envolvem 3 ou mais serviços. Um exemplo, muito conhecido, é o padrão Proxy. Um serviço transmite os seus pedidos para um *proxy* que encaminha as mensagens para o(s) serviço fornecedor.

Reaching Agreement - Para muitas interacções, chegar a um acordo é importante para múltiplos serviços independentes. Um exemplo é *Sender Cancels*: o fornecedor pode enviar uma mensagem de cancelamento ou ao fim de um intervalo de tempo parar.

Terminating Conversations - Quando múltiplos serviços estão ligados entre si e quando um deles perde a ligação, todos os outros, para pouparem recursos, devem terminar a conversação.

Error Handling - Nem tudo corre como planeado, especialmente em sistemas altamente distribuídos onde a comunicação dos serviços podem estar nos mais variados locais. Um padrão comum é *Renewal Reminder*: Dentro de um período de tempo, o fornecedor pergunta ao consumidor se ainda necessita dos recursos; caso este não responda, ou responda "não", os recursos são redistribuídos (*re-alocated*).

2.4.5 *Casos de uso*

Existem muitos casos particulares de aplicações distribuídas que aplicam muitos destes padrões. Alguns mostram que é possível utilizar o paradigma da programação orientada

aos serviços de forma a melhorar a usabilidade e adaptabilidade como é o caso do modelo SOCK [GM09]. Neste exemplo particular, o caso de estudo são as características e os mecanismos dos *web services*. O modelo que simplifica e especifica técnicas base e, ao mesmo tempo, garante suportes do desenvolvimento de ferramentas de desenho e de implementação de sistemas orientados a serviços O próprio Will M.P. van der Aaslt e a sua equipa desenvolveram um projecto [MAVDA07] em que são aplicados vários destes padrões de conversação, tanto de interacção como de comunicação. É exemplificada uma conversação configurável com multi-serviços, multi-mensagens e usa o padrão pedido-resposta. Através de diagramas em UML e de CPN (*coloured petri nets*) são propostas soluções para interacções pedido-resposta com multi-serviços e multi-respostas utilizando os padrões de comunicação.

A próxima secção descreve o conceito de reconfiguração dinâmica que é crucial para os *workflows* de *web services*, como forma de se adaptarem a alterações do sistema ou a novos requisitos do utilizador.

2.5 Reconfiguração dinâmica / Sistemas auto-adaptáveis

Os sistemas de computação chegaram a um tal nível de complexidade onde o esforço humano necessário para manter o seu funcionamento correcto e continuado é impraticável. A computação autónoma surgiu para facilitar e melhorar esse esforço, ambicionando o decrescente envolvimento humano nessa tarefa. O termo computação autónoma descreve sistemas computacionais com auto-gestão, isto é, sistemas que se auto-configuram, auto-optimizam, auto-protegem-se e auto-corrigem-se [KC03].

Entre muitas definições de adaptação dinâmica uma delas é fornecida pela *DARPA Broad Agency Announcement (BAA): "Self–adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."*.

Basicamente a citação tenta transmitir que o ponto-chave da computação autónoma começa com a capacidade de lidar, continuamente, com as alterações externas e com a adição de novos requisitos não planeados no processo de criação do sistema. Dentro desse ponto de partida, a reconfiguração ou adaptação dinâmica faz parte da computação autónoma.

Reconfiguração dinâmica representa o comportamento de evolução e adaptação em tempo real *(run-time)*, e em resposta a estímulos externos. Tais adaptações englobam níveis como por exemplo: desempenho, segurança e gestão de falhas [KC03]. O ideal seria que esse comportamento fosse imperceptível aos utilizadores (ou aos envolvidos) e o impacto na execução do sistema fosse mínimo (por exemplo não ser necessário reiniciar).

Para suportar esta característica é necessário que o ciclo de vida do sistema dividido em análise, desenho, implementação, testes e manutenção, não pare depois de terminada a fase de implementação. Após a instalação, o ciclo terá de ser mantido de forma apropriada, de maneira a que se adapte e, evolua com as mudanças externas. Habitualmente

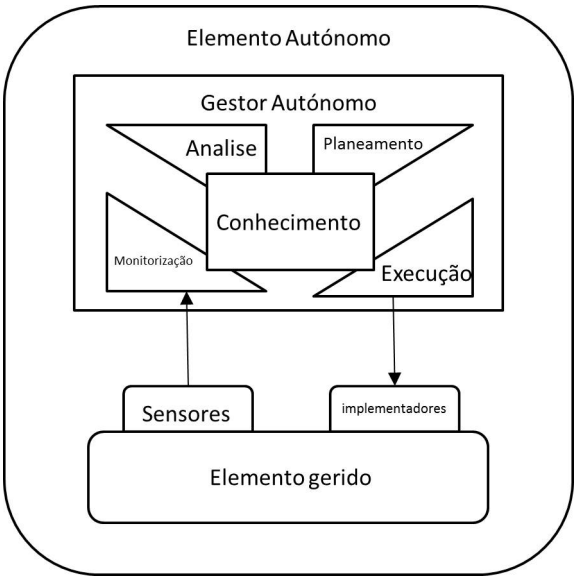


Figura 2.8: *MAPE-K*

nestes casos, o ciclo de vida é representado como um ciclo (*loop*) fechado que ajuda a adaptação dinâmica. Este ciclo fechado suporta mudanças a nível de optimização, configuração, segurança e auto-correcção.

2.5.1 *MAPE-K*

Para atingir computação autónoma com um loop adaptativo, a *IBM* sugeriu, um modelo de referência denominado *loop MAPE-K (Monitor,Analyse, Plan, Execute, Knowledge)* [figura 2.8], que é cada vez mais usado na representação de uma arquitectura de um sistema autónomo. Existem alguns exemplos, tais como, *Autonomic Toolkit* [5] , *ABLE*, etc.

O ciclo *MAPE-K* consiste em vários processos, auxiliados por sensores e implementadores (*effectors*), que são executados numa forma ordenada para que um sistema se adapte a alteração provocada por estímulos externos. Como representado na figura 2.8 um *loop* autonómico é constituído pelos seguintes componentes:

Elemento de controlo representado por sensores e implementadores associados ao elemento gerido, onde é recebido o estímulo externo e é realizada a acção.

Gestor autónomo onde os principais processos do ciclo autonómico são executados. Após receber o estímulo do sensor é percorrido o ciclo, Monitorização-Analise-Planeamento-Execução (utilizando Conhecimento) onde a acção associada será executada pelos implementadores.

Os principais processos do gestor autónomo no ciclo *MAPE-K* são:

[5] Autonomic Computing Toolkit Home page: http://www-128.ibm.com/developerworks/ autonomic/overview.html

O processo de monitorização é responsável por recolher e interpretar os dados dos sensores e convertê-los em padrões de comportamento. Este processo é suportado por disciplinas como ambientes *Wireless Sensor Networks (WSN's)* ou sistemas distribuídos.

O processo de análise e de detecção é responsável por analisar os sintomas fornecidos pelo processo de monitorização, de maneira a detectar que resposta será necessária. Também ajuda a identificar a origem de um novo estado.

O processo de planeamento determina o que tem de ser mudado e como mudar para atingir a melhor forma de ultrapassar o problema.

O processo de execução é responsável por aplicar as acções determinadas no processo de planeamento, utilizando os implementadores para esse efeito.

Conhecimento fornecido pelos programadores que desenvolvem este ciclo, em certos casos, utilizando técnicas de aprendizagem automática, por exemplo, aprendizagem reforçada em que, através de tentativa e erro, o sistema aprende por si próprio.

Como observado, os componentes do ciclo *MAPE-K* podem ser estruturados numa forma hierárquica o que pode originar muitos níveis de abstracção. Consequentemente, é necessário utilizar métodos de engenharia de software e teorias para melhor gerir essa abstracção e garantir o dinamismo necessário. No entanto, esses métodos e teorias não devem afectar o dinamismo do sistema e para isso surgem *trade-offs*.

Por exemplo, as adaptações dos componentes para optimizar o serviço não devem produzir interacções que possam provocar comportamentos indesejáveis a outros componentes. Para isso, é necessário criar comunicações num modelo descentralizado, o que indica para estes casos, o foco da implementação situa-se em soluções distribuídas.

No âmbito da tese, é explorado a articulação entre algumas das fases do ciclo autonómico, e o conceito de padrão representando modelo de comportamento e coordenação recorrentes (i.e. comuns em sistemas distribuídos). Alguns desses padrões foram apresentados nas secções anteriores.

2.6 Reconfiguração em *Workflows*

O dinamismo (ou reconfiguração) de sistemas de *workflow* é um tópico recorrente nos dias de hoje. A necessidade de se poder realizar alterações sobre *workflows* em execução tem sido um grande desafio. Com o crescente número de *workflows* científicos essa necessidade tornou-se ainda maior. Ao contrário de *workflows* da área de negócio, estes *workflows* necessitam de execuções em larga escala durante um largo período de tempo. Estes *workflows* podem ser usados para automatizar actividades repetitivas no domínio cientifico (e.g. Biologia, Física, etc), como por exemplo, acesso a dados, integração, transformações, análise e visualização de dados, etc. Estes automatismos garantem aos cientistas total foco no seu trabalho. Claro que a eficácia destes sistemas depende das ligações

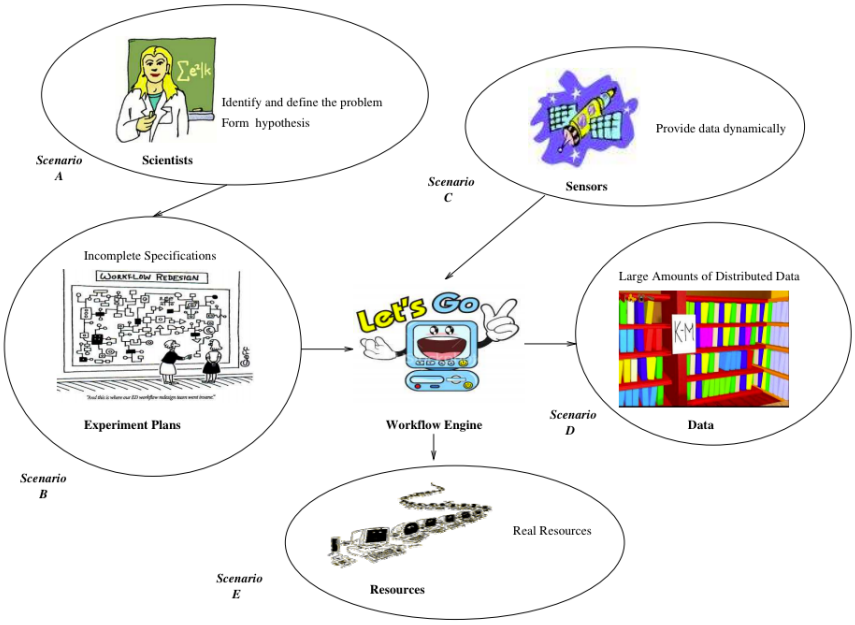


Figura 2.9: Exemplos de *workflows* científicos e razões para dinamismo (retirado do artigo [CRPN08])

e dependências criadas com outros serviços ou arquitecturas. Por exemplo, certos trabalhos podem ser distribuídos por *clusters* de computadores ou até serviços na *cloud*.

Outra grande diferença entre estes dois tipos de *workflow* é o seu paradigma de execução. Os *workflows* científicos são tipicamente orientados a fluxo de dados, i.e., a execução de certas tarefas dependem da produção de dados de outras. Em relação aos *workflows* da área de negócio são mais orientados a fluxo de controlo, ou seja, as tarefas executam pela ordem e composição criada no momento da composição do *workflow*. Esta diferença é relevante visto que a execução de *workflows* científicos é orientada pelos valores produzidos, enquanto que a execução de *workflow* da área de negócios é orientada pela definição do processo bem definido e pela sua estrutura estável.

Assim, no domínio de *workflows* científicos, o artigo [CRPN08] realizou um estudo onde ilustra possíveis cenários na área cientifica da importância do dinamismo, exemplificando posteriormente que requisitos devem ter esses *workflows* e que técnicas podem ser usadas para criar mecanismos que garantam esse dinamismo.

A Figura 2.9 ilustra possíveis cenários reais, no contexto científico, e as razões para as quais seria necessário existir dinamismo (ou reconfiguração). Para ajudar a garantir dinamismo em *workflows* científicos, os autores deste artigo agruparam alguns requisitos que os *workflows* devem suportar: requisitos de mudança e de usabilidade.

Os requisitos de mudança englobam mudanças na composição e execução de um *workflow* em tempo de execução ou no processo de desenho de *workflows*. Dentro desses requisitos os principais são:

1. Modificar a composição do *workflow* em tempo de execução;

2. Possibilidade de criar *workflows* abstractos e transforma-los em concretos realizando as devidas alterações (aplicando regras cientificas);

3. Gestão de dados (adicionar uma nova fonte de dados);

4. Mudança de recursos(por exemplo ser tolerância a falhas)

5. Mudança de serviços em tempo real, para o caso de adquirir fontes mais rentáveis (poder computacional ou menos custos).

Os requisitos de usabilidade são requisitos que oferecem ao utilizador facilidade na observação e gestão do estado e progresso da execução do *workflow*, alguns exemplos são:

Monitorização da execução Através de um *interface* ou, por exemplo, com *outputs*;

Controlo automático Notificar os utilizadores em casos de falha, estar atento a novas fontes de dados como por exemplo o cenário C da Figura 2.9;

Reprodutibilidade No momento em que é alterada, em tempo dinâmico, a estrutura é necessário disponibilizar informação sobre essas alterações;

Execuções "espertas" (*"Smart" re-runs*) Ser possível parar uma execução e retomá-la num dado ponto, útil para casos de falha para não retomar de inicio a execução;

Steering Possibilidade de parar, retomar,pausar a execução num dado ponto;

Modificações do utilizador Oferecer *interfaces* intuitivas para utilizadores não-peritos de forma a adicionar novas tarefas numa forma mais eficaz, por exemplo no cenário A poder inserir novas tarefas para criar novas hipóteses;

Adaptações sobre o *workflow* Possibilidade de adicionar tarefas abstractas e mudá-las no momento de desenho de *workflows*;

Adaptações sobre a execução do *workflow* Realizar adaptações durante a execução de um *workflow*.

Com estes requisitos foram analisadas técnicas para garantir dinamismo em *workflows*, estando essas classificadas em dois grupos:

1. Técnicas ao nível da composição de *workflows* (linguagem de modelação).

   (a) Modelação e desenho hierárquico, garantindo assim formas de controlar a complexidade dos *workflows* e de suportar alguns dos requisitos para dinamismo (dividindo por exemplo o *workflow* por diferentes hierarquias seria mais acessível aplicar alguns do requisitos de usabilidade (verificar o estado))

   (b) Especificar numa forma abstracta os requisitos das tarefas, assim em tempo de execução pode-se configurar a tarefa.

(c) Descrição semântica da tarefa, de forma a que a tarefa se adapte dependendo do estado da execução.

(d) Tarefas abstractas, oferecendo *empty task* no momento de desenho de forma a reutilizar o *workflow*. Antes de executar tem de ter uma acção associada.

2. Técnicas ao nível da execução.

(a) Técnicas relacionadas com tolerância a falhas, por exemplo ao nível da tarefa: tentar novamente a execução, ter um recurso alternativo, existir *checkpoints* para retomar esse passo, replicação da tarefa. Em relação ao nível do *workflow*: tarefa alternativa, redundância, o utilizador poder manipular as excepções ou *workflows* de resgate.

(b) Redefinição do *workflow* dependendo dos dados obtidos, por exemplo re-ordenar as tarefas, inserir novas tarefas, substituição do *workflow* ou de tarefas.

(c) Interpretação da execução, i.e., depende do tipo de execução, se é uma execução compilada ou que se vai executando passo. Claramente esta última oferece maior dinamismo podendo os passos serem alterados.

(d) *Checkpointing*, possibilidade de voltar atrás da execução.

(e) Proveniência, dar a possibilidade ao utilizador de persistir certos dados ao longo da execução.

Algumas destas técnicas já são aplicadas em ferramentas, tais como, *Askalon*[6] que oferece a possibilidade de criar *workflows* abstractos e a existência de *checkpoints* durante a execução. Na ferramenta *Kepler*[LAB+06] existe o conceito de mutação que oferece a mudança da estrutura do *workflow* em tempo de execução. Uma maior descrição para estas ferramentas está no artigo mencionado[CRPN08], para além da análise a outras ferramenta.

No âmbito desta tese, o conceito de padrão ajuda a simplificar e a garantir o uso de algumas das técnicas apresentadas.

[6] http://www.dps.uibk.ac.at/projects/brokerage/

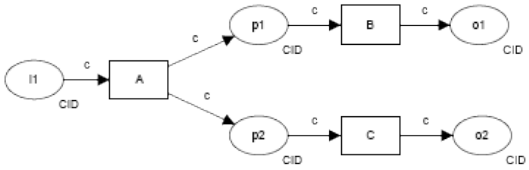
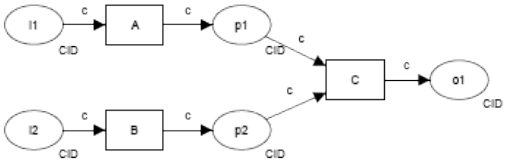
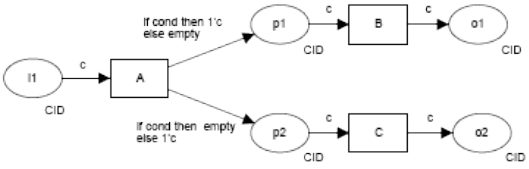
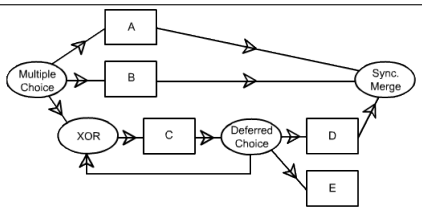
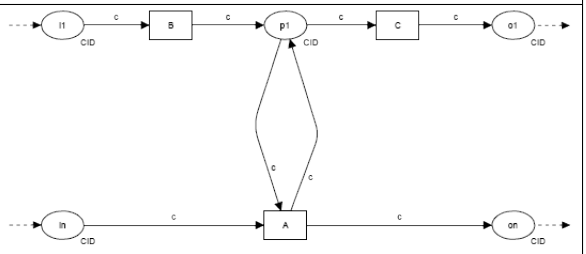
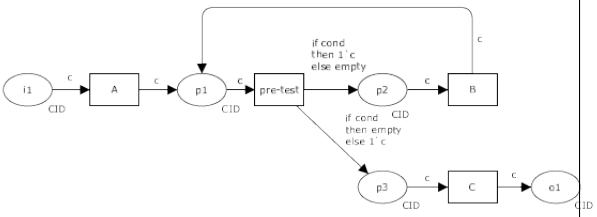
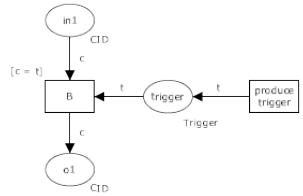
Nome	Descrição	Figura
(1) Sequência	Executa duas ou mais tarefas em sequência.	
(2) Divisão paralela	Executa dois ou mais caminhos(que inclui tarefas) em qualquer ordem ou em paralelo.	
(3) Sincronismo	Convergência de dois ou mais ramos para um único, para sincronizar duas ou mais tarefas que podem ser executadas em ordem ou em paralelo.	
(4) Escolha exclusiva	Escolhe um caminho dentro de vários, dependendo da análise aos dados quando chega ao ponto de escolha exclusiva.	
(5) Junção sincronizada geral	Ideal para usar em composição de tarefas não estruturadas mas altamente concorrentes, podendo existir estruturas em *loop*	
(6) *Milestone*	Fornece um mecanismo que suporta a execução condicional de uma tarefa (p1) quando a instância do processo está num dado estado. Neste caso, estado é quando o fluxo de controlo chegou a um ponto específico (A).	
(7) *Loop* Estruturado	A habilidade de executar uma tarefa ou um sub-processo repetidamente. O *loop* tem uma única entrada e saída.	
(8) *Trigger* persistente	A habilidade de uma tarefa ser accionada por sinal ou por uma tarefa fora do seu contexto ou ou por entidade externas. Os *triggers* mantém-se activos até serem actuados.	

Tabela 2.3: Alguns padrões de controlo de fluxo referidos na tabela 2.2

3

Reconfiguração dinâmica estruturada de *workflows*

Este capítulo descreve o modelo subjacente à solução apresentada neste trabalho, o levantamento tecnológico feito que conduziu à escolha da ferramenta Activiti, bem como a descrição desta ferramenta.

3.1 Solução Proposta

A solução proposta nesta tese baseia-se no conceito de padrão, visto que os padrões são aplicados como formas estruturadas de interacção entre tarefas num *workflow*, e servem de base aos mecanismos de reconfiguração dinâmica suportados.

A solução proposta segue a abordagem apresentada em [GRC08, Gom07], isto é: os padrões são disponibilizados como entidades de primeiro nível, correspondendo a elementos computacionais com o seu próprio ciclo de vida, permite que eles sejam manipulados individualmente, bem como combinados com outros conceitos de primeiro nível (por exemplo os padrões podem ser vistos como componentes num sistema baseado em componentes).

Neste caso, manipulação de padrões pode ser feita desde a fase de desenho da aplicação, até à fase de execução. Na fase de desenho são usados padrões de desenho [GHJV95], quer estruturais quer comportamentais, na forma de *templates* instanciáveis (com componentes, serviços, outros *templates* de padrões, etc). Esses *templates* podem ser refinados, parametrizados, e combinados em estruturas de desenho mais complexas através de operadores próprios.

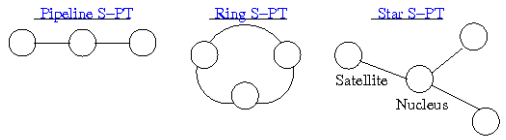


Figura 3.1: Padrões de estrutura.

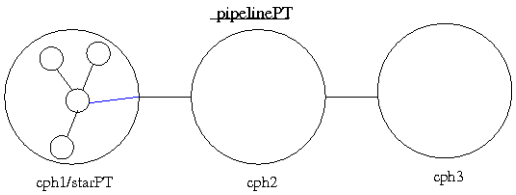


Figura 3.2: Hierarquia de padrões de estrutura.

Na fase de execução, esses mesmos padrões representam, por sua vez, a agregação estruturada (em termos de dependência de dados e controlo) de um conjunto de unidades executantes (componentes em execução, tarefas num *workflow*, etc). Sendo as decisões tomadas em tempo de desenho ainda visíveis em tempo de execução, torna-se mais fácil identificar como reconfigurar a aplicação. Isto porque, como estes padrões são entidades de primeiro nível também em tempo de execução, a sua manipulação (por exemplo suspender a sua execução, substituir um padrão por outro semanticamente/funcionalmente equivalente, de alterar o número de elementos que os constituem), permite um controlo da execução e uma reconfiguração "por padrão", mesmo dentro de uma estrutura hierárquica, tal como descrito em [GRC08, Gom07].

3.1.1 Padrões de estrutura e comportamento

Os padrões de estrutura implementados foram as topologias estrela, anel, e *pipeline*, apresentadas na Figura 3.1, na forma de *templates* de padrões instanciáveis. Por exemplo, a Figura 3.2 apresenta um padrão de estrutura hierárquico, em que a primeira etapa do *pipeline* foi instanciado com um *template* em estrela.

A composição de um padrão de comportamento com um de estrutura, consiste na anotação de cada elemento da topologia com um papel no contexto comportamento escolhido. A divisão entre padrões de estrutura e comportamento permite, por sua vez, compor uma mesma estrutura com dois padrões de comportamento distintos, tal como apresentado na Figura 3.3 e, inversamente, aplicar o mesmo padrão a diferentes estruturas.

Esta divisão permite flexibilidade na composição dos padrões, mas também reconfigurar cada tipo de padrão em separado. No entanto, as reconfigurações possíveis são conformes/obedecem à semântica dos padrões reconfigurados. Por exemplo, qualquer alteração a um *pipeline* tem de produzir como resultado um *pipeline*).

Para mais, e por simplificação, existe a restrição que o mesmo padrão de comportamento é aplicado a todos os elementos de uma estrutura, tal como apresentado na Figura

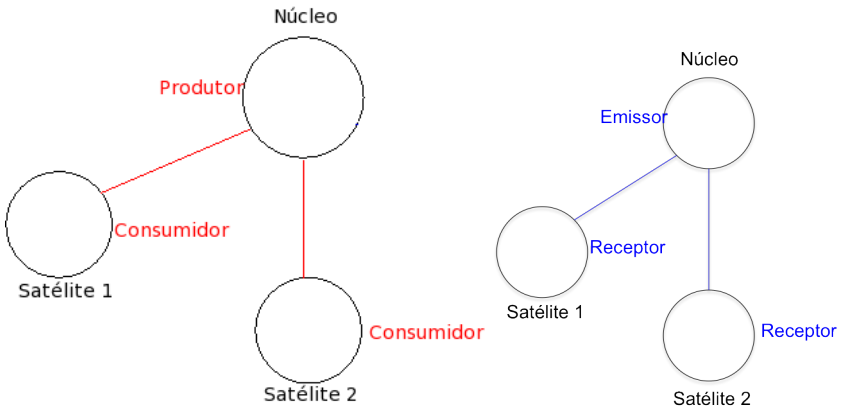


Figura 3.3: Uma instância particular da topologia estrela combinada com dois padrões de comportamento distintos.

3.3. Do mesmo modo, a substituição de um comportamento por outro no contexto de um padrão de estrutura, implica que as anotações de comportamento são substituídas em todos os elementos da estrutura, de acordo o que está definido para o novo padrão. Por exemplo, como ilustrado na Figura 3.3, passar de um comportamento *streaming* para um comportamento produtor/consumidor, implica que todos os satélites passam a ser consumidores.

Finalmente, os padrões de fluxo de dados apenas foram implementados no contexto nos padrões de estrutura, e não na ligação entre duas tarefas de um *workflow*, dado o tempo limitado desta tese de mestrado.

Descreve-se de seguida de como foi definida a especificação destes padrões, no contexto de um *workflow* e, em particular da ferramenta Activiti. Na secção 3.2 é feita uma descrição detalhada desta ferramenta.

3.1.2 Especificação de Worflows nas áreas de negócio e científica

A Figura 3.4 apresenta o significado das dependências entre tarefas num *workflow* abstracto, tal como descrito em [YGN09]. No topo da figura é apresentado um *workflow* abstracto, que apresenta um significado semelhante, na perspectiva do utilizador, tanto na área científica como na área de negócio. No entanto, tal como os autores ilustram na figura, no caso da área científica, as dependências representam dependências de fluxos de dados/*dataflow*, ou seja, uma tarefa B depende de dados gerados por A. Adicionalmente, as tarefas A e B podem ser executadas em paralelo, embora muitas vezes o modelo *dataflow* defina que B só pode executar se tiver disponíveis todos os dados de *input* de que necessita. As dependências na área de negócio, por sua vez, são do tipo fluxo de controlo/*control-flow*, ou seja uma tarefa B só pode executar depois de A terminar, e as tarefas só podem executar em paralelo se estiverem em caminhos distintos do *workflow*.

Na proposta apresentada nesta dissertação, a notação usada para representar os dois

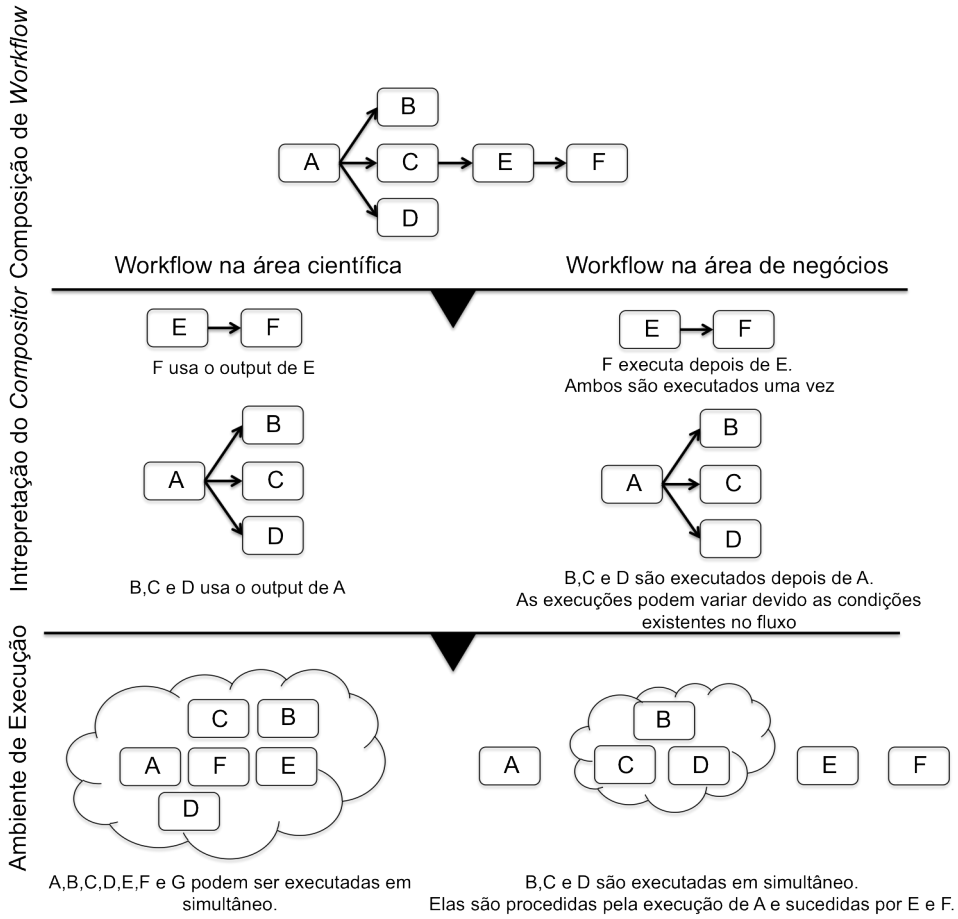


Figura 3.4: Significado das dependências entre tarefas num *workflow* genérico. Figura adaptada de [YGN09]

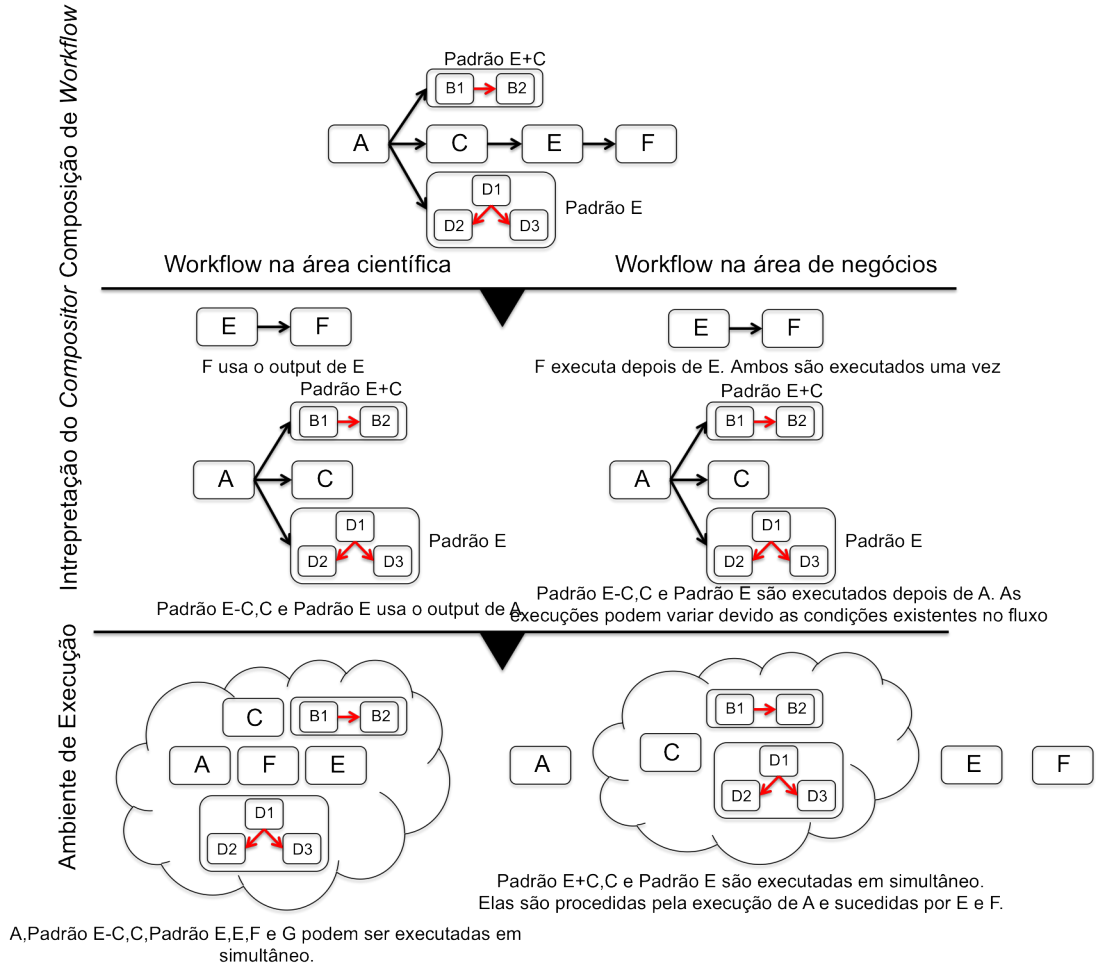


Figura 3.5: Significado das dependências entre tarefas num *workflow* genérico, no usado no contexto neste trabalho. Figura adaptada de [YGN09]

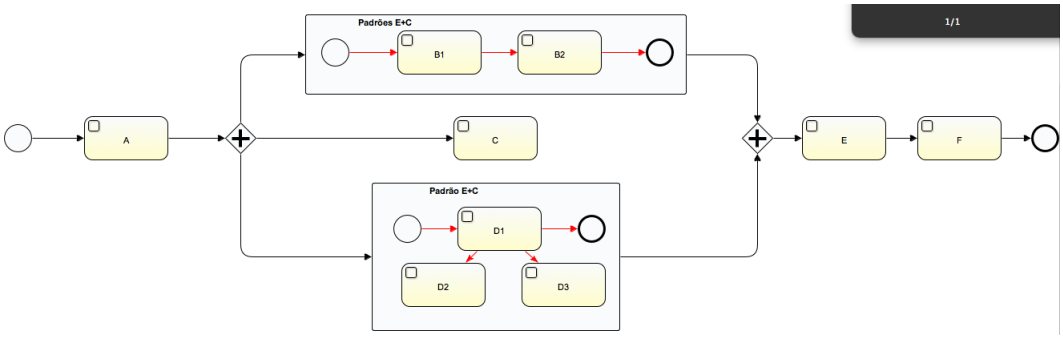


Figura 3.6: Especificação de dependências segundo o modelo *control-flow* e *dataflow*, na ferramenta Activiti estendida.

casos está ilustrada no *workflow* no topo da Figura 3.5: as dependências no *workflow* geral são de fluxo de controlo (a preto) e as dependências em cada padrão são de fluxo de dados (a vermelho), i.e. correspondem às dependências definidas de acordo com o padrão de comportamento usado no contexto desses padrões de estrutura. Na Figura 3.6 é apresentado como esse *workflow* no topo da Figura 3.5 é definido na ferramenta Activiti estendida.

Compreendemos que a notação usada não é a mais clara, nem tem correspondência na notação BPMN, situação esta que pretendemos corrigir em trabalho futuro.

Segue-se uma descrição da arquitectura da solução proposta.

3.1.3 Arquitectura da Solução

No contexto desta tese, é considerado um conjunto limitado de possibilidades no que diz respeito à reconfiguração dinâmica, sendo definidos alguns pressupostos:

a) Um *workflow* pode ser visto como um conjunto de *sub-workflows*.

b) Cada *workflow* pode ser estruturado/coordenado de acordo com um número limitado de padrões.

c) As reconfigurações possíveis ao nível de cada padrão são limitadas e específicas desse padrão (ex. alterar o número de tarefas que constituem um padrão do tipo estrela, corresponde a aumentar o número de satélites);

d) A evolução/adaptação do sistema faz-se com base nas reconfigurações em c); para mais, um padrão de coordenação/comportamento pode ser substituído por outro (ex. mudar a interacção *Streaming* entre tarefas num *workflow* para um modelo de interacção Produtor/Consumidor);

e) Os pontos onde as reconfigurações definidas em d) são permitidas, estão bem definidos à partida, pretendendo-se assim minorar o impacto da reconfiguração.

f) Idealmente, a reconfiguração pode ser despoletada quer a pedido de um cliente, quer automaticamente; neste caso, a reconfiguração pode ser desencadeada com base no estado dos serviços e/ou da interacção dos clientes com os serviços.

Tendo em vista a criação de um protótipo que garanta a reconfiguração dinâmica baseada em padrões referida, é necessário disponibilizar um ambiente que permite composição de serviços, fracamente acoplados, num *Workflow*. Os mecanismos necessários para reconfigurar as interacções entre serviços são parametrizados nessa camada.

Como ajuda e olhando para a Figura 3.7, verifica-se que a camada deve suportar a execução de *Workflows* de *Web Services*, através de *Workflow Engine*, bem como interagir com os serviços segundo o estado de um *Workflow*. Para a composição entre serviços é utilizada a notação *BPMN*, sendo a camada "motor de execução" da notação a responsável pela execução dos comportamento e reconfigurações pretendidas.

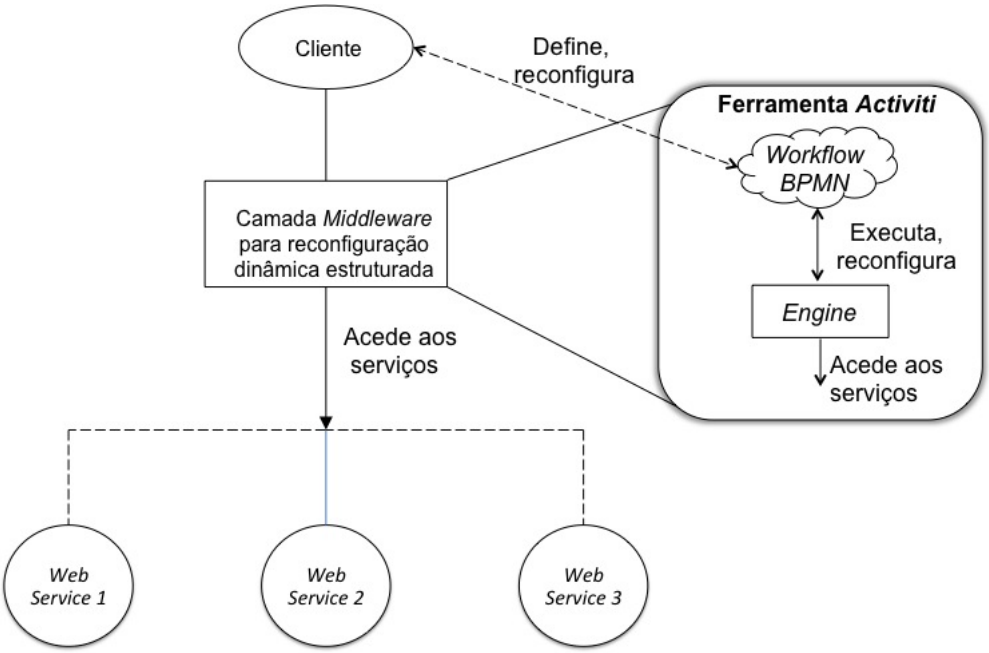


Figura 3.7: Arquitectura proposta

Adicionalmente, considera-se que as adaptações dinâmicas são realizadas a pedido pelo utilizador, sempre que este achar que as alterações trazem benefícios para o sistema. As alterações dinâmicas automáticas teriam como base, por exemplo, alterações resultantes da interacção com os serviços. No entanto, devido a limitações de tempo tal não foi possível ser implementado no contexto desta tese.

A ferramenta de *workflow* escolhida, o *Activiti*, permite a especificação de *workflows* em *BPMN*, e foi estendida com padrões de estrutura e de fluxo de dados. Como o *Activiti* também inclui um motor de execução (*engine*), a execução dos *workflows* instanciados (com ou sem padrões) é também realizada no contexto do *Activiti*.

Resumindo, os objectivos gerais considerados para o protótipo foram: i) definir *Workflows* de *Web Services* construídos com base em *templates* de padrões de estrutura e comportamento; ii) aplicar as reconfigurações em tempo estático, ou em tempo de execução, tanto na dimensão de estrutura como de comportamento.

3.1.4 Sistemas de gestão de *workflows* (*WfMs*) analisados

O primeiro passo para atingir o objectivo proposto passou pela definição de um modelo de suporte à construção de um protótipo de composição estruturada de serviços. Neste seguimento, o modelo teórico apresentado em [ACKM04] foi escolhido como base para o nosso trabalho. Este modelo apresenta quatro componentes necessárias para a construção de um sistema adaptável e de fácil utilização para a composição de serviços. O esquema, representado na figura 3.8, ilustra essas componentes e suas interacções que a seguir se descrevem:

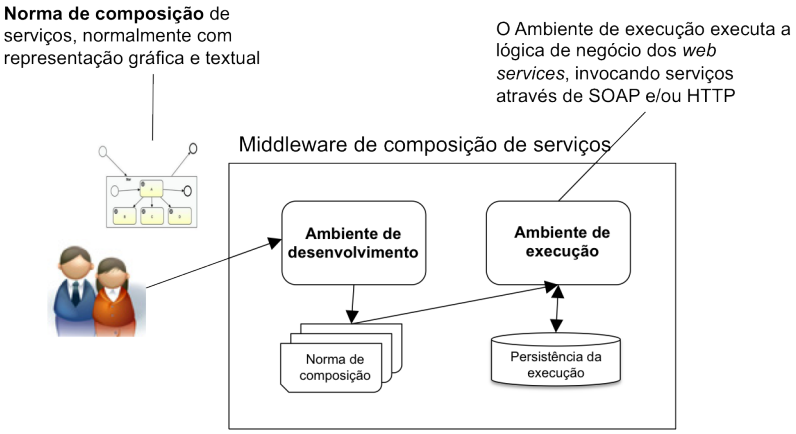


Figura 3.8: Normas fundamentais de um protótipo para composição de serviços[ACKM04]

Norma de composição Tem várias definições: linguagem, notação ou *schema*; o seu objectivo é representar uma especificação na composição de serviços, normalmente a lógica de negócio. Serve também como elo de ligação entre o ambiente de desenvolvimento e o de execução.

Ambiente de desenvolvimento (edição) Habitualmente representado por uma interface do utilizador (UI) tem como objectivo disponibilizar ferramentas ao utilizador para o desenho de composição de serviços, segundo uma notação pré-definida.

Ambiente de execução (ou motor de execução) Executa uma dada lógica de negócio definida na notação de composição. Recorre a uma base de dados para garantir a persistência de execução.

Persistência da execução Na maioria dos casos, a persistência de execução é representada por uma base de dados. Nela são guardadas as variáveis envolvidas na composição e interacção entre serviços.

Com o modelo teórico de base, o próximo próximo passo foi encontrar uma tecnologia que o suportasse. Para tal foi efectuado um levantamento tecnológico da área com o intuito de encontrar a ferramenta que melhor satisfizesse os nossos requisitos.

O levantamento tecnológico realizado incidiu sobre duas vertentes: a componente de composição e ferramentas de suporte ao desenvolvimento e à execução dos *workflows*. Nesta secção descreve-se a avaliação da notação/linguagem e respectiva ferramenta que permite suportar o modelo anteriormente mencionado. Foram analisadas várias linguagens com o pré-requisito de, pela sua estrutura de suporte, conterem uma componente de ambiente de execução.

Como já referimos no capítulo do estado da arte, verifica-se que tanto a linguagem *BPEL* como a linguagem *WS-CDL* demonstram que a composição e estruturação de *web services* em *workflows* é possível. No entanto, sendo ambas linguagens verbosas/descritivas, o tempo despendido no processo de construção de um *workflow* é algo elevado. Para

mais, um dos objectivos deste trabalho é fornecer um ambiente rico de desenvolvimento, e sendo ambas linguagens de execução, não existe uma especificação para a existência de uma componente ambiente de desenvolvimento.

Outra linguagem analisada foi a linguagem *YAWL*, criada pela equipa liderada por Wil M.P. van der Aalst [YAW]. Esta linguagem suporta a utilização de padrões de *workflow* no processo de composição, no entanto não é uma linguagem *standard* para a construção de *workflows* na área de negócio.

Estas conclusões levaram-nos a analisar em maior detalhe a notação *BPMN 2.0* que, para além de ser uma notação pertencente à organização OMG , é uma linguagem de modelação que tem tido uma crescente aceitação nessa área. Esta característica obriga a que esta notação tenha uma especificação, o que significa que todas as ferramentas que a suportam têm de fornecer um ambiente de desenvolvimento ao utilizador. De igual forma e de particular interesse para o nosso protótipo, esta notação suporta a invocação de *web services*, podendo esta, dependendo da ferramenta, ser realizada por execução de código.

Com estas conclusões, decidiu-se escolher esta notação como a norma de composição para o nosso trabalho.

3.1.4.1 Análise de ferramentas

No processo de análise das ferramentas, os requisitos necessários para a escolha de uma ferramenta foram divididos nos dois componentes seguintes:

1. Ambiente de desenvolvimento
   a) Suporte à notação *BPMN 2.0* (No seguimento das conclusões da secção anterior).
   b) Disponibilidade do código fonte para fins de extensibilidade.
   c) Existência ou facilidade de construção de padrões de *workflow*. [1]
2. Ambiente de execução
   a) Disponibilidade do código fonte.
   b) Funcionamento da ferramenta em diferentes plataformas.

Foram analisadas cinco ferramentas:

- Ferramenta YAWL, fornecida pela equipa liderada por W.M.P van der Aalst [YAW] que oferece um ambiente de desenvolvimento e execução, no entanto a notação usada é exclusiva;
- Ferramentas *jBoss*, *jBPM* [JBoa] para *BPMN 2.0* e *Riftsaw* [JBob] para *BPEL*;

[1] Este requisito foi tido em conta para perceber até que ponto podia simplificar o processo de construção de um *workflow*.

- Ferramenta Activiti [Act] que oferece vários ambientes de desenvolvimento, entre os quais, um *plugin* para o *eclipse*;
- E a a ferramenta para construção de *workflows* do *Eclipse*, Eclipse-JWT [JWT].

Todas as ferramentas consideradas são genéricas, isto é, o seu foco principal é o suporte a construção de *workflows* abstractos, garantido assim a sua aplicação em qualquer área de negócio.

Ferramenta	Ambiente desenvolvimento			Ambiente execução	
	BPMN 2.0.	Código fonte	Existência de padrões *workflow*	Código fonte	Várias plataformas
YAWL	-	-	*x*	-	*x*
jBPM	*x*	-	-	*x*	*x*
Riftsaw	-	-	-	*x*	*x*
Activiti	*x*	*x*	-	*x*	*x*
Eclipse-JWT	*x*	*x*	-	-	-

Tabela 3.1: Tabela de análise

Na tabela 3.1 são descritos os requisitos suportados em cada caso, a ferramenta *Activiti* foi a seleccionada dado que todos os requisitos são satisfeito excepto a inclusão de padrões de *workflow*. No entanto, esses padrões podem ser implementados, tal como referido pelo próprio Wil M.P. van der Aalst em[WvdAD+05].

A secção seguinte é dedicada à descrição da ferramenta *Activiti*.

3.2 *Activiti*

O principal objectivo da ferramenta *Activiti* é especificar processos de negócio em linguagem *BPMN 2.0* facilitando a colaboração quer de pessoas da área de negócio quer de responsáveis de desenvolvimento, através de uma comunidade de partilha de documentação e desenvolvimentos.

Esta ferramenta disponibiliza um número vasto de componentes e tem como base um motor de execução de processos *BPMN 2.0*. Estando consolidada em várias empresas, a ferramenta tem sofrido constantes evoluções nas suas funcionalidades. Embora seja direccionada para desenvolvimentos em linguagem *Java*, a ferramenta tem a particularidade de partilhar o código fonte e ser distribuída sobre licença *Apache* [2]. O *Activiti* contém componentes (Figura 3.10) que podem ser integradas em qualquer aplicação *Java*, logo ser instalada num servidor, *cluster* ou até mesmo em serviços *cloud* [Act]. Para esse efeito, as várias componentes estão dividas em três camadas independentes (ver Figura 3.10): camada de ferramentas dedicadas para especificar *workflows* BPMN 2.0; camada de repositórios e de execução de *workflows*; e ferramentas de colaboração entre intervenientes. Como as camadas são independentes entre si, a execução de um processo de negócio é definida por um ficheiro *XML* (ou *JSON*) contendo a informação *BPMN 2.0* necessária.

[2] Apache License, Version 2.0. http://www.apache.org/licenses/LICENSE-2.0

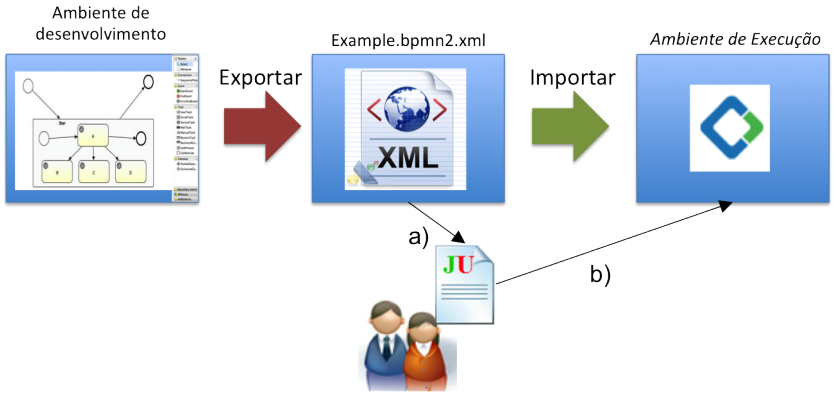


Figura 3.9: Interacção *UI* -> Motor

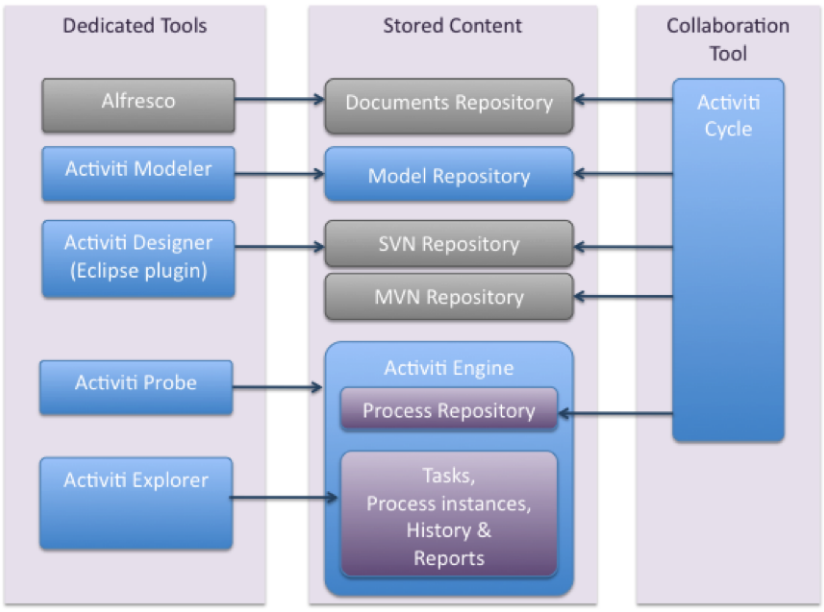


Figura 3.10: Componentes *Activiti* [Act]

Esta inclui, a informação gráfica e as regras de negócio a serem executadas. A figura 3.9 ilustra um exemplo de tal procedimento, a ligação entre uma ferramenta de especificação de *workflows* e a de execução usando um ficheiro *XML*. Este caso em particular representa a forma como é executado um *workflow*, na componente *plugin* do *eclipse*, que na secção 3.2.1 será explicado em maior detalhe.

Outra possibilidade é através da *API Activiti* que será explicada em maior detalhe na secção 3.2.2.

Tal como ilustrado na Figura 3.10, dentro destas três camadas observamos componentes pertencentes ou não ao escopo do *Activiti* cuja a descrição se segue:

- **Ferramentas de especificação de *workflows* ("*Dedicated Tool*")**

  ***Alfresco* (não pertence ao escopo)** Modulo para integração com ferramentas *Alfresco*

(ferramentas que servem para gestão de conteúdos como documentos, imagens, etc.).

Activiti Modeler Tem como objectivo apresentar um construtor complexo de diagramas (*workflows*) num *browser*. A informação *BPMN 2.0* é gravada num ficheiros especificados em formato *JSON* ou *XML*.

Activiti Designer (Eclipse Plugin) Componente construído em Java que tem como objectivo apresentar num *plugin eclipse* um construtor básico de diagramas e integrado com o motor de execução. A informação *BPMN 2.0* é gravada de igual em *JSON* ou *XML*

Activiti Probe Componente de administração e de monitorização de um processo BPM em execução.

Activiti Explorer Componente de visualização de informação envolvida na construção e execução de um processo *BPMN 2.0.* (estatísticas, acessos, duração, pessoas envolvidas, etc).

- **Repositórios e execução ("*Stored Content*")**

  Repositórios (certos repositórios não pertencem ao escopo) Repositórios onde qualquer pessoa pode aceder a informação, tanto ao código fonte como a documentação.

  Activiti Engine Componente fulcral para a execução de *workflows*, interpreta os ficheiros *XML* com notação *BPMN 2.0* e executa o diagrama construído.

- **Ferramentas de colaboração ("*Collaboration Tools*")**

  Activiti Cycle Componente focada em fornecer mecanismos de partilha de experiências e opiniões para ajudar a melhorar a ferramenta e os processos de negócio envolvidos.

Analisando as componentes *Activiti*, concluiu-se que a componente ambiente de desenvolvimento pode ser suportada por duas componentes *Activiti*: *Activiti Designer (Eclipse Plugin)* e *Activiti Modeler*. A escolha recaiu na componente *Activiti Designer (Eclipse Plugin)* devido a maior integração entre o ambiente de desenvolvimento e o ambiente de execução. Em relação à componente de ambiente de execução conclui-se que a componente *Activiti Engine* é a que mais se adequa aos nossos requisitos.

Nas próximas secções serão explicadas ambas as componentes e a sua respectiva extensibilidade a desenvolvimentos.

3.2.1 *Activiti Designer*

Esta componente tem como objectivo oferecer ao utilizador um ambiente de desenho de *workflows BPMN 2.0*. Segue-se uma descrição sobre: a) Elementos *BPMN 2.0* existentes; b) Que mecanismos são disponibilizados para facilitar a construção de *workflow*; c) Como

são realizados os testes de execução; d) Como é feita a interacção entre o ambiente de desenvolvimento e o de execução.

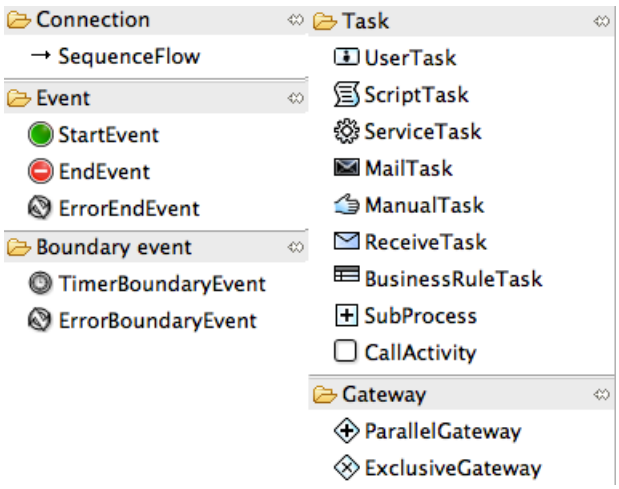


Figura 3.11: Elementos fornecidos pelo *plugin*

Elementos *BPMN 2.0* Estão disponibilizados os elementos básicos *BPMN 2.0*, referidos na secção 2.3.3, necessários para a composição de tarefas, tal como ilustrado na figura 3.11 onde está a palete de elementos oferecida. O utilizador basta arrastar o *icon* para a área de edição de *workflows* para inserir a tarefa.

Apesar da notação *BPMN 2.0* não suportar invocação directa de *web services* é fornecido um elemento de especial interesse para o nosso trabalho, a tarefa *Service Task*. Esta tarefa, como descrito na tabela 2.3.3 na secção 2.3.3, é utilizada para interacções externas à execução de um *workflow* (e.g. execução de código java). Assim a *service task* permite a ligação com *web services* tal como descrito na secção 4 dedicada a apresentação da solução propostas

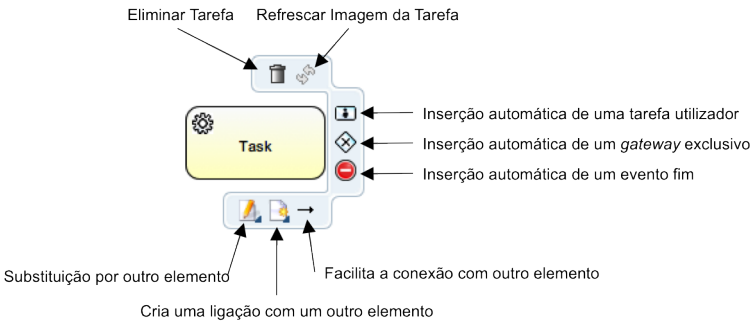


Figura 3.12: Menu de contexto num elemento

Construção de *workflows* Dentro dos mecanismos disponibilizados para facilitar a construção de *workflows* destacam-se os menus de contexto. Cada tarefa disponibiliza um

menu gráfico que permite a execução de várias acções sobre o elemento alvo no momento de construção, tal como ilustrado na figura 3.12.

A extensão das funcionalidades do *Activiti Designer*, traduz-se, em alguns casos, com a inclusão de novas acções ao nível do menu de contexto, tal como será descrito na secção 3.3.1.

Avaliação da execução Um *workflow* previamente construído e guardado como o ficheiro *XML* apresentado na secção 3.2, pode ser executado directamente a partir do *Activiti Designer*, através de um ficheiro *java unit test* que utiliza a framework JUnit. Como podemos verificar na Figura 3.9, ao exportar a informação *BPMN 2.0* para o ficheiro *XML*, esse é importado através da execução deste ficheiro de testes (a)). Ao executar o ficheiro de testes o ficheiro *XML* é interpretado para que o *workflow* seja executado.

Nesse ficheiro acedemos à *API Activiti* e podemos agilizar a construção de cenários teste acedendo aos serviços apresentados na secção 3.2.2. Um cenário pode ser, por exemplo, a execução de um processo repetidamente em intervalos de tempo, para assim criar uma análise temporal de resultados obtidos.

Ligação entre o ambiente de desenvolvimento e de execução Como descrito na secção anterior, o *Activiti Designer* está num ambiente integrado de desenvolvimento, contudo é necessário a geração de um ficheiro de ligação para executar o *workflow*. Esta geração ocorre após a construção do *workflow* e, para além de garantir a integração entre camadas, este ficheiro garante a persistência das parametrizações efectuadas. Nesse ficheiro está incluída a informação *BPMN 2.0*, juntamente com a informação gráfica *BPMN DI (BPMN Diagram)* e, a notação *XML* (*namespace*) específica do *Activiti* para representar regras de negócio. O ficheiro gerado pode ser reutilizado para construção de outros processos de negócio nesta ou noutra componente *Activiti*. A vantagem de conter a informação *BPMN DI* é permitir a visualização do processo em qualquer ferramenta de modelagem *BPMN 2.0*. No entanto, como as regras de negócio são construídas usando a notação *Activiti*, elas não podem ser executadas noutras ferramentas.

Por outro lado, para configurar funcionalidades e/ou serviços a serem executados pelo ambiente de execução, tais como, servidor de email para as *email tasks* ou o tipo de base de dados para *persistência*, existe o ficheiro *XML Activiti.cfg.xml*. Este ficheiro está incluído por omissão na definição de um processo de negócio.

Durante a execução do processo, cada elemento do *workflow* tem um escopo (contexto) de variáveis e configurações. Estas incluem, por exemplo: nome, identificador, se é tarefa temporizada ou/e assíncrona, valores de variáveis de ambiente que podem representar variáveis de escopo de outras tarefas, etc. Assim, um ponto interessante para a interacção é a existência de uma área de parametrizações para cada elemento. Esta área proporciona a possibilidade de alterar essas variáveis de forma a enriquecer um pouco a interacção com o utilizador. Na secção 3.3 será explicado de que forma é que estas parametrizações podem suportar a extensibilidade da ferramenta.

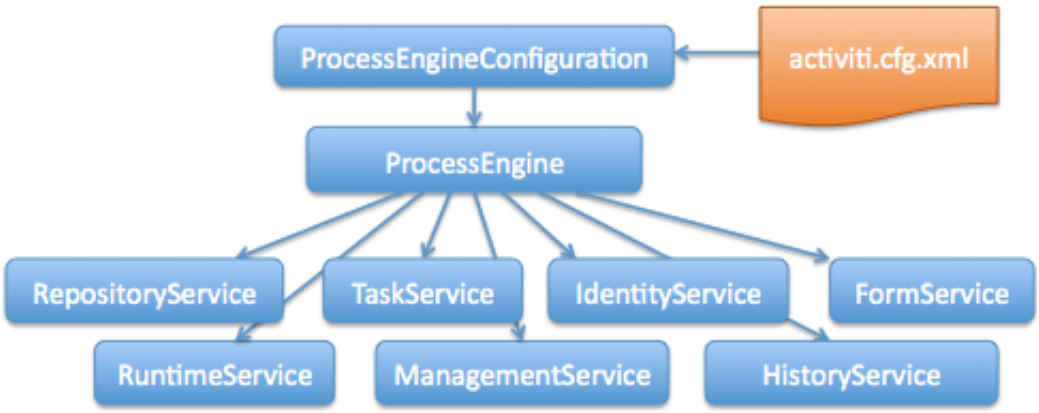


Figura 3.13: A *API* do motor

Concluindo, esta componente fornece: elementos básicos *BPMN 2.0*, sendo de especial interesse a tarefa *service task*; mecanismos de ajuda à construção de *workflows*; ambiente de testes através da utilização da *framework jUnit*; e formas de interacção com o ambiente de desenvolvimento e de execução.

De seguida será analisada a componente de ambiente de execução, *Activiti Engine*.

3.2.2 *Activiti Engine*

A componente *Activiti Engine* é o núcleo central das ferramentas de especificação de *workflows* do *Activiti*. Tem como objectivo executar as regras de negócio parametrizadas e capturar o fluxo de controlo de um *workflow*. Esta componente suporta a *API Activiti* que pode ser acedida para utilizar os vários serviços em execução. Segue-se uma descrição deste suporte bem como os passos do ciclo de funcionamento do *Activiti Engine*.

A figura 3.13 ilustra a arquitectura do *Activiti Engine*, a qual deve ser analisada de cima para baixo, em termos de descrição das etapas/passos a seguir enumerados.

O primeiro passo é a interpretação do ficheiro *Activiti.cfg.xml*, que configura o ambiente de execução com funcionalidades e/ou serviços escolhidos pelo utilizador (*ProcessEngineConfiguration*).

No segundo passo, e depois de aplicadas as configurações, é criada a instância do motor de execução que inicia e disponibiliza os serviços envolvidos (*ProcessEngine*). Segue-se uma breve descrição das funcionalidades oferecidas por cada um desses serviços:

RepositoryService Gere o *deploy* de *workflows*; Através do acesso a este serviço é possível adicionar ou remover *workflows* a instância do motor de execução.

RuntimeService Inicia/procura/obtém as instâncias de execução dos processos. Acede ao modelo de objectos das tarefas existentes num processo.

TaskService Expõe operações para as *user tasks* (referidas na secção 2.3.3)

IdentityService Serviço que gere o acesso dos utilizadores ou de grupos de utilizadores

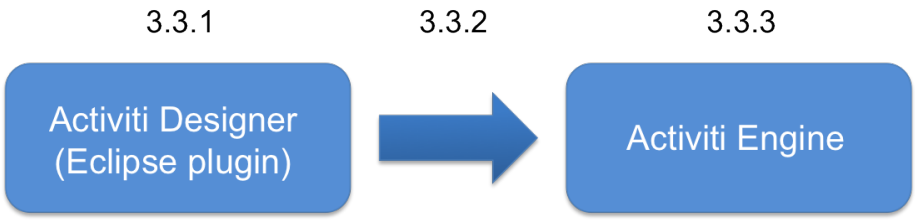


Figura 3.14: Estrutura secção extensibilidade

ManagementService Gere as tarefas temporizadas e assíncronas.

HistoryService Expõe informações de histórico sobre instancias de processos que estão ou que foram executadas.

FormService Acesso a dados de um formulário e renderização de formulários para instâncias de processos. Estes formulários só são utilizados nas componentes que executam num *browser*.

No terceiro passo, e depois dos serviços estarem disponibilizados, ocorre a análise ao ficheiro *XML* gerado pela *interface*. Esse ficheiro é validado com a utilização dos respectivos avaliadores da notação *Activiti* (que representa as regras de negócio) e da notação *BPMN 2.0*. Caso o ficheiro seja válido, são instanciadas as respectivas classes para cada elemento *BPMN 2.0*, tendo em conta as propriedades parametrizadas. Essas classes contêm o escopo do elemento, i.e, todas as parametrizações e especificações características de um elemento. Todas as instâncias são geridas pela *API Activiti* através do serviço *RuntimeService*.

O quarto e último passo, durante a instanciação dos elementos em objectos é-lhes associado um comportamento a ser executado aquando a sua activação (ou invocação). Por exemplo, quando o fluxo chega a uma *end task* o comportamento associado a esta tarefa é terminar a execução do processo.

Concluindo, a componente *Activiti Engine* suporta uma *API* completa e estruturada. O passo seguinte do nosso trabalho foi analisar a fundo a extensibilidade de ambas as componentes *Activiti Designer* e *Activiti Engine*. Na próxima secção será descrito em maior detalhe como estão construídas essas componentes e que mecanismos são disponibilizados para garantir a extensibilidade.

3.3 Extensibilidade das componentes *Activiti*

O ponto fundamental desta secção é encontrar pontos possíveis de extensão nas três diferentes dimensões que estão relacionadas com a execução de um *workflow*. Estas dimensões são explicadas pela ordem apresentada na Figura 3.14:

3.3.1 - Explicação de como a componente *Activiti Designer* está construída e como estendê-la.

3.3.2 - Os mecanismos de extensão possíveis na interacção entre componentes.

3.3.3 - Explicação de como a componente *Activiti Engine* esta construída e como estendê-la.

3.3.1 *Activiti Designer*

Para facilitar a construção de processos de negócio, a equipa de desenvolvimento desta ferramenta focou-se em disponibilizar um único ambiente garantido a integração entre o ambiente de desenvolvimento e o de execução. Para além desse facto, sendo a maioria dos desenhadores de processos responsáveis de desenvolvimento, a solução passou pela construção desta componente como um *plugin* para o *IDE Eclipse*. Por conseguinte, a ferramenta está estruturada nos seguintes três projectos:

1. Um projecto *Plugin Development Environment* [iDEP] que gera um *plugin* totalmente integrável em versões que suportem este tipo de extensões. Depois de instalado, no *Eclipse*, o *plugin* fornece a possibilidade de criar um projecto *Activiti* onde são fornecidas todas as funcionalidades mencionadas na secção 3.2.1.

2. Um projecto em *Eclipse Graphiti* [Gra], que suporta a criação de ferramentas visuais de construção de diagramas seguindo uma determinada notação. No caso da componente *Activiti Designer* permite a construção de *workflows BPMN 2.0*, a adição de uma palete de elementos que podem ser adicionados a um *workflow* (apresentada na secção 3.2.1 na Figura 3.11) , e a construção de menus de contexto para as tarefas (Figura 3.12).

3. Um projecto em *EMF - Eclipse Modeling Framework Project* [Pro] necessário para a gestão do modelo de objectos que representam os elementos envolvidos na construção de diagramas, neste caso, elementos *BPMN 2.0*. Suporta também a construção de secções na área de propriedades das tarefa apresentada na secção 3.2.1.

O código fonte destes projectos é partilhado, o que significa que seguindo as directivas de cada projecto é possível estender a componente *Activiti Designer*. Por exemplo para adicionar uma nova tarefa é necessário: criar a nova tarefa no modelo de objectos *EMF*; criar (ou reutilizar) uma área de propriedades para a tarefa; adicioná-la à palete de elementos de um *workflow* no projecto *Graphiti*; adicionar as opções para o menu de contexto da tarefa no projecto *Graphiti*.

Concluindo, utilizando as tecnologias mencionadas é possível enriquecer a interface adicionando novos elementos e funcionalidades (e.g. novas tarefas).

3.3.2 Ligação entre *Activiti Designer* e *Activiti Engine*

Com o ambiente instalado e o projecto *Activiti* criado, a próxima fase foi analisar, que outras formas de interacção existem entre a componente *Activiti Designer* e *Activiti Engine*, para além do ficheiro *XML* representante do processo.

Um caso é o ficheiro *XML Activiti.cfg.xml* de configurações referido na secção 3.2.1. Este ficheiro representa as configurações a serem aplicadas antes da execução de um *workflow*, bem como que serviços no motor de execução devem ser desactivados/activados. Essas configurações são por exemplo: configuração de um servidor de *email* para as tarefas *Email task*; configuração de um ficheiro *log* de alterações durante a execução; configurações para ligações a base de dados (*PostgreSQL*, *MySql*); activação do executor de tarefas assincronas, etc.

Outro exemplo é a disponibilização, durante a fase de desenho, de uma área de propriedades para cada tarefa (mencionada em 3.2.1). Nesta área parametrizam-se os dados possíveis de serem aplicados durante uma execução. Como exemplo ilustrativo utilizamos a tarefa de maior interesse para o nosso protótipo, a tarefa *service task*. Como se

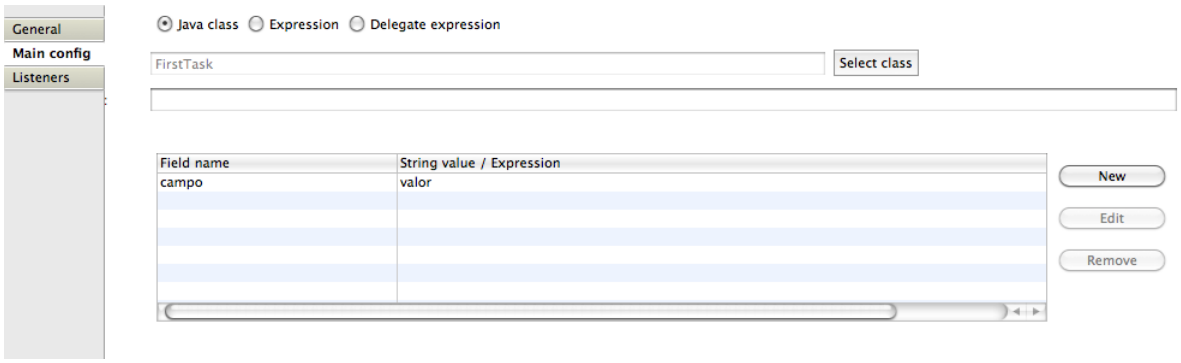


Figura 3.15: Propriedades de uma *service task*

verifica na imagem 3.15, esta componente oferece um formulário de parametrizações dos campos necessárias na execução de uma *service task*. Neste caso os campos parametrizados foram: a classe a ser interpretada (*FirstTask.java*) e uma lista de variáveis a inserir no escopo desta tarefa (variável "campo" com valor "valor"). A classe a interpretar tem de implementar uma interface específica do ambiente de execução de forma a ser instanciada na invocação da tarefa.

Todas estas parametrizações são incluídas no ficheiro *XML* gerado para que o motor de execução as possa interpretar. Para este efeito são utilizados os elementos extensíveis do *BPMN 2.0* (*extensionElements*, secção 2.3.3) e os elementos da notação *Activiti* para representação de regras de negócio. O pedaço de código *XML* seguinte ajuda a ilustrar essa conjugação:

Listing 3.1: BPMN 2.0 gerado com notação do Activiti

```
<serviceTask id="servicetask1" name="Service Task" activiti:class="FirstTask.java">
      <extensionElements>
        <activiti:field name="campo">
          <activiti:string>valor</activiti:string>
        </activiti:field>
      </extensionElements>
    </serviceTask>
```

Como citado na secção 3.2.1, a execução é realizada sob a forma de testes da *framework*

jUnit, que oferece uma consola de *output* de dados da execução de *workflows*, mas também uma linha de comandos (*input*). Estando num ambiente integrado entre ambas as componentes, essa consola é assim partilhada, i.e., no ambiente de execução é possível obter comandos escritos no ambiente de desenvolvimento o que pode garantir uma forma de interacção entre eles.

Outra forma de interacção com o motor de execução é através da utilização da *API Activiti*. Através dos serviços mencionados na secção 3.2.2, é possível obter e submeter informação ao ambiente de execução. O serviço *RuntimeService* é de especial interesse para o nosso trabalho, dado que oferece a possibilidade de utilizarmos uma estrutura de dados global partilhada para a componente *Activiti Designer* e *Activiti Engine*. É utilizada a lógica de comunicação *whiteboard* onde com chave/valor se insere e obtém valores durante a execução de um *workflow*.

Concluindo, verifica-se que existem quatro possibilidades de extensão da interacção entre as componentes *Activiti*, *Activiti Designer* e *Activiti Engine*. Com o ficheiro *activiti.cfg.xml* é possível configurar a execução; através da área de parametrizações é possível fornecer ao motor de execução informação para o escopo de uma tarefa; uma consola partilhada entre as componentes *Activiti*; e através da *API Activiti*;

3.3.3 *Activiti Engine*

Esta secção descreve como é realizada a execução de um *workflow* pelo *Activiti Engine*, com o objectivo de identificar os pontos onde este pode ser estendido. A execução de um *workflow* compreende uma fase de avaliação das suas definições e a fase de execução propriamente dita. A componente *Activiti Engine* é o motor de execução da ferramenta *Activi*. Apresenta-se como um projecto *Java* que alberga as classes necessárias ao suporte dos serviços apresentados na secção 3.2.2.

No início da sua execução, recebe como entrada os ficheiros *XML* de configurações e de definição do *workflow BPMN 2.0* (descritos nessa mesma secção). Estes são validados e guardados numa base de dados [3].

Os elementos *BPMN 2.0* são interpretados e associados a uma estrutura de dados gerida pelo objecto representante do escopo do processo (*ScopeImpl* ou *ProcessDefinitionEntity*), e cada tarefa está representada por uma classe que contem a informação do seu escopo. Primeiro é realizada a importação das tarefas e depois as ligações entre essas (linhas de sequência), criando assim a definição do processo.

Depois de adicionar a definição e criada a conexão à base de dados, é instanciada a execução do *workflow* através do serviço *RuntimeService*.

Ao mesmo tempo é instanciada uma variável local(*ThreadLocal*) ao *thread* que executa o *workflow* e onde contêm os dados privados representantes dos valores de contexto de uma execução (objectos representantes das tarefas, variáveis associadas, comandos, etc).

[3]Por omissão é utilizada a biblioteca *open-source H2 Database Engine e a* API JDBC (Java Database Connectivity) para criar conexões.

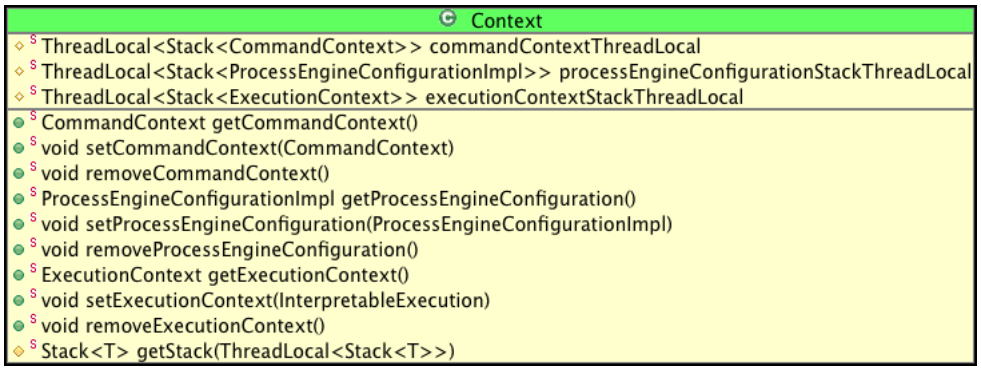


Figura 3.16: Classe *Context*

Como ilustrado na Figura 3.16, a classe *Context* representa o contexto global de uma execução, isto é, onde é gerido o acesso aos objectos instanciados por quatro classes fulcrais ao fluxo de execução. Essas classes têm o seguinte objectivo (em anexo encontra-se o diagrama *UML*, Figura A.1):

Execution Entity representa uma execução e suas variáveis correntes. Determina também qual o próximo passo a tomar no fluxo de execução, invocando um método no *Command Context* para executar a próxima operação.

Command Context Esta classe interpreta o comando e inicia a operação respectiva, i.e., instancia a *Atomic Operation* respectiva.

Atomic Operation representa as operações que ocorrem durante o fluxo de execução, que podem ser: "iniciar uma tarefa", "eliminar o contexto de uma tarefa", "execução de um comportamento", "inicio de um processo", etc. Em algumas operações existe a necessidade de invocar outras operações, o que implica interagir com o *Command Context*.

Behavior representa a classe que depois de instanciada é responsável por executar o comportamento associado a uma tarefa.

Para explicar da melhor forma como é realizado o fluxo de execução de um *workflow*, com suporte a essas quatro classes, decidiu-se dividir a explicação em dois níveis. Numa descrição mais geral, ilustrado na Figura 3.17, são enumeradas as acções e passos principais subjacentes ao fluxo de execução do *Activiti Engine*. Numa descrição mais detalhada, são descritos alguns dos passos com mais detalhe, tal como ilustrado na Figura 3.18.

Descrição geral A figura 3.17 ilustra o fluxo de execução entre os objectos instanciados das 4 classes mencionadas.

Em primeiro lugar são criadas instâncias das classe *Command Context* e *Execution Entity* (a) da figura). No momento de validação do ficheiro *XML* do *workflow BPMN 2.0* é

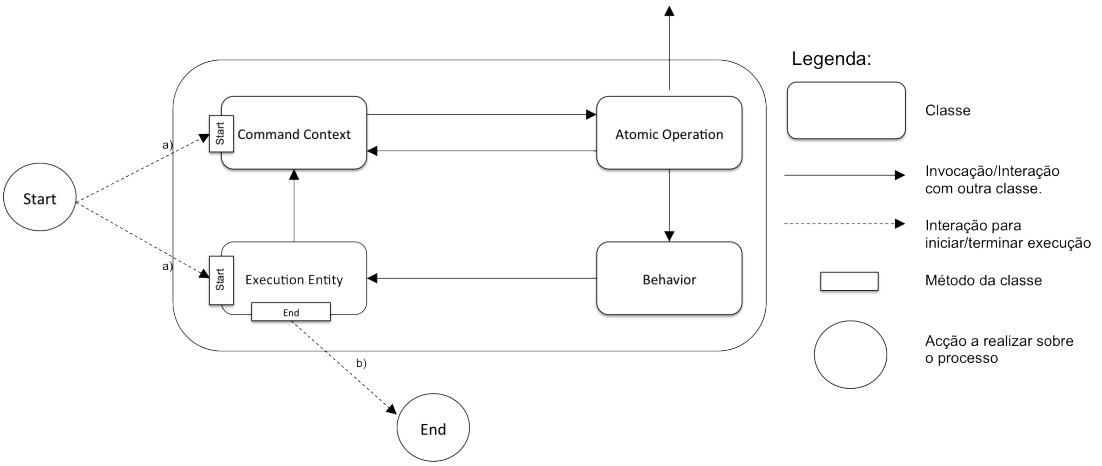


Figura 3.17: Esquema do fluxo de execução (Descrição geral)

referenciada qual a primeira tarefa a ser executada, portanto no inicio do fluxo a *Execution Entity* associa essa como a tarefa corrente e invoca um método no *Command Context* com essa informação. O *Command Context*, como mencionado, invoca as várias *Atomic Operations* necessárias para a execução de uma tarefa. Essas operações estão identificadas por um identificador único que representa a classe a ser executada. Normalmente a ordem de comandos para a execução de uma tarefa é:

- "termina a tarefa anterior";
- "limpa as variáveis de escopo da tarefa anterior";
- "cria escopo para tarefa corrente"
- "executar comportamento da tarefa corrente"
- "terminar a execução do comportamento"

Após a execução do comando ”terminar a execução do comportamento”, na variável local *ThreadLocal* é obtida a próxima tarefa a ser executada. Assim é retomado o ciclo de execução no objecto *Execution Entity*. Um processo chega ao fim quando é executado o comportamento de uma *end task* ou quando ocorre uma excepção durante a execução do motor (b)).

Descrição detalhada Para explicar de uma forma mais detalhada o funcionamento do motor de execução, de seguida é explicado o fluxo de execução num *workflow*, usando a Figura 3.18 como suporte (em anexo segue uma explicação dos métodos envolvidos A):

1. Neste passo a *Execution Entity* analisa qual o próximo passo do *workflow* a ser tomado. Caso seja só uma tarefa é executado o método *take*. Caso seja a execução de múltiplas tarefas é executado o método *All* que invoca o método *take* para cada uma dessas.

2. Nesta passo é verificado, no método *Perform Operation* se a tarefa a executar é assíncrona ou não. Caso seja, é invocado o método *Send Job* que a envia para a *pool* de *threads*

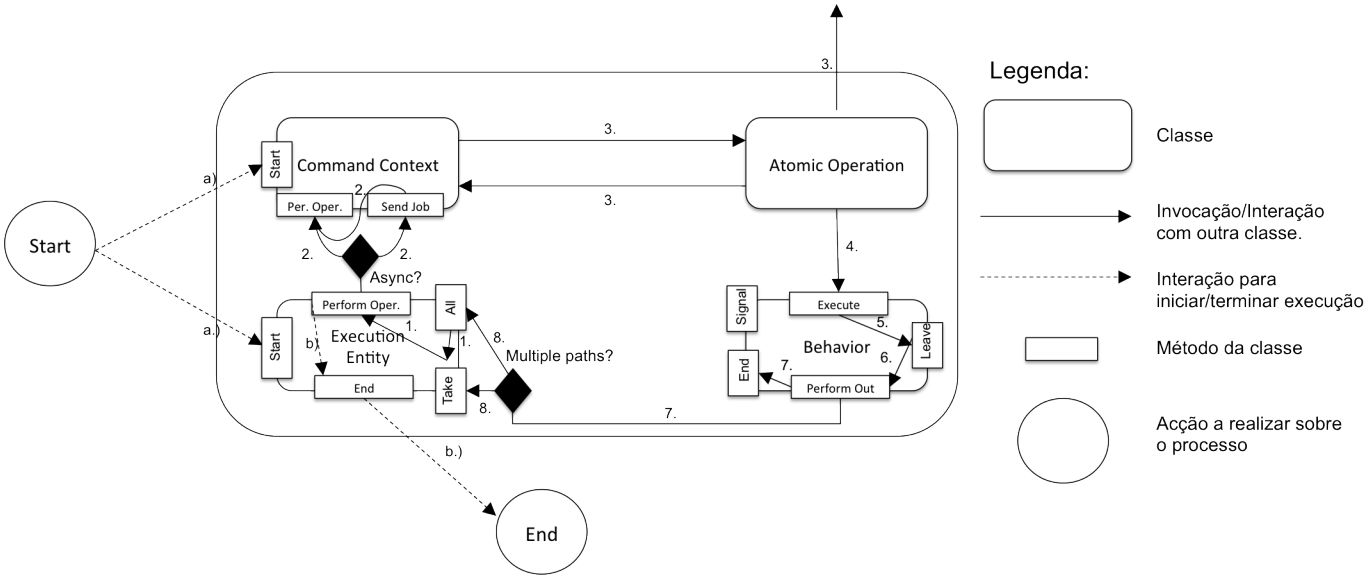


Figura 3.18: Esquema do fluxo de execução (Descrição detalhada)

gerida pelo serviço *ManagementService*. O comando "executa a seguinte tarefa"é enviado por parâmetro ao método *Perform Operation* e o serviço *ManagementService* executará a tarefa concorrentemente. Caso não seja assíncrono é enviado o comando "executa a tarefa corrente"directamente para o método *Perform Operation* do *Command Context*.

3. Esta fase é bi-direccional visto que a execução de uma operação pode requerer a execução de outras, por exemplo, antes da execução do comportamento associado a uma tarefa é necessário as seguintes operações: "termina a tarefa anterior", "limpar as variáveis de escopo da tarefa anterior", "criar escopo para tarefa corrente", "executar comportamento da tarefa corrente".

4. Ao interpretar a operação ”executar comportamento da tarefa corrente” é obtido o comportamento da tarefa corrente, ou seja, a classe que representa o comportamento (*Behavior*) para executar o método *Execute*.

5. Nesta fase é executado o comportamento associado à tarefa; se for uma *service task* é executada a classe *java* associada; se for uma *Manual task* é esperado um *input* de um utilizador para a continuação da execução. Se for um evento de *EndEvent* entidade de execução executará a tarefa de terminar o processo, etc.

6. Depois de executado o comportamento, o método *leave* é invocado para realizar acções necessárias após a execução da tarefa, como por exemplo, obter quais são as próximas tarefas a executar.

7. Nesta interacção são verificadas as tarefas a executar e termina o comportamento invocando o método *End*, que indica que a execução da tarefa corrente deixou de estar activa.

8. Quando existem múltiplos caminhos, todas as tarefas são adicionadas a uma estrutura de dados e enviadas por parâmetro ao método *takeAll*. Nesse, a estrutura é iterada

e é executado o método *take* para cada uma das tarefas retomando o ciclo do fluxo de execução.

Todas as classes mencionadas neste modelo de interacção estão suportadas no projecto *Java* deste motor de execução. O que implica que a extensibilidade desta ferramenta passa por estender este modelo de fluxo de execução. A forma como está estruturado possibilita essa extensão, i.e., cada entidade é independente das restantes porque estão ”ligadas” por invocação de métodos.

No capítulo seguinte será apresentada a solução proposta para a criação do protótipo.

4

Implementação sobre a ferramenta *Activiti*

A solução proposta para o desenvolvimento desta dissertação baseia-se na utilização do modelo teórico e nos mecanismos de extensibilidade apresentados anteriormente.

Os componentes da plataforma *Activiti* escolhidos encaixam-se de forma natural no modelo teórico apresentado em [ACKM04]. Ao seguir as directivas apresentadas nesse modelo, é criada uma plataforma que suporta a composição de estruturada de *workflows*, como ilustrado na figura 4.1.

Em seguida, foi analisado como garantir os outros objectivos propostos, i.e.:

- Adição de funcionalidades, de agregação estruturada de tarefas, com base em padrões estruturais e de comportamento.

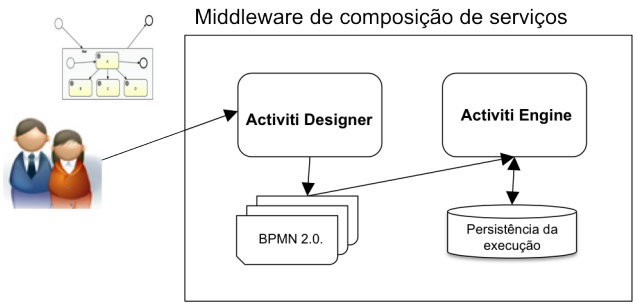


Figura 4.1: Protótipo para composição de serviços

- Mecanismos de reconfiguração estática e dinâmica, sobre esses padrões, nas dimensões de estrutura e comportamento, i.e. reconfigurações em termos da dependências entre tarefas, quer estruturais, quer de fluxo de dados e controlo.

A disponibilização de padrões no nosso protótipo, foi realizada no contexto de duas fases da construção de *workflows*:

a) o ambiente de desenvolvimento da ferramenta *Activiti* foi estendido de forma a conter os padrões estruturais estrela, *pipeline* e anel, bem como um *template* para composição de tarefas, designado *composite*;

b) Entre as tarefas, foram adicionados novos modelos de interacção do tipo "fluxo de dados" – Produtor/Consumidor, Publicador/Subscritor e *Streaming*.

4.1 Padrões estruturais

Para permitir novas composições estruturadas de elementos, num *workflow*, foram escolhidos para a implementação algumas topologias típicas, tais como: o *pipeline*, estrela, e anel. Todos eles foram adicionados à ferramenta Activiti sob a forma de esqueletos ou *templates* de padrões (*Pattern Templates*), tal como especificado em [GRC08]. Os *templates* de padrões de estrutura, (*TPEs*) capturam relações estruturais típicas que é conveniente re-utilizar, devendo ser parametrizados/instanciados com outros elementos do sistema.

Os *templates* de padrões de estrutura, são também considerados como entidades de primeiro nível, tal como outras entidades do sistema Activiti (e.g. *task*, *sub-process*, etc). Assim, é possível aplicar-lhes as funcionalidades básicas disponíveis no Activiti para a manipulação destas entidades. Por exemplo, arrastar o *icon* de um TPE a partir da palete de elementos, (fazer "*drag and drop*") para a área de construção de *workflows*, bem como, associá-los a outros elementos formando um *workflow*.

No contexto do *Activiti Designer*, a implementação dos *templates* de padrões é suportada por sub-processos BPMN 2.0 e, os elementos que constituem o padrão, podem ser elementos BPMN 2.0 ou outros *templates* de padrões. De notar que, o uso de um subprocesso, resulta da necessidade de tornar a execução de um padrão independente da execução principal do *workflow*, no qual esse padrão esteja inserido, tal como, explicado na descrição da implementação dos padrões de comportamento.

Inicialmente, os elementos que constituem um qualquer *template* de padrão são representados por *empty tasks*, simbolizando "*tarefas genéricas*". Indicando assim, que o *template* deve ser parametrizado/instanciado com os elementos referidos. i.e. elementos BPMN 2.0 ou outros *templates* de padrões. Como esta ferramenta não disponibiliza uma tarefa abstracta, sem qualquer comportamento associado, foi criada para esse efeito a *empty task*, a qual pode ser trocada, por exemplo com uma *task* BPMN 2.0, a quando da instanciação do *template* de padrão. Basta para isso, arrastar a tarefa executável, para cima da representação gráfica da *empty task*.

A utilização de padrões, para além de simplificar o processo de construção, permite assim construir *templates* de *workflows* para futuras reutilizações. Estes *templates* representam *workflows* abstractos, os quais podem ser transformados em *workflows* executáveis pela instanciação das *empty tasks* tal como descrito acima.

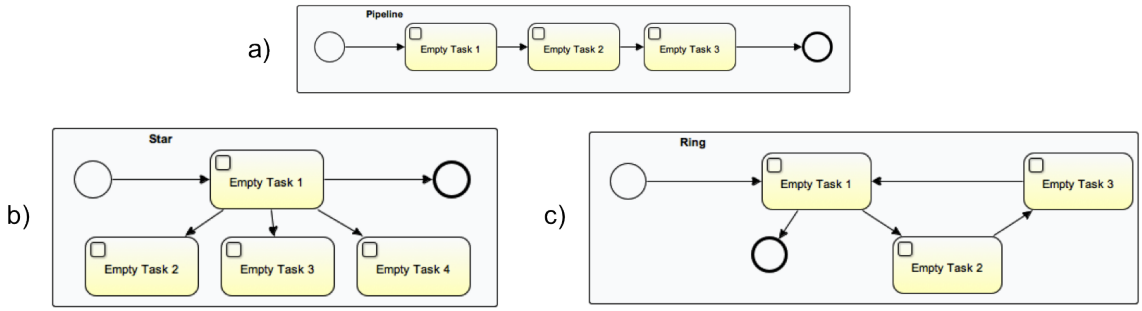


Figura 4.2: Padrões estruturais (topologias)

A Figura 4.2 mostra três *templates*, de padrões de estrutura, correspondentes às três topologias. Obedecendo a estruturação de cada *template* à definição do padrão estrutural correspondente. Assim,

- A topologia *pipeline*, define um conjunto de etapas,tal que, a primeira está estruturalmente ligada a segunda, a segunda está ligada à primeira e terceira, e a terceira e última, só está ligado à penúltima. No caso do *pipeline* na Figura 4.2 a), e no contexto da notação BPMN 2.0, está já explícito que a *Empty Task 2* dependerá da *Empty Task 1*, em tempo de execução, e que a *Empty Task 3* também dependerá da *Empty Task 1*. A especificação desta dependência, em concreto, será feita com a combinação do padrão de estrutura, com um padrão de comportamento (ver secção 4.2).

- A topologia estrela, ilustrada por sua vez na Figura 4.2 b), representa a associação estrutural ponto a ponto, entre um elemento central (núcleo da estrela), e os restantes elementos da estrela (satélites). No caso do exemplo na Figura, as dependências estão também já explícitas, e processam-se do núcleo para os satélites.

- A topologia anel, representada na Figura 4.2 c), tem uma definição semelhante ao *pipeline*, excepto que, não existe noção de primeira e última etapa, (cada uma está ligada unicamente à etapa que a antecede e à seguinte).

Em qualquer topologia é possível existir tarefas representadas por topologias. Assim é possível construir uma hierarquia de topologias. Como ilustrado na Figura 4.3, em que uma das etapas da topologia *pipeline* é uma topologia em estrela.

O elemento *composite*, por sua vez, representa um sub-processo contendo, como seus elementos, um sub-processo e duas tarefas. Tal permite uma definição recursiva incremental, dado que, um sub-processo pode conter outros sub-processos ou tarefas. O *template composite* permite que, um grupo de elementos possa ser tratado como um todo, e serve de base à implementação dos *templates* de padrões.

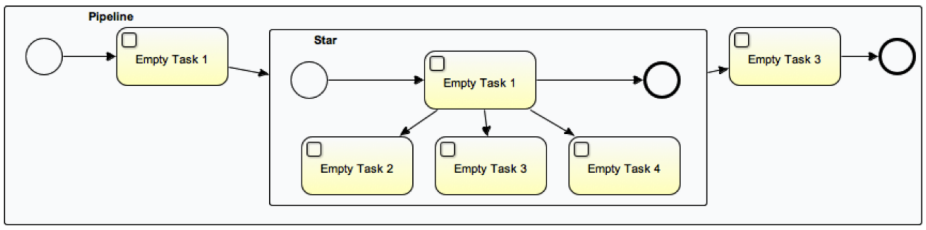


Figura 4.3: Hierarquia de topoligas

Segue-se a explicação, de como estes novos elementos foram implementados.

4.1.1 Processo de extensão

O processo de extensão, referente à criação de novos elementos, centrou-se na extensão da componente *Activiti Designer*. O maior desafio, foi analisar os seus mecanismos de extensibilidade possíveis, por forma a incluir os novos elementos na componente visual. Assim, a solução, passou por adicionar os novos elementos à estrutura EMF, que suporta esta componente. O primeiro passo foi, adicionar todos os novos elementos a esse modelo, sendo definida uma *EClass* para cada padrão, bem como para a *empty task*.

Um padrão particular, sendo definido através de um sub-processo *BPMN 2.0*, herda todas as classes, pelo que, existe uma estrutura de dados onde estarão definidas todas as tarefas incluídas nesse sub-processo. No construtor do padrão, é incluído o *workflow* que o representa, e que é então adicionado a essa estrutura de dados, para ser usado na representação gráfica do padrão estrutural. Esta estrutura, é percorrida no momento de construção gráfica do *workflow*, de forma a criar os elementos. Nesse momento, a representação *BPMN* também é gerada. Com esta solução tentámos usar ao máximo a implementação *BPMN* disponibilizada por esta versão do *Activiti*.

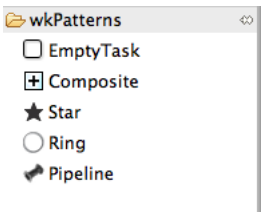


Figura 4.4: Novos elementos na palete

Os padrões foram também adicionados à palete suportada pela *framework Graphiti*, a qual ficou com o aspecto ilustrado na Figura 4.4. Assim, para serem utilizados na definição de outros *workflows*, basta arrastá-los para a área de construção.

Para criar a *empty task*, e sabendo que só é utilizada para *workflows* abstractos, a solução limitou-se à sua representação visual usando a notação *BPMN DI*.

4.2 Padrões de comportamento

Nas ferramentas que suportam a notação *BPMN 2.0*, tal como a ferramenta *Activiti*, o modelo básico de fluxo de execução existente, é representado pela direcção das linhas de sequência. Isto é, estando uma tarefa A liga a uma tarefa B, implica que B só executa depois de A a executar. Uma sequência de dependências similares define por sua vez o fluxo de controlo num *workflow*.

No entanto, com o aumento da complexidade das interacções entre sistemas de informação, sentiu-se a necessidade de enriquecer o número de modelos de interacção existentes. Para esse efeito, foram analisados dentro dos padrões de desenho, explicados na secção 2.4.2, os padrões de comportamento, que caracterizam as diferentes formas de cada classe ou objectos a interagirem e distribuírem responsabilidades entre si. No âmbito desta tese, foram escolhidos dentro desses comportamentos, aqueles que são mais reconhecidos em sistemas distribuídos, ou seja, os modelos de interacção. Publicador/-Subscritor, Produtor/Consumidor e *Streaming*.

No contexto desta ferramenta, decidiu-se aplicar estes modelos de interacção a estruturas complexas de elementos, neste caso, as topologias representadas por sub-processos na notação *BPMN 2.0*. A razão para esta decisão deve-se ao facto, das topologias serem representada por estruturas bem definidas, que simplifica o processo de distinção de papéis entre as tarefas.

Essa distinção é realizada pelas linhas de sequência existentes nessas topologias. A direcção da linha distingue quem produz e quem consome os dados. Isto é, num padrão de comportamento, é a linha de sequência que representa a dependência de dados e de controlo que existente entre os intervenientes. Por exemplo, se aplicar um comportamento produtor/consumidor a uma topologia em estrela, o nó do núcleo representa o produtor e restantes consumidores, sendo que esses, só executam quando o produtor produzir valores. No caso de uma estrutura em *pipeline*, com o mesmo comportamento, existe o caso de nós com duplo papel. Estes nós intermédios, vão ter o papel de consumidor para o nó anterior e, de produtor para o nó seguinte. Na próxima secção será explicado como estes nós são distinguidos para cada comportamento.

Neste contexto, de seguida serão explicados que mecanismos e alterações foram necessárias nas três dimensões de extensibilidade, para garantir a aplicabilidade destes padrões.

4.2.1 Processo de extensão

Para o âmbito desta ferramenta, um comportamento existe no contexto de um padrão estrutural, ou seja, os padrões de comportamento são aplicados aos elementos de padrões estruturais. Assim, de forma a "combinar" um comportamento numa estrutura, a aplicabilidade de comportamentos, passa pela utilização das áreas de parametrizações existentes nos elemento *BPMN 2.0*.

Essas parametrizações são geradas para o ficheiro *XML* de integração (secção 3.2) das componentes *Activiti Designer* e *Activiti Engine*. Os elementos extensíveis *BPMN 2.0* e, a notação *Activiti* são usados de forma a enviar instruções, ao motor de execução para executar o comportamento escolhido. A notação *Activiti*, é utilizada para representar as instruções, para o qual são usados elementos *activiti* com o atributo *name*, para diferenciar a instrução. Foi escolhida esta solução, para não modificar e, utilizar ao máximo as notações envolvidas. Ao aplicar um comportamento, a sua execução, tem de garantir a dependência de dados e de controlo. Para além disso, e pela própria definição dos comportamentos, ao aplicá-los, é exigido que a sua execução, possa estar em constante actividade durante um largo período tempo.

Nos sistemas computacionais modernos, cada vez mais, é exigível que exista uma execução contínua. Um exemplo, são os serviços que disponibilização informações em zonas problemáticas de cheias, em que, a obtenção da informação do estado dos níveis da água, é fulcral para essas zonas. No âmbito deste trabalho, esta conclusão é importante para a execução de um sistema composto por *web services*. Como esta ferramenta, por omissão tem o fluxo de execução em sequência, significa que as tarefas são invocadas uma única vez e, rapidamente o processo termina. Logo, estando os comportamentos associados a padrões de estrutura, procurou-se um forma simples de garantir esse requisito, pela própria definição do comportamento. Não obstante, que é importante que a consistência de execução, de um *workflow BPMN 2.0* seja garantida durante esse período de tempo.

Assim, para suportar a execução contínua de um comportamento, é gerado um *workflow* de suporte. Com a expressividade de um *workflow*, é possível construir estruturas que conseguem suportar a execução de um modelo de interacção. Através de vários padrões de controlo de *workflows*, como *loops*, *sequências*, *sincronismo*, etc. É possível capturar as dependências do fluxo de dados necessárias, dos modelos de interacção escolhidos.

Para além desse facto, a não necessidade de alterar qualquer notação envolvida, e a garantia da consistência de execução, foram outras razões para escolher esta solução. Nas próximas secções, descreve-se os *workflows* que foram criados para os comportamentos escolhidos.

No contexto do motor de execução, todos os elementos têm de ter um identificador único, no caso dos comportamentos, decidimos representar esse identificador com junção de dois termos: o identificador do sub-processo com comportamento, mais a cadeia de caracteres "publisher_behavior", para o comportamento publicador/subscritor e "producer_behavior" para o caso de produtor/consumidor e *streaming*. Este identificador é associado a tarefa publicadora/produtora de forma a quando essa for executada, a instância de execução execute o comportamento associado (posteriormente será descrito como).

Para facilitar o nosso processo de extensão, e criar dinamismos na execução de um *workflow*, foram criadas duas tarefas. Essas tarefas são denominadas tarefas *Dispatcher* e *Constructor*, que na tabela 4.1 são apresentadas.

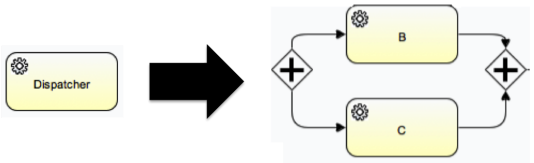
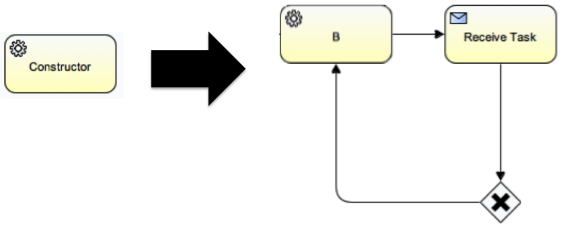
Nome tarefa	**Descrição**	Exemplo	Descrição exemplo
Dispatcher	Tarefa responsável por, em tempo de execução, construir um *workflow* a partir de um grupo de tarefas. Pode ser aplicada várias vezes ao longo da execução.		Com um conjunto de tarefas, que inclui A e B, é construído o *workflow* apresentado, que executa ambas as tarefas em paralelo.
Constructor	Tarefa responsável por, em tempo de execução, construir estruturas comuns em *workflows*. A sua execução transforma-a numa estrutura, o que implica que só pode ser usada uma única vez.		A figura representa uma estrutura típica em *workflows*, *loops*. Com o *gateway* exclusivo o ciclo é mantido, até a condição associada for falsa.

Tabela 4.1: Tarefa *Dispatcher* e *Constructor*

A geração *BPMN 2.0* destas tarefas é semelhante à geração da tarefa *service task* tendo contudo instruções que diferenciar o seu papel. Por exemplo, para distinguir o papel da tarefa *Dispatcher*, o código *BPMN 2.0* gerado contem o identificador do comportamento activo (listagem 4.1). Assim no momento em que a tarefa é executada, sabe-se qual o comportamento activo, de forma a obter o grupo de tarefas a serem executadas. Na explicação de cada comportamento, na secção *Interacção entre componentes*, é descrito em maior detalhe como é realizada a transformação desta tarefa para executar o grupo obtido.

Listing 4.1: Código BPMN 2.0 para tarefa *Dispatcher*

```
...
<extensionElements>
   <activiti:field name="activitirunBehavior">
     <activiti:string>subpipeline1publisher_behavior</activiti:string>
   </activiti:field>
 </extensionElements>
...
```

Para o caso da tarefa *Constructor* para a sua distinção incide em elementos que representam a estrutura a construir (*activititypeConstructor*) e o identificador da tarefa a executar nessa estrutura (*activitirunConstruct*) (listagem 4.2).

Listing 4.2: Código BPMN 2.0 para tarefa *Constructor*

```
...
<extensionElements>
   <activiti:field name="activitirunBehavior">
     <activiti:string>substar1producer_behavior</activiti:string>
   </activiti:field>
   <activiti:field name="activititypeConstruct">
```

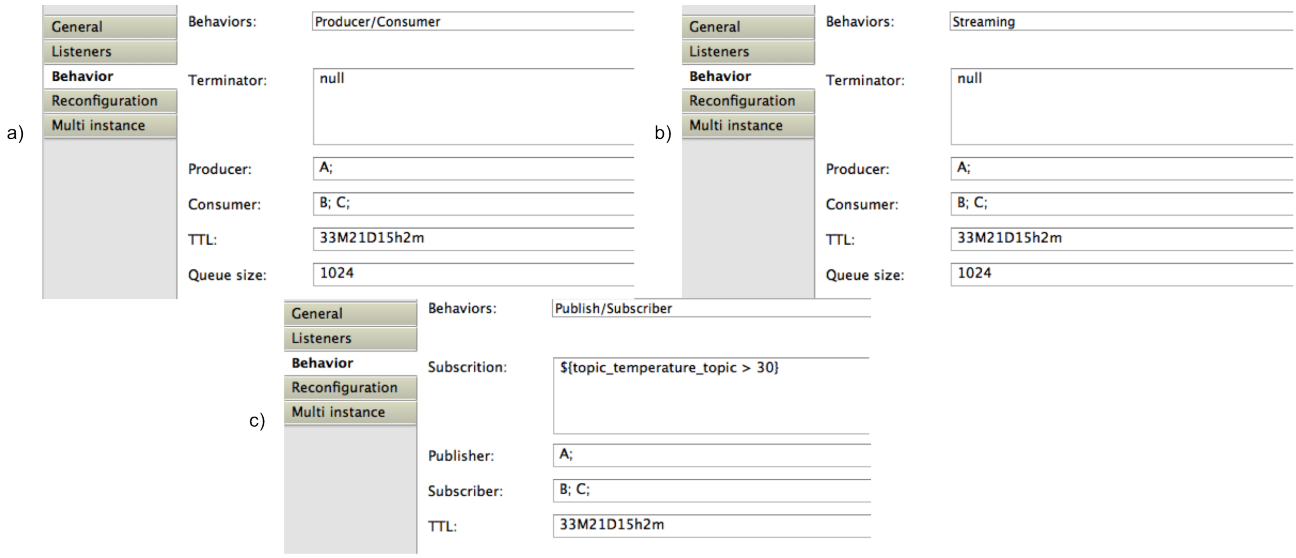


Figura 4.5: Parametrizações comportamentos

```
7       <activiti:string>loop</activiti:string>
8     </activiti:field>
9     <activiti:field name="activitirunConstruct">
10      <activiti:string>star1servicetask1</activiti:string>
11    </activiti:field>
12  </extensionElements>
13 ...
```

Com estas novas tarefas e, os mecanismos de extensibilidade das três dimensões, de seguida são explicadas as alterações realizadas para a execução destes comportamentos.

4.2.2 Extensão do *Activiti Designer*

Para esta dimensão, foi necessário criar uma nova secção na área de parametrizações dos elementos representando padrões estruturais, como se pode verificar na Figura 4.5. Foi criada uma secção *Behavior* onde é possível seleccionar qual o padrão de comportamento, bem como instanciar campos necessários para a sua execução. Para este efeito, foi criado um formulário que altera a sua estruturação dependendo do valor do campo *Behaviors*, i.e., qual o padrão de comportamento seleccionado.

De forma a refinar a execução de um padrão de comportamento, é oferecida a possibilidade de parametrizar campos específicos. Por exemplo, nos casos a) e b) da Figura 4.5, os comportamentos produtor/consumidor e *streaming*, é possível configurar o tamanho da fila onde os valores produzidos serão guardados (*Queue size*). Outro exemplo, ainda, é a parametrização de como o comportamento termina. No que se refere à terminação de execução, existem três possibilidades:

- Com um valor por omissão (*Terminator*), i.e., quando é produzido esse valor o comportamento termina;

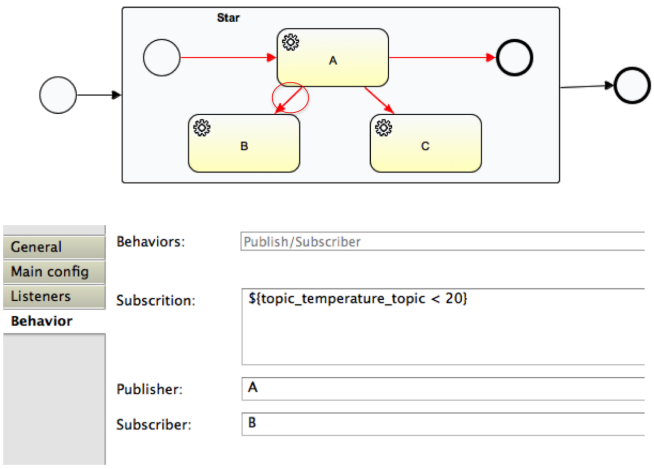


Figura 4.6: Parametrização comportamento linha de sequência

- Adicionando um tempo de vida (*TTL*) para a execução de um padrão.
- Através da linha de comandos oferecida pela *framework JUnit*: Com o identificador do comportamento, o comando para o terminar é: "terminate "+identificador do comportamento.

Para o caso de Publicador/subscritor (Figura 4.5, c)) não existe *terminator*, mas existe o campo *Subscrition* que serve para aplicar a condição de subscrição dos elementos do padrão. Esta subscrição tem de seguir as regras de construção de condições da ferramenta *Activiti*, isto para serem suportadas na ferramenta e para na exportação seguirem o *XML namespace Activiti*. A definição de uma subscrição é aplicada a todas as linhas de fluxo existentes no sub-processo. Contudo, como ilustrado na Figura 4.5, permite-se também a associação de condições de subscrição a linhas de fluxo específicas. Assim, pode-se alterar a subscrição de elementos em particular sobrepondo-se a condição parametrizada nas propriedades do sub-processo. Como na notação *BPMN 2.0* as linhas de sequência representam fluxo de controlo do *workflow*, decidimos apresentar essas linhas com a cor vermelha quando estas topologias têm comportamento. A razão para esta escolha é diferenciar que não existe explicitamente um fluxo de controlo, mas sim que representam uma diferenciação de papéis no fluxo de dados existente no comportamento escolhido.

De seguida apresenta-se um resumo dos diferentes campos que são possíveis de parametrizar na nova secção *Behavior*:

Behavior Escolha do comportamento.

Terminator Valor que faz o comportamento terminar. No caso da imagem a) e b) o comportamento termina com valor *null*.

Subscrition Subscrição a aplicar a todos os nós subscritores (pode ser alterada nas linhas de execução). Subscrições podem ser uma disjunção ou uma conjunção de condições compostas pelos nomes de tópicos.

Publisher/Producer Mostra o(s) nome(s) do(s) nó(s) publicador(es)/produtor(es)

Subscriber/Consumer Mostra o(s) nome(s) do(s) nó(s) subscritor(es)/consumidor(es)

TTL *Time to live* do comportamento. A parametrização deve seguir uma especificação, neste caso, dd/MM/yy.hh:mm:ss ou 3d2h4m3s4ml.

Queue size Parametrização da fila de valores produzidos.

4.2.3 Ligação entre *Activiti Designer* e *Activiti Engine*

A descrição que se segue está organizada segundo as particularidades de cada um dos comportamentos, estando os passos descritos numa ordem cronológica necessária de parametrizações. Em primeiro lugar, que mecanismos são utilizados para gerar as parametrizações da nova secção; em segundo o resto das parametrizações a efectuar, tais como, parametrização dos papéis das tarefas e das fontes de dados; e por fim a explicação do *workflows* de suporte à execução de um comportamento para garantir a consistência de execução.

No contexto desta ferramenta, produtores e publicadores são representados por uma *service task*, visto ser esta tarefa a que permite a obtenção de dados a partir de diferentes fontes, como por exemplo, *rss*, *web services*, etc. No caso dos consumidores, para se obter os valores produzidos, de igual modo, tem-se de representar a tarefa como uma *service task*, já que os valores são obtidos programaticamente como descrito mais a frente.

De seguida, agrupado por comportamentos, estão descritos todos os passos necessários de parametrizações.

4.2.3.1 Publicador/Subscritor

Como descrito, na secção introdutória da solução dos padrões de comportamento, a aplicação de um comportamento a uma estrutura, incide sobre todas as linhas de fluxo da última. Recorrendo ao exemplo da Figura 4.6, ao aplicar o comportamento publicador/-subscritor, as linhas de sequência geradas, com os elementos extensíveis, ficam com a seguinte informação:

Listing 4.3: Comportamento numa linha de sequência Publicador/Subscritor

```
...
 <extensionElements>
      <activiti:field name="activitibehavior">
        <activiti:string>Publish/Subscriber</activiti:string>
      </activiti:field>
      <activiti:field name="activitibehaviorCondition">

        <activiti:expression><![CDATA[${topic_temperature_topic == 20}]]></activiti:expression>
      </activiti:field>
      <activiti:field name="activiticustomProperty">
        <activiti:string>TTL=33M21D15h2m</activiti:string>
      </activiti:field>
  </extensionElements>
...
```

Os próprios valores dos elementos identificados com o atributo *name*, transmitem que tipo de informação está envolvida, i.e: que comportamento foi escolhido (*activitibehavior*),

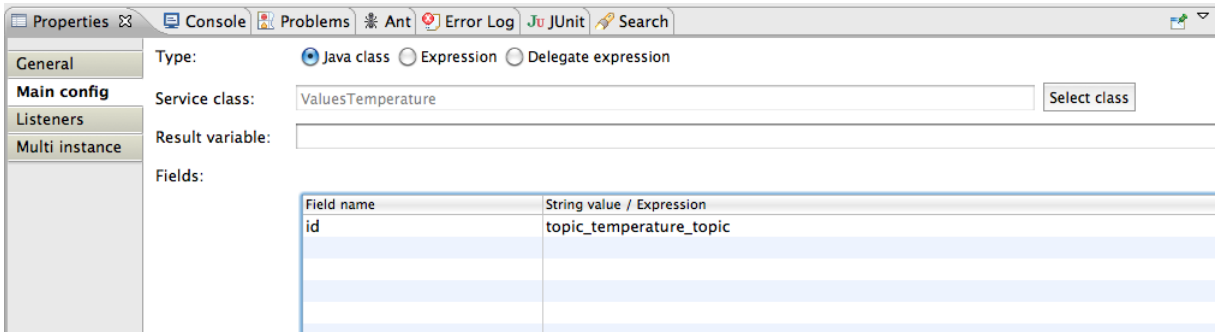


Figura 4.7: Propriedades *service task* publicadora

que subscrição vai ser aplicada (*activitibehaviorCondition*), e informação extra para motor de execução (*activiticustomProperty*), neste caso, o tempo de vida do comportamento.

Em relação ao preenchimento de parametrizações obrigatórias e específicas desse comportamento. Apesar dos papéis estarem definidos pela linha de sequência, certas informações não são obtidas, como por exemplo como são produzidos, guardados ou diferenciados os valores dos produtores/publicadores.

A próxima descrição tem o objectivo de explicar as parametrizações necessárias para as tarefas envolvidas.

No comportamento publicador/subscritor, as parametrizações necessárias para os diferentes papeis são:

A) **Tarefa publicadora**

1) Para as tarefas publicadoras disponibilizarem subscrições (ou tópicos), é necessário parametrizar na área de parametrizações de uma *service task* os tópicos como apresentado na Figura 4.7. Na listagem *fields* adiciona-se os tópicos com a nomenclatura, *topic_nometopico_topic* no campo *string value*. Foi escolhida esta nomenclatura de parametrização, para facilitar a interpretação do campo através de uma expressão regular no motor de execução *Activiti Engine*. O valor do campo *Field name* é ao critério do utilizador.

2) Para associar valores a tópicos no momento de execução, na classe associada à tarefa (*Service class*), tem-se de programaticamente adicionar a estrutura *whiteboard*, mencionada na secção 3.3, o tópico com a seguinte chave-valor:

   <Key,Value> = <topic_nometopico_topic, Classe para obter valores>.

   O código a adicionar a tarefa publicadora tem de ser o apresentado na listagem A2.

   Listing 4.4: Comportamento numa linha de sequência Produtor/Consumidor e Streaming

```
1    IBehaviorSource<?> source = new BehaviorSource(execution);
2      if(topic_temperature_topic != null)
3        if(execution.getVariable(topic_temperature_topic.getExpressionText()) == null)
4          execution.setVariable(topic_temperature_topic.getExpressionText(),source);
```

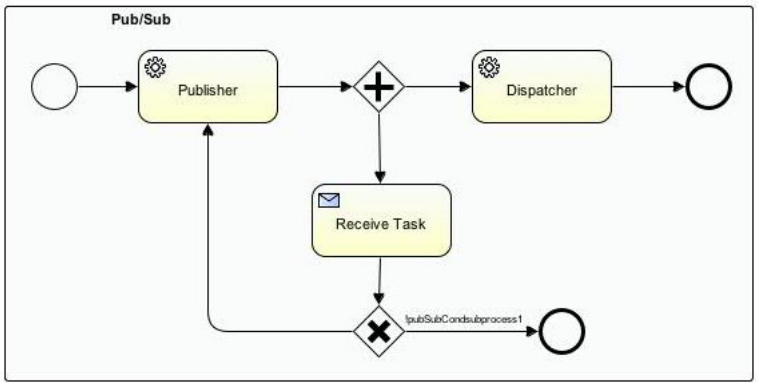


Figura 4.8: *Workflow publisher/subscribe*

3) a classe associada, no exemplo da listagem A2 é a classe *BehaviorSource*, tem de implementar uma interface específica do motor de execução, denominada *IBehaviorSource*. Esta interface representa a fonte de dados do comportamento em execução, para garantir a abstracção na gestão dos tópicos ao utilizador. É oferecida a possibilidade de programar três métodos: o método *GetValue*, onde se programa a obtenção do valor; o método *GetInitial*, que devolve o valor por omissão no início da execução do comportamento; e o método *LastCall*, é invocada quando o comportamento termina, pode ser utilizado para terminar quaisquer ligações efectuadas. Normalmente é no construtor que se realiza a ligação à fonte, isto é, ligação a um *web service*, abertura de um ficheiro, etc. O motor de execução depois de invocar o método *GetValue*, copia o valor obtido a todos os respectivos subscritores , na estrutura *whiteboard*, com chave-valor:

<Key,Value> = <Identificador da tarefa subscritora + "value", valor produzido>.

B) **Tarefa subscritora** Para o caso de se querer obter o valor, basta programaticamente,na classe parametrizada na área de parametrizações, aceder a estrutura *whiteboard* com a chave: Identificador da tarefa subscritora + "value".

C) **Tarefa publicador e subscritora** Tendo este papel o procedimento é uma junção dos dois anteriores.

Os comportamentos ficam parametrizados seguindo as alíneas anteriormente descritas, no entanto a sua aplicação requer uma execução contínua durante um largo período de tempo. Para esse efeito são utilizados os *workflows* de suporte a comportamentos mencionados na secção introdutória.

Certas composições de estrutura e comportamento requerem que a mesma tarefa tenha ambos os papéis de publicador e de subscritor, para este caso é gerado de igual modo um *workflow* de suporte. Por exemplo, no caso de um *pipeline* com três elementos, é gerado um *workflow* para a primeira tarefa (publicadora) e um *workflow* para a segunda tarefa (subscritora da primeira mas publicadora para a terceira). Assim o consumidor

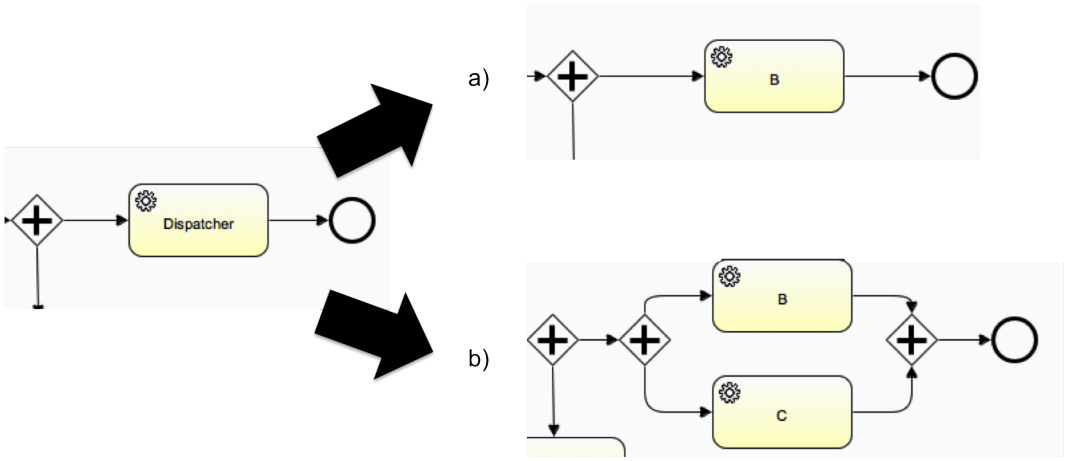


Figura 4.9: Dinamismo da tarefa *dispatcher*

para o primeiro *workflow* passa a ser o *workflow* gerado para a segunda tarefa, no qual a terceira é subscritora. Esta definição demorou um pouco a ser escolhida e para trabalho futuro pode ser alterada.

Para uma ligação entre publicador/subscritor(es), o *workflow* construído segue a configuração apresentada na Figura 4.8. A existência da tarefa *Receive task* é para colmatar uma falha que existe na execução de ciclos (*loops*). Na secção 4.5 será descrita qual falha e como esta tarefa a corrige.

No comportamento publicador/subscritor, a execução das tarefas subscritoras está dependente do valor produzido, o que significa que o número de tarefas é variável. Neste caso decidimos aplicar a tarefa *Dispatcher* para resolver esta questão. Esta tarefa tem o papel de construir em tempo de execução um *workflow* contendo as tarefas que, mediante a subscrição indicada, irão receber o valor publicado.

Com esta característica a composição está construída de forma a respeitar o funcionamento de um comportamento publicador/subscritor, i.e., quando um publicador (*Publisher*) produz um valor, existe a necessidade de avaliar que subscritores devem ser atendidos (*Dispatcher*) ,ao mesmo tempo (*Gateway* paralelo) são produzidos e avaliados novos valores (retoma à tarefa publicadora). O ciclo existente obriga a produção de um valor pelo publicador para a tarefa *Dispatcher* construir o *workflow* com as tarefas subscritoras. A execução termina com o *TTL* parametrizado ou a pedido na consola. O *gateway* exclusivo avalia a variável *pubSubRunsubpipeline1* e caso essa seja falsa termina a execução.

Aplicando este *workflow* ao exemplo apresentado nas parametrizações de linhas de sequência (Figura 4.6), o publicador (*Publisher*) seria a tarefa A e a tarefa *Dispatcher* teria o papel de avaliar se a tarefa B e C seriam atendidas. Supondo que o publicador produz o valor 15 e a tarefa C tem subscrição com a condição "==10", quando o fluxo de execução chega à tarefa *Dispatcher* essa tem o papel de construir um *workflow* dinamicamente de forma a executar a tarefa B. São avaliadas todas as subscrições envolvidas, neste caso as de B e C, e o *workflow* dinâmico assume a configuração apresentada na opção a) da

Figura 4.9. Contudo, supondo que enquanto o *workflow* dinâmico é executado, foi produzido o valor 10, as subscrições são de novo avaliadas e o *workflow* dinâmico fica com a composição apresentada em b) na Figura 4.9. Com estes passos a sustentabilidade da execução é garantida, independentemente do número das tarefas a serem executadas.

Para este *workflow* de suporte ser executado é necessário substituir, em todas as linhas de sequência com ligação a estrutura com comportamento (topologias), a ligação para a estrutura por este *workflow* de suporte.

É utilizado o objecto *EMF* representante das topologias, para confirmar se tem ou não comportamento.

4.2.3.2 Produtor/Consumidor e *Streaming*

Da mesma forma que o comportamento publicador/subscritor, ao parametrizar na topologia o padrão de comportamento, esse comportamento será aplicado as linhas de sequência, ficando o código *BPMN 2.0* com o seguinte aspecto apresentado na listagem 4.5.

Listing 4.5: Comportamento numa linha de sequência Produtor/Consumidor e Streaming

```
...
 <extensionElements>
  <activiti:field name="activitibehavior">
      <activiti:string>Producer/Consumer</activiti:string>
  </activiti:field>
  <activiti:field name="activiticustomProperty">
      <activiti:string>1024</activiti:string>
  </activiti:field>
    <activiti:field name="activiticustomProperty">
      <activiti:string>TTL=33M21D15h2m</activiti:string>
    </activiti:field>
  </extensionElements>
...
```

Os elementos gerados são os mesmos que o comportamento publicador/subscritor excepto o elemento representante das subscrições, no entanto existe a necessidade de parametrizar o tamanho da fila para gestão dos valores produzidos, ao qual utiliza o elemento de informação extra (*activiticustomProperty*).

Em relação a distinção de papéis nestes comportamentos, as parametrizações necessárias são:

A) **Tarefa produtora**

1) Para o caso da tarefa produtora, o campo a parametrizar na listagem *fields* no campo *string value* é: identificador da tarefa produtora + "source".
2) O procedimento 2. do comportamento *publicador/suscritor* aplicasse de novo nesta alínea. sendo a diferença o valor da chave, que utiliza o identificador mencionado na alínea anterior:

<Key,Value> = <Identificador da tarefa produtora+ "source", Classe que implementa *IBehaviorSource*>.

O código a adicionar a tarefa produtora tem de ser o apresentado na listagem A2.

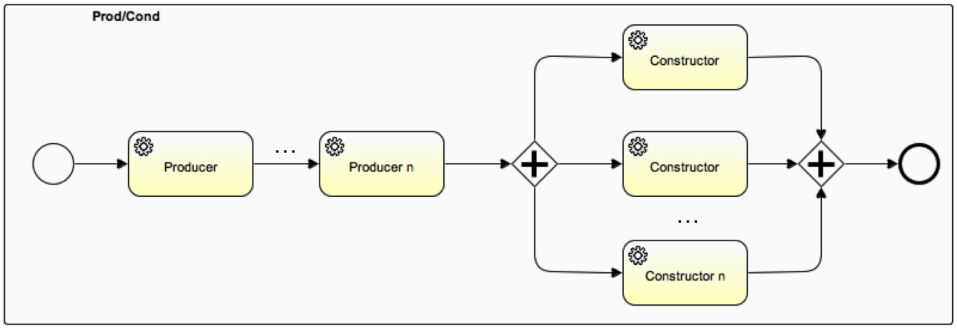


Figura 4.10: *Workflow producer/consumer*

Listing 4.6: Comportamento numa linha de sequência Produtor/Consumidor e Streaming

```
1        IBehaviorSource<?> source = new BehaviorSource(execution);
2        if(sourceClass != null)
3          if(execution.getVariable(sourceClass.getExpressionText()) == null)
4            execution.setVariable(sourceClass.getExpressionText(),source);
```

3) A implementação da classe é igual ao do comportamento anterior, o motor de execução invoca o método *GetValue* para obter o valor e posteriormente adiciona o esse a estrutura *whiteboard* para todos os consumidores, com chave-valor:

<Key,Value> = <Identificador do consumidor + ”value”, valor produzido>.

B) **Tarefa consumidora**

Para o caso de se querer obter o valor, basta programaticamente,na classe parametrizada na área de parametrizações, aceder a estrutura global mencionada, com chave ”Identificador do subscritor + ”value””.

C) **Tarefa produtora e consumidora** Neste caso é necessário, para distinguir os diferentes papeis, para além do valor ”identificador da própria tarefa + ”source””, adicionar outro campo, com valor ”identificador da própria tarefa + ”value””. Com este último valor, programaticamente obtém-se o valor do produzido pelo produtor anterior, com o primeiro campo é para se realizar o mesmo procedimento de um produtor normal.

Em relação ao *workflow* de suporte para o produtor/consumidor e *streaming* tem a composição apresentada na Figura 4.10.

Com n produtores, como é o caso de um *pipeline*(a etapa do meio é consumidora e consumidora), a fase inicial do *workflow* é composta por uma sequência com n etapas representando o inicio da execução de cada produtor. O rácio de consumo das tarefas consumidoras, na definição do padrão produtor/consumidor e *streaming*, tem de ser independente do rácio de produção de valores. Assim, com n consumidores são criados n caminhos paralelos (execuções assíncronas), suportando de tal forma a execução de consumidores com diferentes rácios de consumo. Para esse efeito aplica-se a tarefa construtora para construir ciclos. Estes ciclos servem como suporte à execução de consumidores, visto ser a única forma de manter em execução, numa determinada fase, um *workflow*.

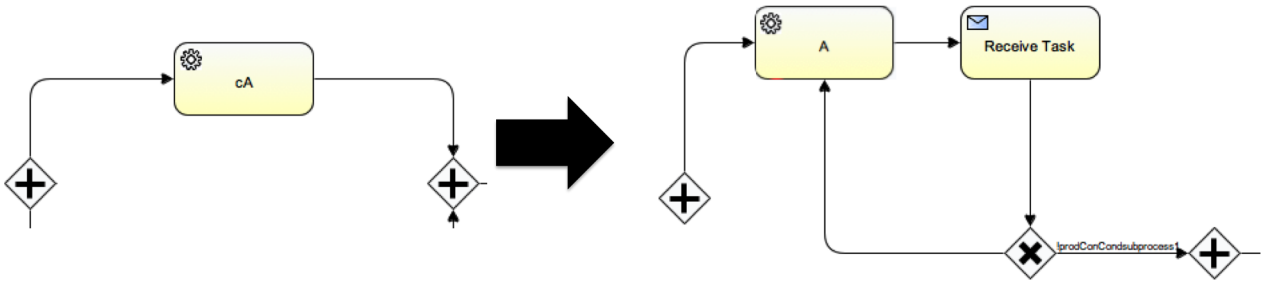


Figura 4.11: Dinamismo da tarefa *constructor*

Executando as tarefas construtoras, o resultado obtido é o observado na Figura 4.11. Em relação às tarefas que são produtoras e consumidoras em simultâneo, as tarefas são suportados por um ciclo, mas também adicionadas à sequência inicial de produtores. A razão para esta decisão será explicada na próxima secção.

Este *workflow* é gerado com o mesmo procedimento realizado no comportamento publicador/subscritor.

Todos estes passos são transparentes ao utilizador, i.e., tanto a geração do *workflow*, como as transformações em tempo de execução envolvidas na execução dos comportamentos.

4.2.4 *Activiti Engine*

No seguimento dos passos descritos anteriormente, esta secção está dividida na seguinte forma: como são interpretadas e aplicadas as parametrizações efectuadas; e como são suportadas as execuções de comportamentos.

O primeiro passo desta dimensão é a importação do ficheiro *BPMN 2.0* gerado. Como já referido, os elementos são interpretados e associados a uma estrutura de dados gerida pelo objecto representante do escopo do processo (*ScopeImpl* ou *ProcessDefinitionEntity*), e cada tarefa é representada por uma classe que contem os dados sobre o seu escopo. Primeiro é realizada a importação das tarefas e depois as ligações entre essas (linhas de sequência).

Contudo, para existir o conceito de comportamento na execução de um *workflow*, adicionámos à classe *ScopeImpl* um mapa que projecta comportamentos na classe que os representa. Para isso foi criada uma classe abstracta denominada *Behavior* que contém os métodos comuns a todos os comportamentos. Para implementar métodos específicos existem as classes *PublisherSubscribeBehavior* e *ProducerConsumerBehavior* (esta também representa o comportamento *Streaming*) que implementam a classe *Behavior*. Em anexo (Figura A.2) está o diagrama de classes completo.

Assim, estando as parametrizações nas linhas de sequência e nas tarefas geradas, o primeiro passo passou por alterar a forma como estes elementos são importados. Através dos elementos extensíveis, é possível distinguir o papel das tarefas e, o que deve ser importado e usado para a execução de comportamentos.

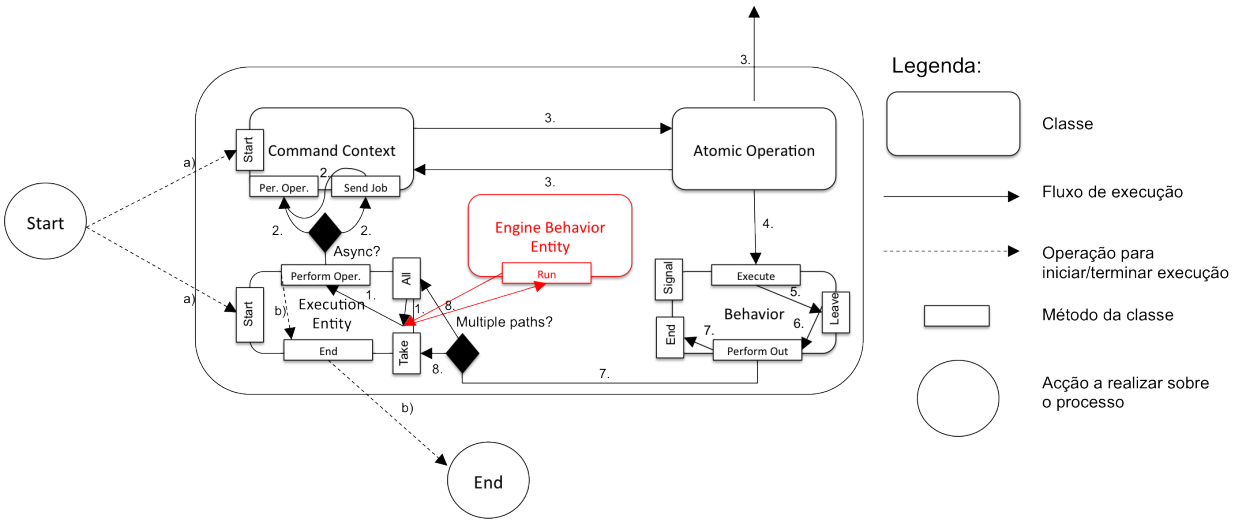


Figura 4.12: Modelo representante do fluxo de execução (com entidade *Engine Behavior Entity*)

Por exemplo, tendo a linha de sequência comportamento parametrizado, ao interpretar essa linha é possível obter a tarefa fonte e de destino, com isso é obtido o identificador da tarefa fonte e, caso ainda não tenha sido, é inicializado o comportamento. A tarefa destino é adicionada como consumidora ou subscritora, e para este último caso, é adicionada a sua subscrição. Para o resto das tarefas, como *dispatcher* ou *constructers*, os elementos extensíveis fornecem informação para o seu escopo de forma a quando forem executadas se saiba o seu papel.

Para suportar estas características, criámos a noção de entidade de motor de comportamentos (*Engine Behavior Entity*). Nesta entidade são executados os comportamentos e, as novas tarefas construtoras de *workflows*. Relembrando a secção 3.3.3, a execução de processo baseia-se em 4 entidades fundamentais e, foi apresentado um esquema para o fluxo de execução. Para não modificar a lógica envolvida, procuramos dentro dos vários passos do fluxo encontrar um ponto em que fizesse sentido aplicar o comportamento. Chegamos à conclusão que, sendo na entidade de execução (*Execution Entity*) onde é decidido o próximo passo a tomar na execução, o fluxo de execução seria re-direccionado para a nova entidade *Engine Behavior Entity*. Como verificamos na Figura 4.12 essa entidade decide qual o próximo passo a tomar, isto é, interpreta o papel da tarefa corrente e aplica as transformações necessárias devolvendo o próximo passo. Caso não tenha qualquer papel, a tarefa corrente é executada normalmente.

De suporte a esta afirmação e usando a Figura 4.13, existem dois pontos fundamentais para esta entidade. O comportamento activo *(Behavior)* e as classes de suporte que transformam as novas tarefas *Dispatcher* e *Constructor* em *workflows*.

Para explicar como este modelo prossegue com o fluxo de execução, os passos envolvidos são explicados de seguida:

1. Neste ponto são analisadas as características da próxima tarefa, acedendo ao escopo

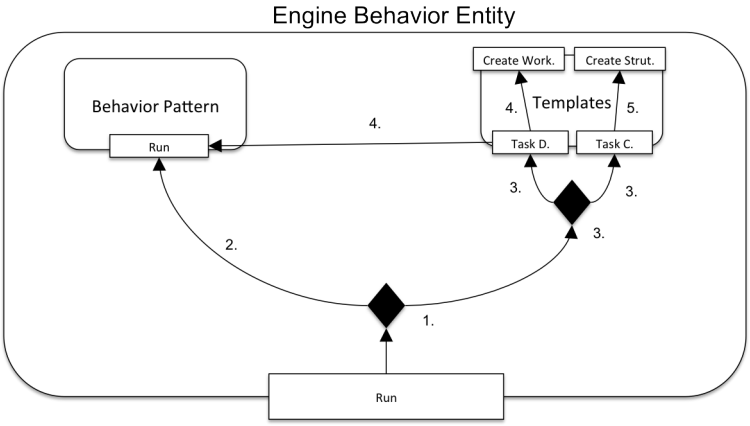


Figura 4.13: Esquema representante do fluxo de execução na entidade *Engine Behavior Entity*

é verificado se o comportamento deve ser executado ou se é uma tarefa *Dispatcher* ou *Constructor*. Caso não tenha nenhum desses papeis é retornada a tarefa, senão continua o fluxo.

2. Se a tarefa recebida pertence a um comportamento (publicador, produtor), é executado o método *run* que vai proceder à execução do comportamento. Para além desse objectivo, o método *run* tem também o objectivo descrito na alínea 4.

3. É verificado se é uma tarefa *Dispatcher* ou *Constructor*

4. Caso seja uma tarefa *Dispatcher*, é executado o método para construir um *workflow*, sendo o grupo de tarefas obtido pela invocação do método *run*, devolvendo a primeira tarefa desse *workflow*, neste caso, um *gateway*.

5. Caso seja uma tarefa *Constructor*, é aplicado o *template*, i.e, é invocado o método *Create Structure* e dependendo do valor obtido pelos elementos extensíveis (*activititypeConstruct*) é gerada a estrutura e retornada a primeira tarefa dessa.

De forma a explicar, com mais pormenor, como esta nova entidade funciona, as próximas descrições estão divididas por comportamento. Primeiro, como se comporta o motor de execução, alterado para o comportamento publicador/subscritor e posteriormente para o produtor/consumidor e *streaming*.

4.2.4.1 Publicador/Subscritor

O suporte à execução deste comportamento está concentrado na classe *PublisherSubscriberBehavior*. É criada uma instância desta classe para todas as linhas de sequências parametrizadas com o comportamento.

O objecto resultante assume o papel de publicador e é parametrizado com os tópicos subscritos, o que implica interpretar os campos provenientes da listagem *fields* da tarefa

fonte. Essa interpretação é realizada através de uma expressão regular que obtém todos os campos com nomenclatura *"topic_nometopico_topic"*. Para cada campo é criada uma instância da classe *Topic*, onde a informação de uma subscrição é gerida , i.e., o seu nome, o valor corrente e os seus subscritores.

Na continuação da importação das linhas de sequência, cada tópico (instância da classe *Topic*) é parametrizado com os respectivos consumidores (nó destino) e suas condições de subscrições (nos elementos extensíveis).

Ao mesmo tempo que são lidas estas parametrizações é verificado se existe informação extra para o comportamento (*activiticustomProperty*), como TTL, se encontra é aplicada as variáveis respectivas do comportamento (ver anexo A.2).

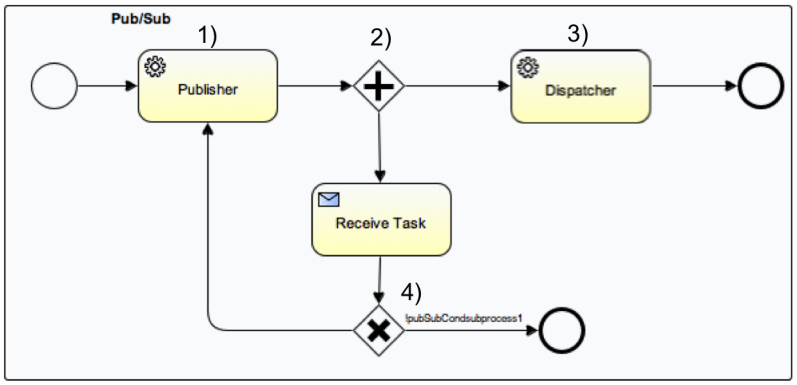


Figura 4.14: *Workflow publisher/subscribe*

Interpretada toda a informação, o esquema do fluxo de execução entra em acção, i.e, todas as entidades são iniciadas. O fluxo ao chegar ao *workflow* de suporte, na Figura 4.14, o comportamento é executado. A partir de agora ao chegar ao método *take*, o fluxo de execução passa pela entidade *Engine Behavior Entity* e através do método *run* o escopo das tarefas é interpretado para se perceber o papel de cada tarefa. Para este comportamento o fluxo de execução na nova entidade comporta-se na seguinte forma:

1) O método *run* ao analisar esta tarefa é verificado que é uma tarefa publicadora, com o seu identificador (que foi substituído pelo identificador do comportamento), é obtido o comportamento activo na estrutura gerida por *ScopeImpl*. O método *run* do comportamento é invocado, tal como, se verifica no passo 2. da Figura 4.13, e na primeira vez que é invocado é criada uma *thread* e uma *timer task*. A *thread* é criada para atender os pedidos provenientes da linha de comandos, por exemplo terminar o comportamento, que altera o valor da variável *pubSubCondsubprocess1*. A *timer task* muda essa variável quando o tempo parametrizado no campo *TTL* for atingido. O publicador ao executar, adiciona na estrutura *whiteboard* as fontes dos tópicos , i.e, as classes quem implementam *IBehaviorSource*. Depois de executado o publicador, são percorridos todos os tópicos e, obtendo a classe na estrutura *whiteboard*, para cada uma é executado o método *GetValues* para actualizar o valor corrente do tópico. A execução do publicador está num ciclo devido dependência dos valores produzidos e a tarefa *Dispatcher*,

para construir o *workflow* de subscritores.

2) Neste ponto, em paralelo é executada a tarefa *Dispatcher* e, o fluxo continua para actualizar os valores dos tópicos.

3) O método *run* do comportamento (passo 2. da Figura 4.13) é de novo invocado. Com a propriedade obtida nos elementos extensíveis (*activitirunBehavior*) é obtido o comportamento. No entanto a execução do método vai realizar outra acção, é pedido para cada tópico, a lista das tarefas que as suas subscrições sejam verdadeiras. Com o grupo obtido, a tarefa *Dispatcher* constrói o *workflow* necessário para executar todas as tarefas, com o passo 4. da Figure 4.13.

4) Neste ponto é analisada a variável *pubSubCondsubprocess1*, caso seja falsa, ou porque o TTL terminou ou foi alterado a pedido, o comportamento termina. Caso seja verdadeira, o fluxo continua e retoma o passo 1) que actualiza os valores dos tópicos.

4.2.4.2 Produtor/Consumidor e *Streaming*

O suporte à execução destes comportamentos está concentrado na classe *ProducerConsumerBehavior*. Tal como o comportamento publicador/subscritor, é criada uma instância desta classe, para todas as linha de sequência parametrizadas com o comportamento.

Com a classe comportamento instanciada, a tarefa fonte das linhas associada é como produtora e as tarefas destino como consumidoras.

No entanto, no caso do Produtor/Consumidor é parametrizada a gestão de uma fila de valores de entrada.

Existe esta noção de gestão de uma fila de valores, porque no âmbito desta tese, o comportamento produtor/consumidor foi desenvolvido de acordo com a lógica existente na tese apresentar no capítulo 1, isto é, conceito de sessão. Todos os consumidores têm a mesma visão de conceito de sessão, o que implica que, o produtor ao ser cliente de uma sessão, a informação obtida deve propagar-se para os consumidores. Desta forma não foi implementado a visão geral deste comportamento, porque um dos objectivos desta tese, é oferecer um *front-end* para a construção de *workflows*, com tarefas como clientes de sessões. Desta forma, os dados mantêm-se coerentes com os valores produzidos no contexto da sessão do produtor.

Para garantir esta gestão, a fila é representada como uma estrutura auxiliar. lista que quando uma posição é acedida n vezes, sendo n o número de consumidores, esse valor é retirado da lista.

No momento em que é instanciada a classe é verificado também se existe informação extra para o comportamento (*activiticustomProperty*), como TTL, tamanho da fila, etc. Ao encontrar essa informação, os valores obtidos são associados as variáveis respectivas (ver anexo A.2).

De novo é utilizada a entidade *Engine Behavior Entity*, para interpretar os papeis das tarefas e, retornar qual o próximo passo do fluxo de execução. O *workflow* de suporte

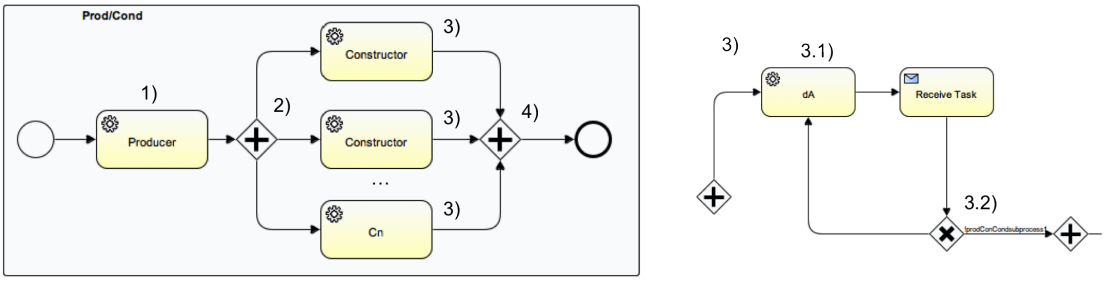


Figura 4.15: *Workflow producer/consumer*

ao comportamento, na Figura 4.15, é executado e o fluxo de execução comporta-se na seguinte forma:

1) O método *run* ao analisar que esta tarefa é produtora, como o seu identificador (identificador calculado da forma apresentada na secção do publicador/subscritor) é obtido o comportamento activo na estrutura gerida por *ScopeImpl*. O método *run* do comportamento é invocado, tal como, se verifica no passo 2 da Figura 4.13. Neste caso o produtor é executado em primeiro lugar, adicionando a fonte de dados à estrutura *whiteboard*. Caso houvesse mais do que um produtor esta fase teria mais etapas, sendo cada uma responsável por iniciar e adicionar a fonte de dados na *whiteboard*. Depois de executado o produtor são executadas três *threads* e uma *timer task*: uma *thread* é criada para atender os pedidos provenientes da linha de comandos (por exemplo terminar o comportamento); as outras duas para a gestão das taxas de produção e consumo, como descrito de seguida; e a timer task para o *TTL* do comportamento.

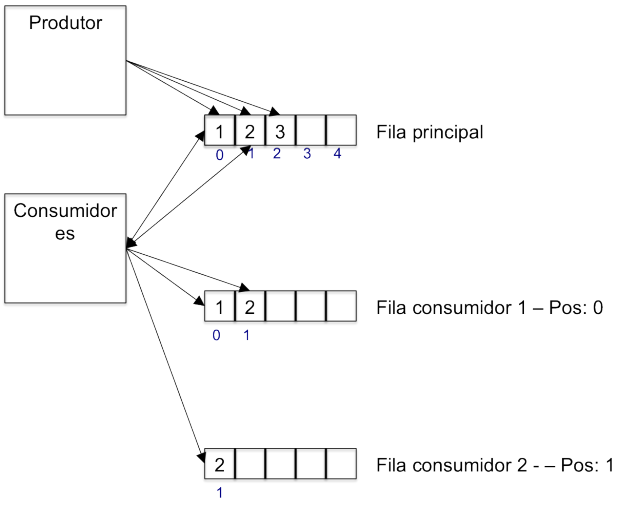


Figura 4.16: Gestão da dos valores na fila

As duas *threads* mencionadas representam a independência no rácio de produção e de consumo, como apresentado na Figura 4.16. A fila tem comportamentos diferentes consoante o comportamento em execução. No caso de *streaming*, quando a fila está cheia, a *thread* responsável por gerir a fila remove o primeiro valor, para o caso do

produtor/consumidor remove o elemento quando todos os consumidores consumiram. No caso da fila principal estar cheia, é inicializada uma nova fila para adicionar valores em *cache*.

Para garantir o ritmo a diferentes velocidades de consumo, decidimos que cada consumidor tem uma fila de valores, a *thread Consumidores* tem a responsabilidade de obter os valores na fila principal e distribuir por essas filas, tendo para cada consumidor uma posição corrente de consumo. Por exemplo Figura 4.16, vemos que o consumidor 2 já consumiu o valor 1 (está na posição 1 da fila principal, número por baixo da posição da lista), ao contrário do outro consumidor.

Caso exista uma tarefa produtora/consumidora essa gestão é diferente. Por exemplo a Figura 4.17, ilustra que a fila do *consumidor 1* também é uma fila principal, mas neste caso para o *consumidor 2*. Para garantir que o *produtor 2* só produz quando o produtor 1 produziu, é realizada uma gestão com monitores, i.e, quando a tarefa *produtor 1* produzir notifica (*notify*) a *thread* associada ao *produtor 2* para acordar. Depois dessa consumir e produzir fica de novo a espera (*wait*) de outros valores. A tarefa *Consumidor 2* no ciclo de atendimento não é executada quando não existem valores para consumir.

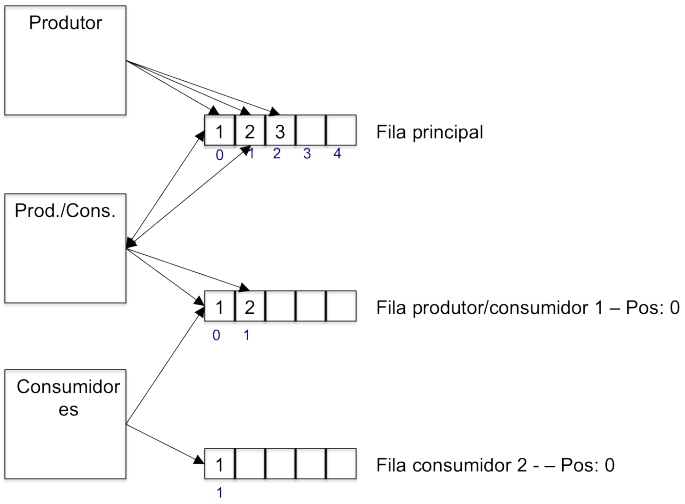


Figura 4.17: Gestão dos valores na fila com tarefa produtora/consumidora

2) Neste ponto, em paralelo, são executadas as n tarefas construtoras dos n consumidores existentes.

3) A tarefa construtora, para este comportamento, transforma-se num ciclo de suporte ao consumidor, para garantir que a execução se mantém activa e a tarefa consumidora é executada. Como podemos verificar em **3.1)**, aqui o método *run* do comportamento (passo 2. da Figura 4.13) é executado para que a tarefa consuma o primeiro valor da fila. É retirado o primeiro valor da fila e o valor obtido é adicionado à estrutura *whiteboard* com chave o identificador da tarefa+"value". O passo **3.2)**, da mesma forma

que o comportamento anterior, avalia a variável *proConCondsubprocess1*, que pode ser alterada a pedido, pela *timer task* ou pelo valor produzido, i.e, sendo igual ao valor parametrizado no campo *terminator*. Caso seja falsa, termina o ciclo de todos os consumidores, caso seja verdadeiro, a execução mantém-se activa.

4) O comportamento ao terminar, a execução do *workflow* termina quando todos os ciclos terminarem.

4.3 Reconfiguração estática

No âmbito desta ferramenta, com comportamentos associados a estruturas, as reconfigurações não são genéricas, isto é, são confinadas ao funcionamento de comportamentos e a própria composição nos padrões estruturais. Assim, dentro desse contexto, é possível aplicar reconfigurações antes e durante a execução de um *workflow*.As reconfigurações antes da execução baseiam-se na alteração da estrutura e na associação de comportamento, no qual chamamos reconfigurações estáticas. No que se refere as adaptações aplicáveis durante a execução de um *workflow*, intituladas reconfigurações dinâmicas, introduzimos suporte a alteração do comportamento associado tal como adição de novas tarefa. De mencionar que, para oferecer diferentes formas de aplicar estas reconfigurações, foi criada a possibilidade de usar um ficheiro *XML*, com o *schema* apresentado em anexo (A.2), para aplicar as reconfigurações mencionadas. Este ficheiro também é utilizado para aplicar as reconfigurações dinâmicas à componente *Activiti Designer* visto que, como podemos verificar na secção 3.2.1, as componentes envolvidas são independentes entre si.

A explicação de como foram implementadas as reconfigurações estáticas segue no texto, estando as reconfigurações dinâmicas descritas na secção 4.4.

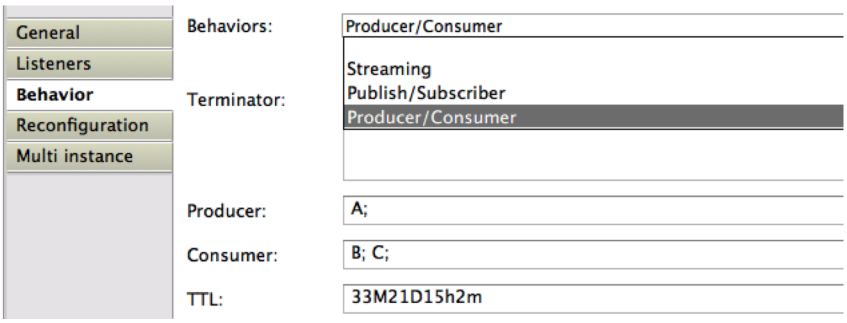


Figura 4.18: Parametrizar o comportamento.

As reconfigurações estáticas estão concentradas na componente *Activiti Designer* e abrangem a adição e remoção de tarefas, e a mudança de comportamento nos padrões estruturais.

Dois operadores criados foram *increase* e *decrease* para os padrões estruturais. Para

esse efeito foram adicionados os operadores à secção de propriedades dos padrões estruturais (recorrendo à técnica descrita na secção 4.2). Através da Figura 4.19 é possível verificar como ficaram apresentadas essas operações.

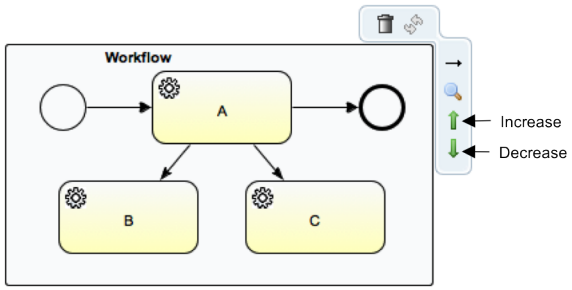


Figura 4.19: Novas opções no menu de contexto.

Os operadores *increase* e *decrease* estão inseridos no menu de contexto dos padrões estruturais de forma a facilitar a aplicabilidade dessas operações. Ao pressionar a opção *increase* é adicionada uma *empty task* com uma linha de sequência. A criação da linha de sequência varia por estrutura, i.e., sendo o novo elemento o destino, o elemento fonte é:

Estrela O elemento produtor/publicador da estrela, i.e., elemento núcleo.

Pipeline O último elemento do *pipeline*.

Anel O último elemento do anel, contudo a ligação desse para o primeiro deixa de existir e o novo elemento fica com essa ligação.

A linha de sequência herda o comportamento parametrizado na estrutura. A operação *decrease* remove a última tarefa inserida nessa estrutura, no caso da Figura 4.19 a tarefa removida seria a tarefa C.

A implementação destas operações requereu a incorporação da noção de *increase* e *decrease* sobre um elemento sub-processo com comportamento. No projecto *Eclipse Graphiti*, onde são construídos os menus de contexto, criaram-se as classe *decreasetask* e *increasetask*, que são instanciadas quando o respectivo botão é pressionado. O primeiro passo a ser realizado quando é pressionado um dos botões é a obtenção da *EClass* do sub-processo para ser usado nessas classes. Para a *decreasetask* é removida a última tarefa na estrutura de tarefas incluída na *EClass* do sub-processo, e invocada a função da *framework Eclipse Graphiti* para apagar a tarefa graficamente. Em relação a *increasetask* é criada uma *empty-task*, inicializando um objecto EMF, e essa é adicionada à estrutura de tarefas da *EClass* do sub-processo. Para se visualizar graficamente, é invocado o método de adicionar tarefas da *framework Eclipse Graphiti*.

Como foi brevemente referido no texto introdutório a esta secção, as reconfigurações podem ser realizadas via um ficheiro XML , sendo porém necessário adicionar o ficheiro ao projecto *Activiti* do *workflow* e seguir a ordem de passos da Figura 4.20:

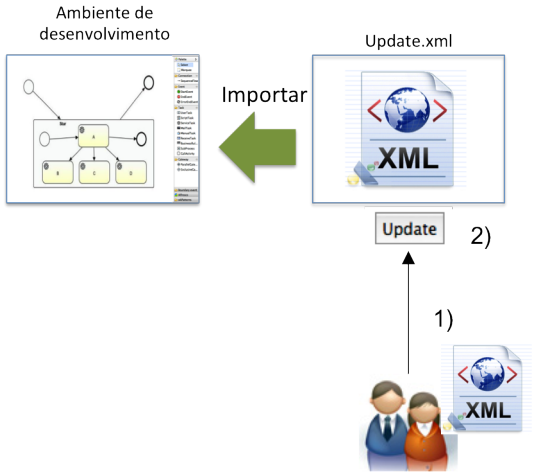


Figura 4.20: Interacção do utilizador com *XML* de reconfigurações.

1) criar o ficheiro com o seguinte formato:

Listing 4.7: Exemplo ficheiro *XML*

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
  <reconfigurations id = "pipeline1">
    <behavior id="Publisher/Subscriber">
      <condition>
        topic_value_topic > 30
      </condition>
    </behavior>

    <addtask id= "servicetask">
      <condition>
        topic_value_topic == 30
      </condition>
      <fields>
        <field>
          <name>teste</name>
          <value>value</value>
        </field>
        <field>
          <name>teste2</name>
          <value>value2</value>
        </field>
      </fields>
      <javaClass>
        teste.java
      </javaClass>
    </addtask>
    <addtask id= "servicetask2">
      <fields>
        <field>
          <name>teste</name>
          <value>value</value>
        </field>
        <field>
          <name>teste2</name>
          <value>value2</value>
        </field>
      </fields>
      <javaClass>
        teste.java
      </javaClass>
    </addtask>
  </reconfigurations>
```

Neste ficheiro as seguintes parametrizações são aplicadas:

(a) O comportamento é alterado para publicador/subscritor com subscrição para todos ">30", contudo, como se pode verificar no ficheiro, na adição da tarefa *servicetask* a subscrição é alterada para "==30"(linha 11).

(b) As novas tarefas usam a classe "teste.java"e os campos *fields*(elemento *XML fields*) para a sua execução.

2) foi adicionada na área de parametrizações do diagrama principal um botão *update*. Ao pressionar é realizada a interpretação do ficheiro e aplicados os operadores graficamente no *workflow*, neste caso mudança de comportamento e adição de duas tarefas.

Foi disponibilizada esta funcionalidade, de forma a oferecer diferentes escolhas para parametrizar/reconfigurar o *workflow* em tempo estático.

Na próxima secção é explicado (como e porquê) o uso deste ficheiro nas reconfigurações dinâmicas.

4.4 Reconfiguração dinâmica

O principal objectivo das reconfigurações dinâmicas é garantir que o sistema adapte a sua execução (ou funcionamento) a alterações proveniente do exterior ou interior ao seu contexto.

No âmbito desta tese, a aplicabilidade das reconfigurações está restringida à execução de comportamentos, mas também à alteração dos padrões estruturais (e.g. inserir tarefas em tempo de execução).

Alguns exemplos de aplicabilidade destas reconfigurações, podem ser em situações em que a obtenção de novas variáveis ou a necessidade de alterar o comportamento, são necessárias por questões de ordem financeiras ou por cenários de reacção a uma emergência.

A razão para a criação nesta ferramenta destas operações, deve-se ao facto de oferecer mecanismos importantes para garantir a adaptabilidade em diferentes cenários. No entanto está planeado para trabalho futuro adicionar/criar novas operações.

4.4.1 Processo de extensão

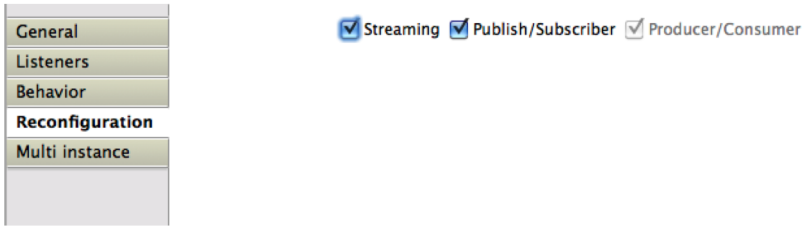


Figura 4.21: Configuração das reconfigurações dinâmicas de comportamentos.

O foco principal de desenvolvimento foi na interacção entre componentes e na componente *Activiti Engine*. É oferecida a possibilidade ao utilizador de aplicar estas reconfigurações por comando, utilizando a mesma lógica envolvida para terminar um comportamento (secção 4.2).

A interpretação dos comandos emitidos em tempo de execução requer que o *workflow* de suporte aos comportamento tenha a possibilidade de aplicar reconfigurações. Para isso, e seguindo a mesma lógica de garantia da consistência de execução usada nas tarefas *constructor* e *dispatcher*, foi criada uma tarefa com o papel de "aplicar reconfigurações em tempo real", denominada reconfiguradora (*"reconfigurator"*). Esta tarefa foi adicionada aos *workflows* de suporte de comportamentos e utiliza os elementos extensíveis mais a notação *Activiti* para representar a tarefa. A tarefa foi posicionada nesses *workflows* de forma a estar em execução em paralelo com a execução de comportamentos, com o objectivo de interpretar os comandos enviados pelo utilizador. Os detalhes para cada comportamento são descritos na próxima secção.

As alterações na componente *Activiti Designer* resumem-se à criação de uma nova área de parametrizações para os padrões estruturais, onde é possível escolher que reconfigurações serão aplicadas em tempo real. Para suportar essas reconfigurações foi adicionado, ao objecto EMF do sub-processo, o atributo reconfigurações (representado por uma estrutura de dados). A estrutura é preenchida com as opções seleccionadas como apresentado na Figura 4.21, para ser usada na geração *BPMN 2.0.* da tarefa reconfiguradora, ficando com a estrutura apresentada em 4.8.

Listing 4.8: Tarefa reconfiguradora *BPMN 2.0*

```
1  <serviceTask id="substar1reconfigurations" name="reconfigurations">
2     <extensionElements>
3       <activiti:field name="activitireconfiguration">
4         <activiti:string>Streaming</activiti:string>
5       </activiti:field>
6       <activiti:field name="activitireconfiguration">
7         <activiti:string>Publish/Subscriber</activiti:string>
8       </activiti:field>
9       <activiti:field name="activitireconfiguration">
10        <activiti:string>Producer/Consumer</activiti:string>
11      </activiti:field>
12    </extensionElements>
13  </serviceTask>
```

O atributo *name* (linhas 3,6,9) é utilizado para representar os possíveis comportamentos a reconfigurar e para diferencia o papel da tarefa na entidade *Engine Behavior Entity*.

4.4.2 Ligação entre *Activiti Designer* e *Activiti Engine*

Para aplicar reconfigurações em tempo real decidimos utilizar a linha de comandos disponibilizada pela *framework JUnit*, de maneira a oferecer ao utilizador uma forma simples de aplicar reconfigurações. Como as componentes *Activiti Designer* e *Activiti Engine* são independentes entre si durante a execução de um *workflow*, as reconfigurações aplicadas a estruturas não são visíveis graficamente. Para garantir a persistência e oferecer a possibilidade de aplicar essas reconfigurações graficamente, sempre que se realiza um comando

é adicionado o elemento respectivo ao ficheiro *XML*, seguindo a estrutura apresentada na secção anterior.

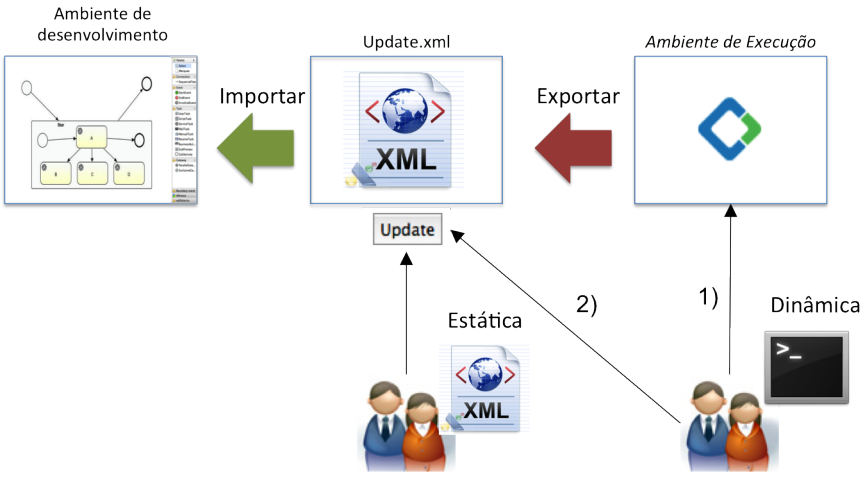


Figura 4.22: Interacção do utilizador com *XML* de reconfigurações.

Os passos envolvidos para esse efeito estão exibidos na Figura 4.22:

1. Os comandos disponibilizados são:

   - Para alterar comportamento:
     reconf: comportamento [tarefa 1 := subscrição 1 ... tarefa n : subscrição n]
     Exemplo: "reconf: pubsub star1servicetask1 := value == 30 | star1servicetask2 := value == 20 | star1servicetask4 := value < 10"
   - Para adicionar tarefa:
     add: servicetask | | classe java [| | subscrição]
     Exemplo: "add: servicetask | | JavaTaskPS | topic_value_topic > 35"

   Ao interpretar e aplicar estes comandos a componente *Activiti Engine* cria/adiciona essas reconfigurações ao ficheiro *XML*.

2. Depois de executado o *workflow*, o utilizador pode aplicar as reconfigurações aplicadas graficamente pressionado o botão *update*.

Os *workflows* de suporte para suportar reconfiguração contêm têm na sua composição a tarefa reconfiguração. Assim sendo, de seguida são explicadas as alterações realizadas para cada comportamento.

4.4.2.1 Publicador/subscritor

Como explicado anteriormente, o *workflow* de suporte tem o objectivo de construir dinamicamente o atendimento das tarefas subscritoras, sendo a tarefa *Dispatcher* a tarefa responsável por esse dinamismo. Assim, o *workflow* de suporte foi alterado de forma a executar paralelamente a nova tarefa reconfiguradora com a tarefa *dispatcher*. Assim é

garantida a execução em paralelo dos subscritores e da tarefa que interpreta e executa os comandos para reconfiguração do utilizador.

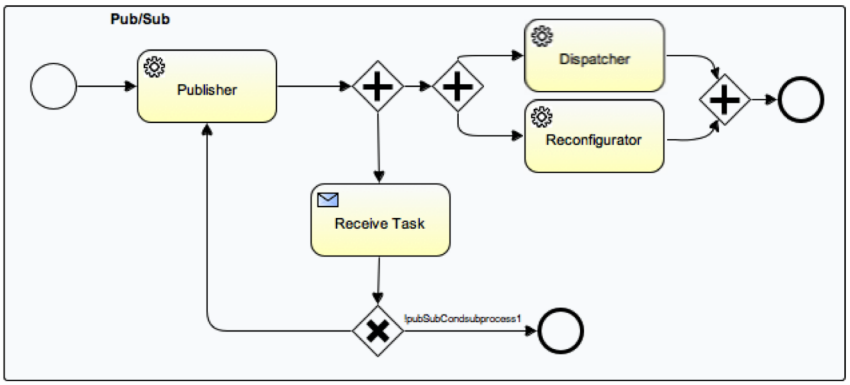


Figura 4.23: *Workflow* publicador/subscritor com tarefa reconfiguradora

O *workflow* fica com a estrutura ilustrada na Figura 4.23. A operação *inserir tarefa* não afecta a estrutura do *workflow*, contudo, se o comando emitido requerer a mudança de comportamento para produtor/consumidor ou *streaming* o papel desta tarefa reconfiguradora é transformar-se no *workflow* de suporte desses comportamentos.

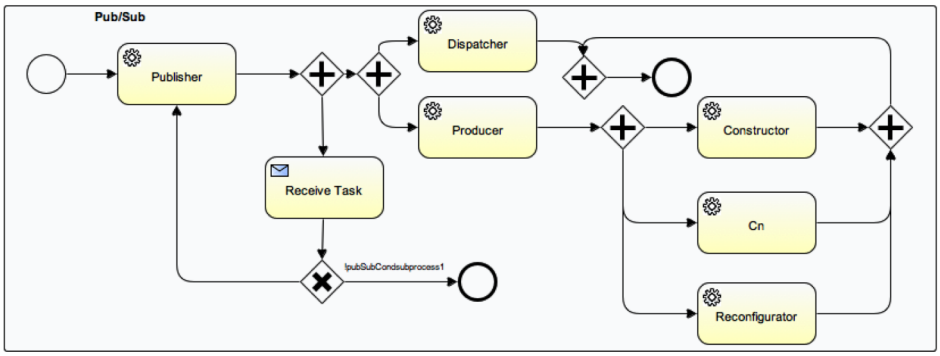


Figura 4.24: *Workflow* publicador/subscritor com tarefa reconfiguradora (Resultado)

Por exemplo, ao aplicar uma reconfiguração para produtor/consumidor o *workflow* de suporte passa a ter a composição apresentada na Figura 4.24. O número de tarefas construtoras varia consoante o número de subscritores. O *workflow* gerado está também preparado para aplicar os operadores de reconfiguração, já que contem a tarefa reconfiguradora.

4.4.2.2 Produtor/consumidor e *Streaming*

O *workflow* de suporte destes comportamentos contêm as tarefas construtoras que transformam-se em ciclos de atendimento aos consumidores. Desta forma a tarefa reconfiguradora é executada em paralelo com as tarefas *construtoras*. Assim é garantida a execução em paralelo dos construtores e da tarefa que interpreta e executa os comandos de reconfiguração

do utilizador.

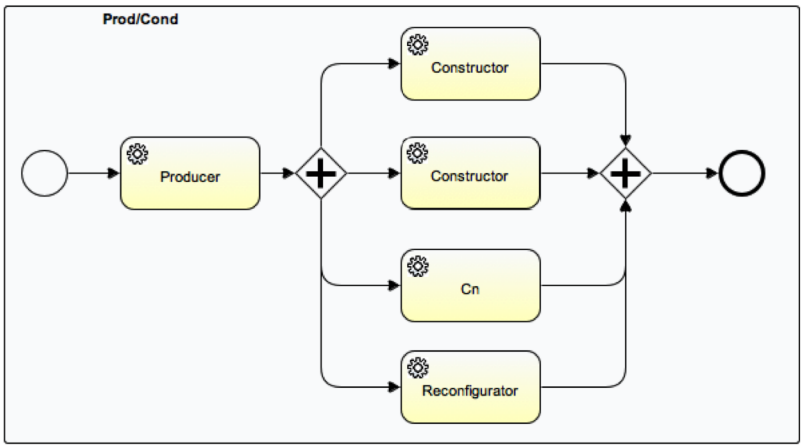


Figura 4.25: *Workflow* produtor/consumidor com tarefa reconfiguradora

O *workflow* fica com o aspecto da Figura 4.25. O operador inserir tarefa não afecta a estrutura do *workflow*, contudo se o comando for mudar de comportamento para publicador/subscritor, o papel desta tarefa reconfiguradora é transformar-se no *workflow* de suporte desses comportamentos, de forma a que a execução passe a ser sobre o novo comportamento. Caso seja para *streaming* deixa de haver gestão da fila, na secção do *Activiti Engine* é explicado em maior detalhe.

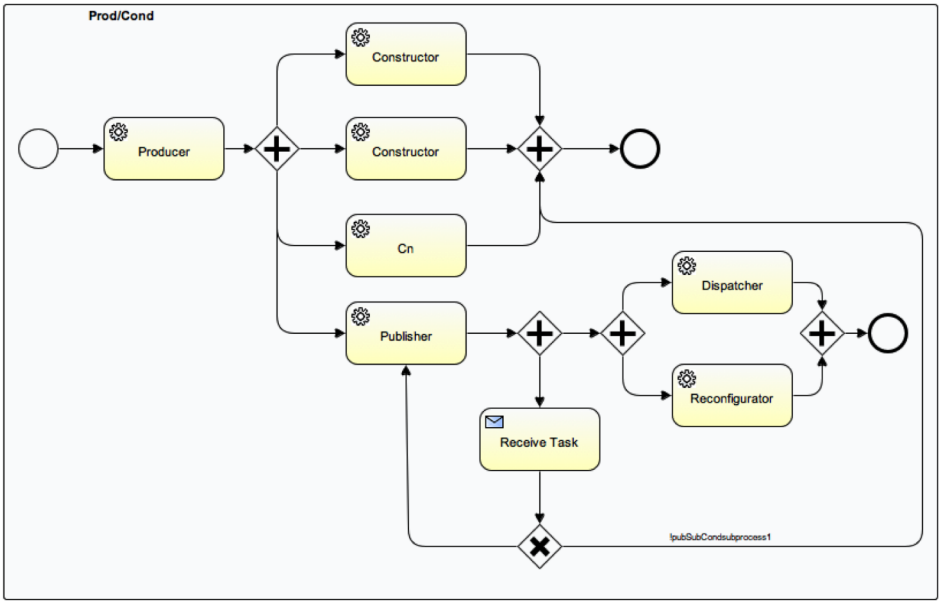


Figura 4.26: *Workflow* produtor/consumidor com tarefa reconfiguradora (Resultado)

Por exemplo, ao aplicar uma reconfiguração para publicador/subscritor o *workflow* de suporte passa a ter a composição apresentada na Figura 4.26. Existe o ciclo de dependência entre a tarefa publicadora e a tarefa *dispatcher*, e existe também o paralelismo entre

a tarefa *dispatcher* com outra tarefa reconfiguradora, o que implica que é possível aplicar de novo as operações de reconfiguração.

4.4.3 *Activiti Engine*

As acções de reconfigurações dinâmicas têm de ser interpretadas pelo motor de execução (*Activiti Engine*), para que possam alterar a execução em curso. Para esse efeito é utilizada a mesma *thread* que atende os pedidos de terminação.

Em relação ao fluxo de execução dos comportamento. foi criada uma classe denominada *Reconfiguration* para representar uma reconfiguração activa do comportamento. O utilizador ao escrever um comando, implicitamente diz que essa reconfiguração é a reconfiguração activa no comportamento em execução. Para isso, foi criado um atributo na classe *Behavior*, para criar a noção de reconfiguração activa (como verificado no diagrama de classes em anexo (*Behavior* Figura A.2)).

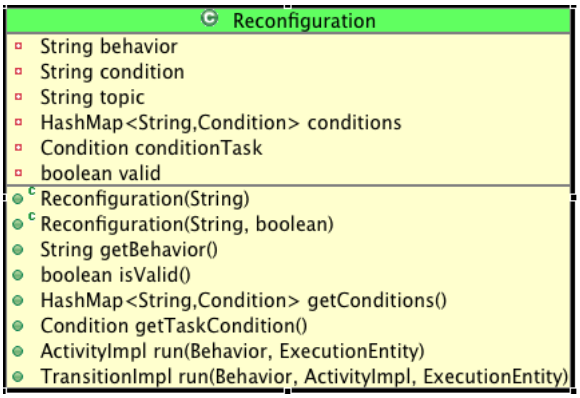


Figura 4.27: Classe *Reconfiguration*

Esta classe representa as duas operações mencionadas, i.e., mudança de comportamento ou adição de tarefa, sendo o atributo *"behavior"* da classe apresentada na Figura 4.27 o discriminador na execução do método *run* para realizar a operação devida.

A execução de uma tarefa reconfiguradora recorre a entidade *Engine Behavior Entity* para interpretar o seu papel na execução. Para acomodar tal funcionalidade foi adicionada a noção de entidade de reconfiguração e o esquema do fluxo de execução na entidade *Engine Behavior Entity* fica com a estrutura da Figura 4.28.

A execução desta entidade, como mencionado na secção 4.2, avalia o papel da tarefa activa, como nessa mesma secção foram explicados os passos 1 ao 5, de seguida são explicados o resto dos passos:

6) A entidade avalia o papel da tarefa e verifica que é uma tarefa reconfiguradora.

7) Neste ponto através do atributo *"behavior"* executa o método respectivo, ou seja, mudar de comportamento ou mudar de tarefa

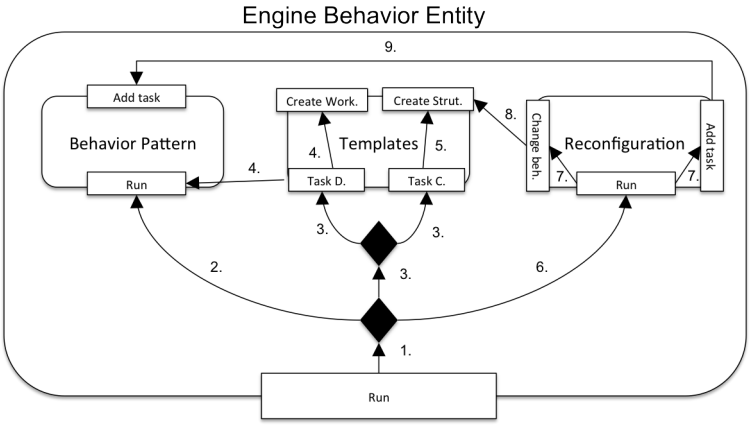


Figura 4.28: Esquema representante do fluxo de execução na entidade *Engine Behavior Entity* com reconfiguração

8) Caso seja para mudar de comportamento, o primeiro passo realizado é a instanciação da classe do comportamento a ser aplicado e a transferência de informação (publicador/produtor, subscritores/consumidores,etc) do comportamento activo para o comportamento novo. O novo comportamento depois de instanciado fica activo e o antigo é terminado. De seguida, são utilizados *templates* que representam a composição de um *workflow* de suporte a comportamentos, i.e, caso seja para mudar para publicador/subscritor aplica o *template PubSubConstructor*. Caso contrário aplica o *template ProConConstructor*. Estes *templates* contêm uma função que que cria programaticamente a composição dos *workflows* de suporte a comportamentos. O caso particular da reconfiguração entre produtor/consumidor e *streaming* apenas altera-se a parametrização do comportamento corrente, para activar, ou desactivar, a gestão da fila (alterar o valor da variável *"management"* da classe *ProducerConsumerBehavior*).

   A razão é porque a única diferença entre estes dois comportamentos é a gestão da fila de resultados.

9) Para o caso de adicionar uma tarefa, é invocado um método no comportamento activo que recebe a informação interpretada, cria a tarefa e adiciona-a à estrutura *"consumers-Subscribers"*, representante das tarefas consumidoras/subscritoras, da classe *Behavior*. Se a tarefa for subscritora, o atributo *"condition"* da classe *Reconfiguration* será a subscrição a associar.

De referir que quando são realizados os passos 8) e 9) são criados os elementos *XML* respectivos no ficheiro *XML* para aplicar as reconfigurações como apresentado na Figura 4.22.

4.5 Discussão critica

A ferramenta Activiti contem várias vantagens desde logo a facilidade e os mecanismos oferecidos para estender as componentes *Activiti Designer* e *Activiti Engine*. A robustez das funcionalidades existentes cresceu devido ao aumento do uso da notação *BPMN 2.0* e a existência de uma comunidade activa. Contudo, no decorrer do desenvolvimento desta tese, foram encontrados vários problemas inerentes principalmente à componente *Activiti Engine*. Todo ocorre durante fluxo de execução, como verificamos nos esquemas anteriores o fluxo avança com invocações de diferentes métodos das diferentes entidades, no entanto, esta sequência provoca que os métodos estejam dependentes que os outros terminem, o que provoca que a pilha de execução JVM encha, provocando a excepção de *Stack OverFlow*.

Como mencionamos anteriormente na secção 4.2 (Processo de extensão - interacção entre componentes - publicador/subscritor) , esta situação ocorre primordialmente nas estruturas em ciclo, e as *receive tasks* têm um papel fundamental neste problema. O propósito desta tarefa é parar a execução e limpar a pilha de execução, mantendo o processo persistentemente, para programaticamente invocá-lo com o método *signal* (entidade *Behavior*). Para utilizar esta particularidade, damos a possibilidade ao utilizador de gerar o ficheiro *JUnit* com gestão de *receive tasks*. Quando uma *receive task* executa, é adicionado o seu identificador a um estrutura fila, sendo essa fila adicionada a estrutura *whiteboard* para ser usada no ficheiro *JUnit*. Nesse ficheiro gerado é criado um ciclo, que mantém viva a execução percorrendo essa estrutura de *receive tasks*. Encontrando-se vazia significa que o processo terminou.

Outro dos problemas foi com as tarefas assíncronas, isto é, a primeira chamada assíncrona de uma tarefa é realizada correctamente, no entanto quando essa ocorre é alterado o campo ”estado” na linha de base de dados, para o valor ”executada”, o que implica que ao executar segunda vez retorna a excepção que já foi executada uma vez. A solução para este problema, e tendo requisito ser uma *service task*, decidimos criar uma *thread* para cada tarefa consumidora/subscritora. Essa *thread* fica em *wait* até ser notificada para executar o comportamento da tarefa associada. Esta gestão, como se verifica na classe *Behavior* em anexo (Figura A.2), é mantida na estrutura *consumersBehaviors* que é percorrida sempre que seja necessário executar as tarefas consumidoras/subscritoras.

5

Exemplos de Aplicação

Este capítulo tem como objectivo ilustrar as potencialidades do nosso protótipo na construção e execução de *workflows* no contexto de uma ferramenta *BPM*. São utilizados as funcionalidades criadas na ferramenta de forma a compor *workflows* mais ricos e com potencialidades que não são oferecidas numa ferramenta comum de composição e/ou de execução *BPMN 2.0*. Para esse efeito, decidimos dividir a validação do protótipo em três exemplos :

1. Como a notação *BPMN 2.0* é específica da área de *business*, procurámos no meio empresarial um exemplo de um *workflow* aplicado a um caso real. Foi feita a implementação *BPMN 2.0* desse *workflow* com e sem padrões de forma a demonstrar as vantagens do nosso protótipo.

2. O segundo exemplo, no domínio dos sistemas *Dynamic data driven application system (DDDAS)*, descreve execuções de padrões de comportamentos combinados com padrões estruturais, sendo aplicadas reconfigurações dinâmicas e estáticas a esses padrões.

3. O exemplo anterior é enriquecido com alguns exemplos conceptuais reforçando a validade e suas funcionalidades variadas, no contexto de integração eco-sistemas computacionais.

De seguida são apresentados esses exemplos.

5.1 Exemplo de um *workflow* aplicado a uma empresa

O primeiro exemplo baseia-se num caso real de um *workflow* na área de *business*. A Figura 5.1 apresenta um *workflow* usado no sistema de gestão de projectos da empresa *Quidgest*[1] que é executado sempre que existe uma oportunidade de negócio na área de gestão documental. Durante a execução do *workflow* os vários intervenientes (clientes e colaboradores *Quidgest*) são notificados e interagem de forma a adjudicar e a fechar o projecto com sucesso.

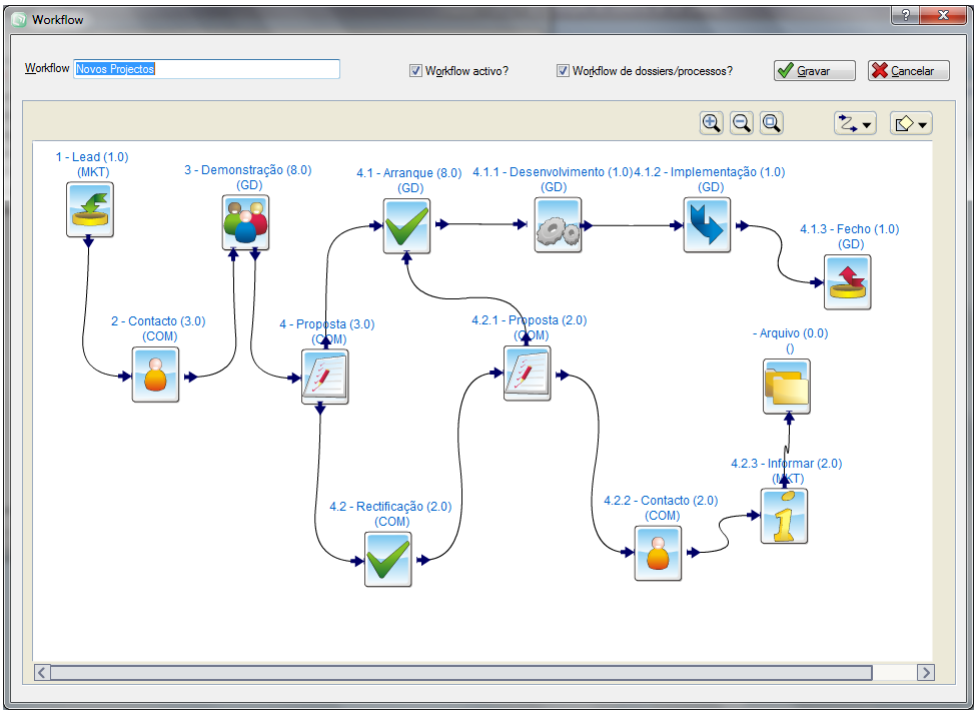


Figura 5.1: Exemplo "Novos Projectos" (passo 1)

A notação usada para este motor é específica da empresa, ou seja, as tarefas estão associadas a um funcionamento interno à aplicação, em particular , à noção de "encaminhamento de informação" entre áreas. Essa informação engloba metadados, documentos digitalizados específicos dos projectos, e é gerida por um sistema *CRM - Customer relationship management* suportado por uma base de dados. Como pode ser observado na Figura 5.1, os passos realizados entre áreas são os seguintes:

1. **Lead** - A primeira tarefa é executada quando existe uma nova *Lead* (oportunidade) para um novo projecto, i.e., quando qualquer colaborador da área de *Marketing (MKT)* percebe que existe uma nova oportunidade de negócio. O colaborador digita a informação obtida que a encaminha para o próximo passo. Neste caso, a informação é encaminhada para a área de *Comunicação Interna (COM)*.

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

2. **Contacto** - Os colaboradores da área *COM* têm o papel de enviar a informação à área de gestão documental.

3. **Demonstração** - Todos os colaboradores da área de *Gestão Documental (GD)* recebem a informação proveniente da área *COM* e verificam se podem marcar uma reunião de demonstração com o cliente. A execução só passa à próxima fase quando essa demonstração for realizada, e for inserida a informação obtida durante a demonstração (e.g. requisitos necessários, etc.)

4. **Proposta** - Com todos os dados inseridos no *CRM* é gerada uma proposta com base num *template* contendo os dados a enviar ao cliente. Nesta tarefa é esperada a resposta do cliente e a área *COM* encaminha-a para processamento na tarefa seguinte. Neste caso, existe uma bifurcação que depende da resposta do cliente: caso a resposta seja afirmativa, a informação é enviada para a área *GD*; caso contrário, a informação terá de ser rectificada:

   4.1) **Arranque** - A área *GD* analisa a informação (requisitos obtidos no cliente) e organiza a equipa responsável pelo projecto, iniciando a sequência de tarefas responsável por implementar o projecto:

      4.1.1 **Desenvolvimentos** - Este passo é meramente representativo do estado do desenvolvimento, i.e., enquanto o desenvolvimento da aplicação não for concluído, o fluxo de execução não passa desta fase; quando terminar, a execução avança para a implementação do sistema no cliente.

      4.2.2 **Implementação** - Esta fase representa o aguardar de uma resposta afirmativa de que o sistema foi instalado e entrou em produção isto é, que se encontra em total funcionamento.

      4.3.3 **Fecho** - Projecto é terminado e informação é guardada.

   4.2) **Rectificação** - A área *COM* analisa que informação (requisitos) está ainda em falta e notifica membros da *GD* sobre o que é preciso alterar na proposta (notificação é normalmente pessoal):

      4.2.1 **Proposta** - Com os dados alterados, é de novo gerada uma proposta que é enviada ao cliente, aguardando-se a sua resposta. Quando a resposta é afirmativa a execução continua no ponto 4.1. Caso contrário o projecto falha.

      4.2.2 **Contacto** - Como a proposta foi rejeitada, a área *COM* envia a informação ao *MKT*.

      4.2.3 **Informar** - O *MKT* analisa o porquê da recusa do projecto e informa todos os colaboradores envolvidos, arquivando posteriormente esses dados.

      4.2.4 **Arquivo** - Adiciona todos os dados a base de dados do *CRM*.

A notação de suporte para o motor de execução e para o editor destes *workflows* é

específica da *Quidgest*, o que implica que não existem os tipos de tarefas existentes na notação *BPMN 2.0*.

Assim, uma proposta de implementação deste *workflow* num *workflow BPMN 2.0* é apresentado na Figura 5.2. Este *workflow* serve de base à ilustração das vantagens de adicionar alguns padrões de estrutura e comportamento, tal como descrito nas seguintes secções. Os dados usados como suporte a este exemplo são gerados aleatoriamente, simulando-se uma situação real.

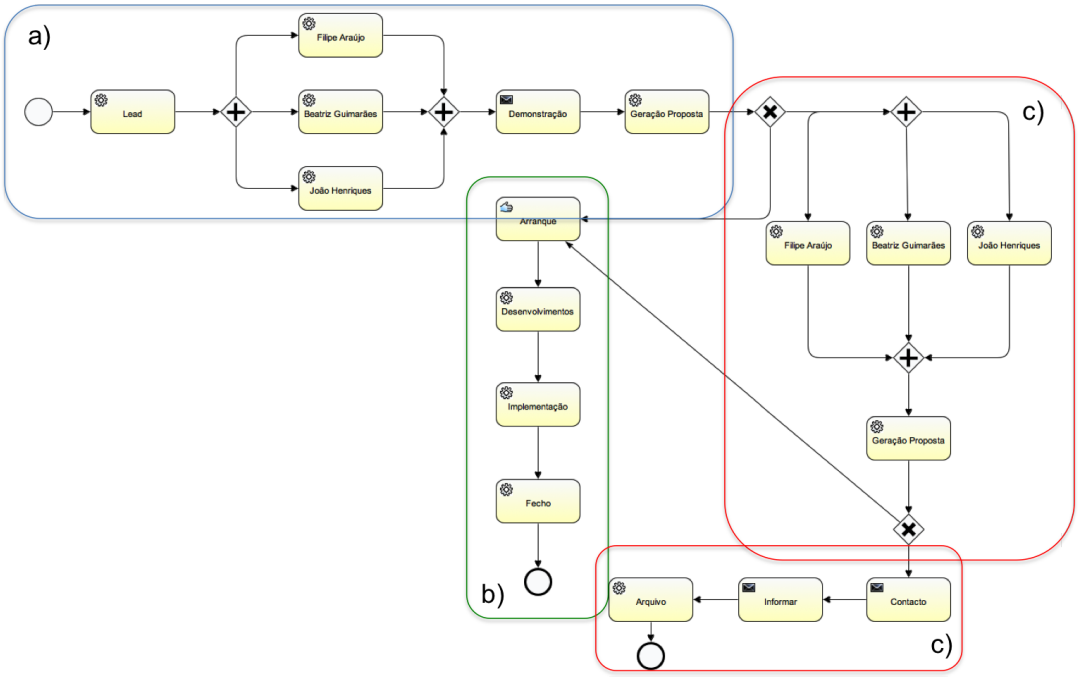


Figura 5.2: Exemplo "Novos Projectos" (*workflow BPMN 2.0* sem comportamentos)

5.1.1 *Workflow BPMN*

Por simplificação, a Figura 5.2 está dividida em três módulos, que são descritos de seguida, e ainda como são processados os *outputs* e *inputs* (Figura 5.4) durante a execução do *workflow*:

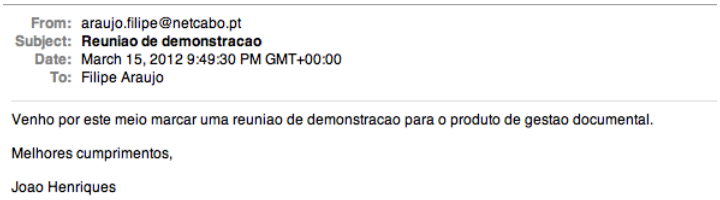
a) Fase inicial do *workflow*

Lead A informação da *lead* e a data de demonstração proposta pelo cliente são obtidos na base de dados do CRM.

Tarefas em paralelo Os colaboradores da área de gestão documental são solicitados para realizar essa demonstração. A informação da disponibilidade é obtida na base de dados *CRM* onde os utilizadores adicionaram as datas de ausência. Parte-se do pressuposto que pelo menos um deles tem disponibilidade para a demonstração (neste caso é o João Henriques). No sistema *CRM* o estado do colaborador, isto

é, se está de férias, está ausente, ou tem disponibilidade, tem mensagens *template* associadas (como apresentado no *output* da Figura 5.4).

Demonstração É enviado um email ao cliente a combinar a reunião de demonstração (o aspecto do *email* gerado é ilustrado na Figura 5.3).


From: araujo.filipe@netcabo.pt
Subject: Reuniao de demonstracao
Date: March 15, 2012 9:49:30 PM GMT+00:00
To: Filipe Araujo

Venho por este meio marcar uma reuniao de demonstracao para o produto de gestao documental.

Melhores cumprimentos,

Joao Henriques

Figura 5.3: Exemplo do aspecto do E-mail enviado no contexto deste exemplo

Geração proposta Na tarefa "Geração proposta", após a submissão dos dados da demonstração no *CRM*, é gerado um documento de proposta (geralmente através de um gerador de propostas) que é posteriormente enviado ao cliente. Ainda nesta tarefa, aguarda-se a resposta do cliente que é inserida no CRM pelos colaboradores, ou então através de uma *UI* oferecida ao cliente (e.g. *website*).

b) Cliente aceitou proposta

Arranque Caso a resposta seja aceite, o projecto arranca e é utilizada uma *manual task* que suporta a definição de passagem para a próxima fase.

Desenvolvimentos O desenvolvimento é iniciado, ficando uma *UI* (formulário) a aguardar o *feedback* dos *"developers"* quando o desenvolvimento terminar.

Implementação Esta fase é semelhante à anterior, no entanto, o *feedback* é dado após a aplicação ser instalada no cliente.

Fecho O *workflow* termina com sucesso e o projecto é fechado.

c) Cliente rejeitou proposta

Tarefas em paralelo Caso a proposta necessite de alterações, os colaboradores são de novo notificados com a informação fornecida pelo cliente; os intervenientes têm de inserir novos dados no CRM, relacionados com a correcção da proposta, e pelo menos um deles deve dar como terminada a edição dos dados.

Geração proposta Uma nova proposta é gerada para avaliação pelo cliente.

Arranque Caso a proposta corrigida tenha sido aceite, o fluxo de execução passa para o *workflow* definido em b).

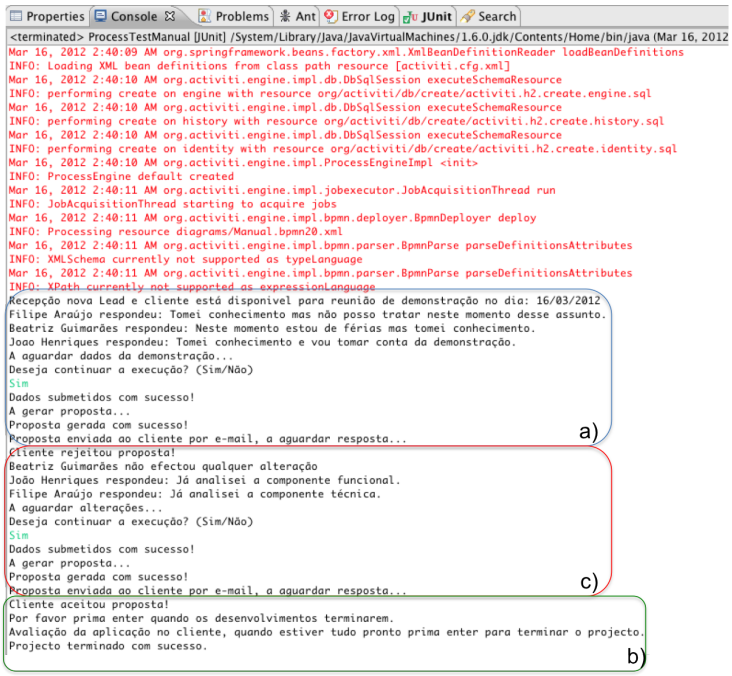


Figura 5.4: Exemplo "Novos Projectos" (*Output workflow* sem comportamentos)

Contacto/Informar/Arquivo Caso a proposta seja rejeitada, o projecto é dado como perdido e os colaboradores são notificados por email, e os dados são arquivados no CRM.

No entanto esta execução é restrita à definição inicial do *workflow*, não existindo modificação em tempo de execução. Por exemplo querendo explicitar que colaboradores podem tratar da elaboração da proposta e sua correcção (tarefas "2 - Contacto" e "4.2 Rectificação" no *workflow* da Figura 5.1), é necessário definir quantos são, e este número não pode ser alterado em tempo de execução. Assim, descreve-se na próxima sub-secção como se pode usar a estruturação e reconfigurações dinâmicas com base em padrões, para introduzir alterações no *workflow* em tempo de execução.

5.1.1.1 *Workflow BPMN* com padrões de estrutura e comportamento

Para ilustrar as funcionalidades do nosso protótipo, o *workflow* anterior foi alterado contendo agora um caminho de execução para projectos simples e outro para projectos complexos. A execução para os projectos simples corresponde ao que foi apresentado anteriormente. A execução de projectos complexos considera que os membros da equipa de gestão documental envolvidos podem variar em tempo de execução do *workflow*. Neste caso, é suposto existir um colaborador responsável por desencadear as reconfigurações dinâmicas necessárias.

O *workflow* alterado é apresentado na Figura 5.5, onde foram incluídos dois *workflows* com estruturação baseada em padrões nas acções correspondentes às tarefas 2 e 4.2 do

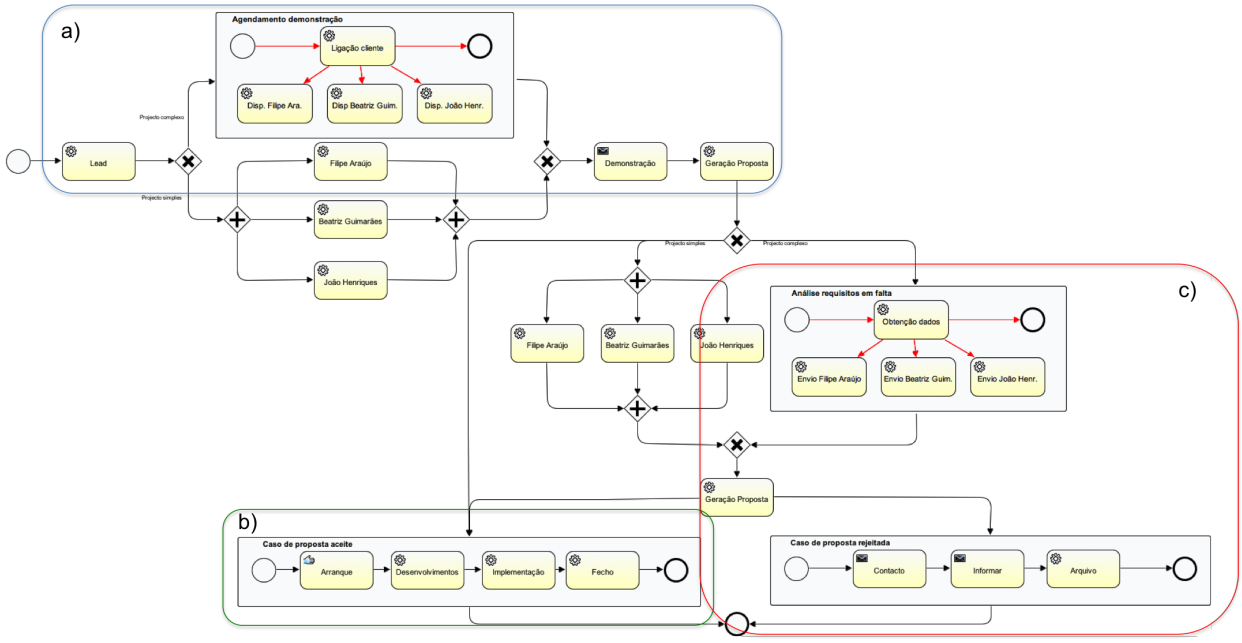


Figura 5.5: Exemplo "Novos Projectos" (*workflow BPMN 2.0* com comportamentos)

workflow original. As alterações ao *workflow* de processamento de projectos complexos, são descritas no contextos dos módulos **a)**,**b)** e **c)**, como apresentado na Figura 5.5.

a) Fase inicial do *workflow*

Lead A informação da *lead* e a data de demonstração proposta pelo cliente são obtidos na base de dados do CRM.

Sub-processo "Agendamento demonstração" Um sub-processo com estrutura em estrela e comportamento publicador/subscritor é executado, existindo de inicio três colaboradores que subscrevem notificações de participação em novos projectos. Para tal definem o período de tempo em que estão disponíveis: o Filipe Araújo subscreve ter disponibilidade a partir de datas superiores a 19/03/2012; a Beatriz para datas superiores ou iguais a 18/03/2012; em relação ao João Henriques subscreve para datas superiores a dia 20/03/2012. A parametrização na linha de fluxo do Filipe Araújo foi realizada da seguinte forma, sendo as outras semelhantes:

Figura 5.6: Exemplo "Novos Projectos" (Parametrização subscrição "calendário")

Para efeitos de demonstração, a tarefa "ligação cliente"cria uma ligação *TCP* com o cliente que, através de uma consola, pode alterar a data inicial. Por exemplo, suponhamos que o cliente digitou que tem disponibilidade para dia 18/03/2012. Essa informação é gravada no CRM resultando na notificação dos colaboradores que subscreveram um período de tempo que inclua o definido pelo cliente. O resultado é o apresentado na Figura 5.7. Entretanto, o cliente verificou que a data que

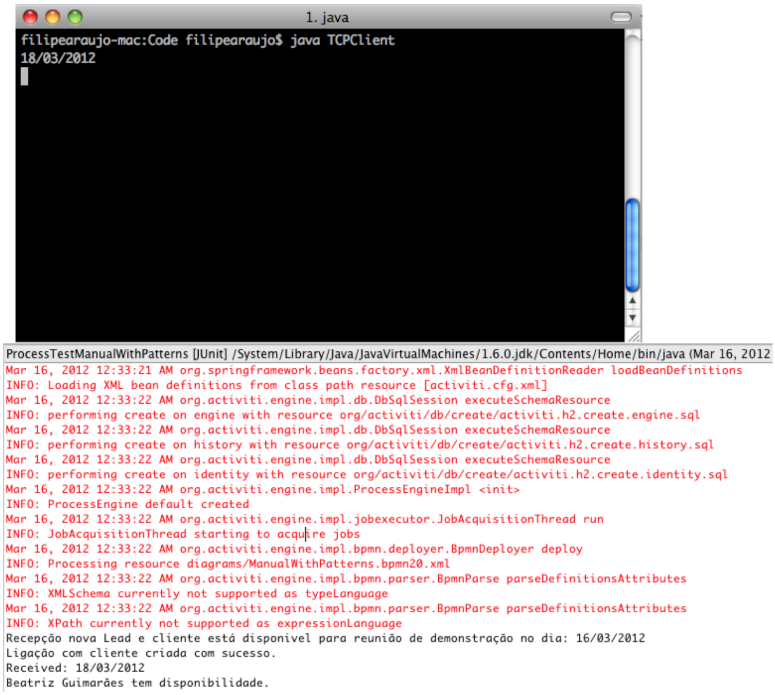


Figura 5.7: Exemplo "Novos Projectos" (Interacção cliente)

inseriu não era a correcta e mudou para dia 30/03/2012. O cenário altera-se então para o apresentado na Figura 5.8.

Suponhamos agora um cenário com novas restrições, dado que o cliente altera a marcação para a data inicial, a qual é impossível para todos os colaboradores envolvidos, isto é 16/03/2012. O administrador verifica que a data se aproxima e que nenhum dos colaboradores pode realizar a demonstração. Dinamicamente insere um colaborador que pode realizar a demonstração, ou seja, insere uma nova tarefa com o comando de inserção. O *workflow* e o *output* ficam com o aspecto apresentado na Figura 5.9. Com esta reconfiguração permite-se assim a adição de novos colaboradores em tempo real.

Resumindo, o sub-processo "Agendamento demonstração" no módulo a) do *workflow*, que é suportada pelo padrão topológico estrela com comportamento publicador/subscritor, está assim continuamente em execução, permitindo a alteração dinâmica da data da apresentação, caso necessário. Para terminar este sub-processo, e permitir a continuação da execução pelas tarefas seguintes do *workflow*, existem duas opções: expira o tempo de vida do padrão (*TTL* pré-definido); a terminação

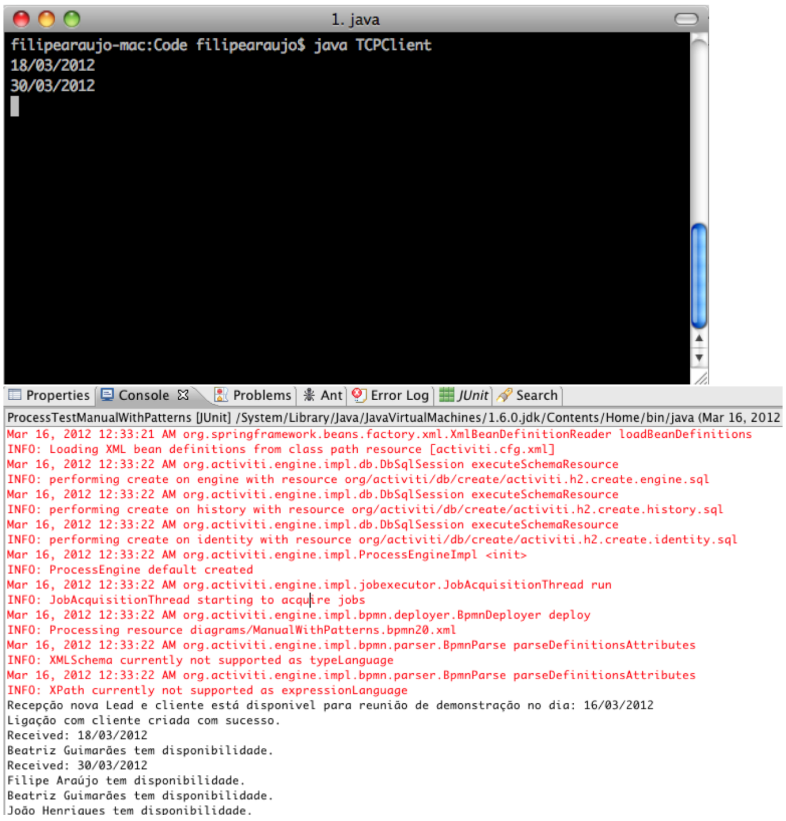


Figura 5.8: Exemplo ”Novos Projectos” (Interacção com cliente)

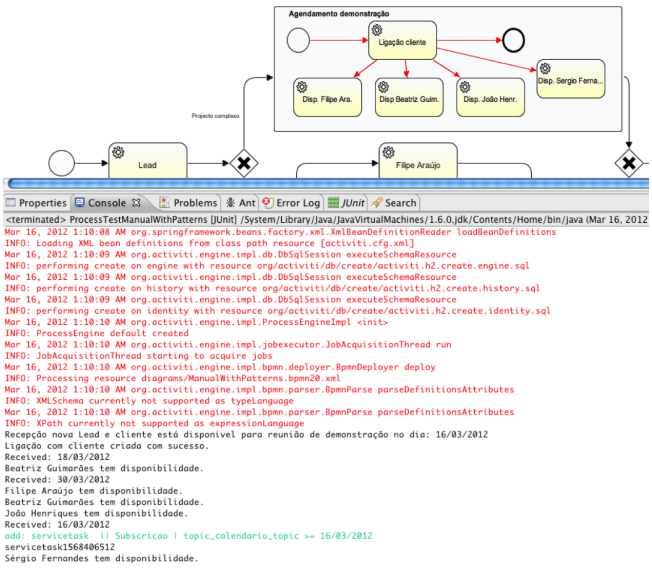


Figura 5.9: Exemplo ”Novos Projectos” (Interacção com cliente (Reconfiguração))

é realizada a pedido. No caso deste exemplo particular, o administrador termina a execução logo que tem um colaborador disponível.

O resto das tarefas têm a mesma execução que o *workflow* sem comportamentos.

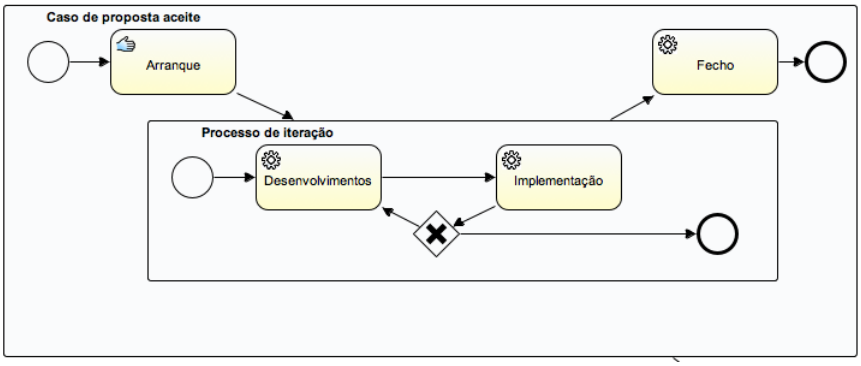


Figura 5.10: Exemplo "Novos Projectos" (*Pipeline* com anel)

b) Cliente aceitou proposta A diferença neste ponto, em relação ao *workflow* da Figura 5.2, é que as suas tarefas estão inseridas num padrão *pipeline* possibilitando a sua alteração Por exemplo, com a reconfiguração em tempo estático, é possível adicionar tarefas à fase de desenvolvimento no fim do *pipeline*. Em alternativa, é possível incluir como etapa intermédia um anel representando a associação entre as tarefas "desenvolvimento" e "implementação" (ver Figura 5.10). Este anel captura explicitamente um ciclo de desenvolvimento incremental da solução proposta. Tal explicita um processo iterativo entre os colaboradores e o cliente no processo de criação de um sistema que geralmente é o que acontece.

c) Cliente rejeitou a proposta

O cliente rejeitou a proposta e a informação que é necessário rever foi catalogada em três categorias, funcional, técnica e de administração, o que permite relacioná-la com o perfil do colaborador que o poderá alterar.

Sub-processo "Análise requisitos em falta" Neste sub-processo seguinte, na Figura 5.5, um padrão estrela combinado com comportamento publicador/subscritor é executado, existindo de inicio três colaboradores. Cada um subscreve possíveis notificações relacionadas com o tipo de alterações a fazer na proposta. Por exemplo, o Filipe Araújo subscreveu notificações sobre a informação técnica a alterar (UML, arquitecturas apresentadas, etc); a Beatriz subscreveu a informação de administração (preços, cronogramas, etc) a alterar; o João Henriques subscreveu notificações sobre alterações à informação funcional, por exemplo, alterações na área de negócio, neste caso, de gestão documental. A parametrização na linha de fluxo do Filipe Araújo é realizada tal como ilustrado na Figura 5.11.

Estas subscrições permitem assim, filtrar e direccionar a informação para os colaboradores com mais competências. A tarefa "obtenção dados", situada no núcleo

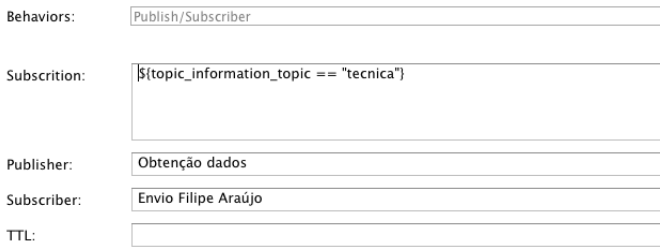


Figura 5.11: Exemplo "Novos Projectos" (Parametrização subscrição "information")

da estrela que suporta o sub-processo "Análise requisitos em falta" obtém a informação catalogada na base de dados CRM e envia-a de acordo com o tópico subscrito. Para ilustrar essa funcionalidade, as tarefas de envio criam uma ligação *TCP* e enviam os tópicos subscritos para uma consola que o utilizador lê. Neste caso a consola 1 é para o Filipe Araújo que subscreveu informação técnica e a consola 2 é para a Beatriz (informação administrativa), tal como mostra a Figura 5.12.

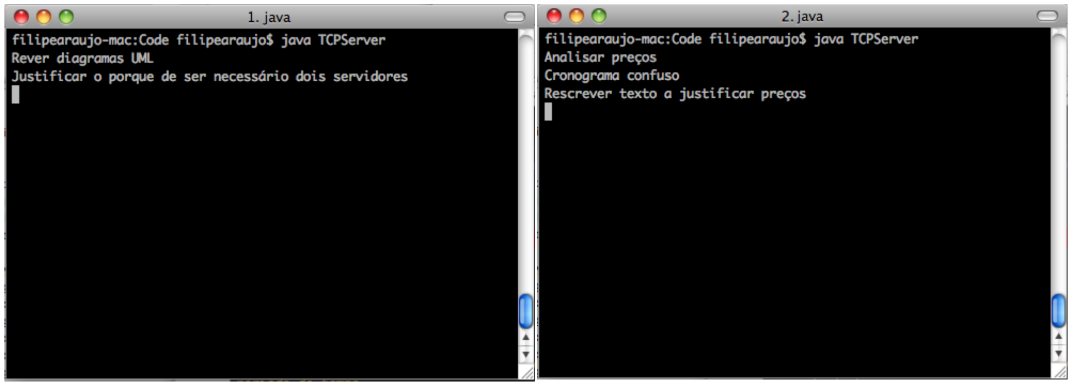


Figura 5.12: Exemplo "Novos Projectos" (Envio de informação)

Suponhamos agora que o prazo de entrega da proposta a alterar está a terminar, sem que todas as alterações tenham sido realizadas. Sabendo que os colaboradores também podem tratar de outros tópicos para além do subscrito, o administrador pode mudar o comportamento para *streaming* como apresentado na Figura 5.13, para que todos os utilizadores recebem toda a informação que falta tratar. Por exemplo a consola 2 que só mostrava a informação do tipo "Administração", mostra agora notificações sobre informação técnica que falta alterar (ver Figura 5.14).

Outras reconfigurações úteis seriam, por exemplo, adicionar uma nova tarefa, com uma consola para o próprio supervisor do projecto, com subscrição de todos os tópicos, de modo a que ele possa saber qual a informação em falta.

Finalmente, o tempo de vida para a execução do padrão pode ser parametrizado com o prazo limite para as alterações do projecto.

Sub-processo "Caso proposta rejeitada" O fluxo de execução no caso da proposta ser

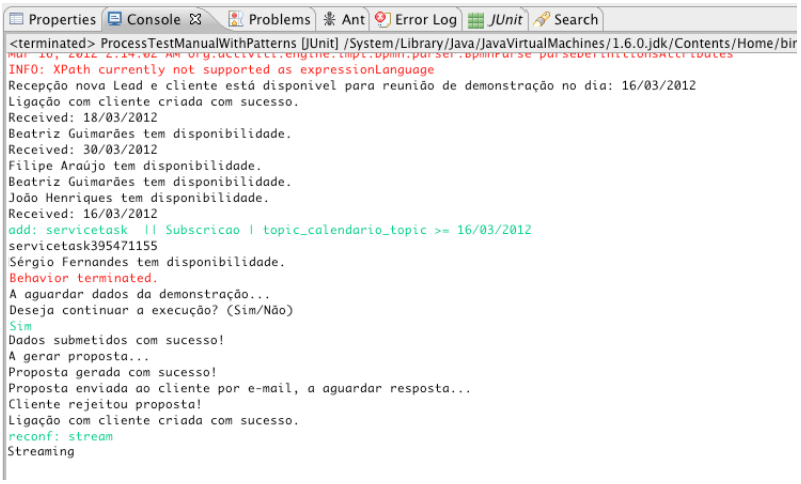


Figura 5.13: Exemplo ”Novos Projectos” (Reconfiguração comportamento)

Figura 5.14: Exemplo ”Novos Projectos” (Resultados reconfiguração)

rejeitada, também passou a ser suportado por um padrão estrutural *pipeline* de forma a simplificar futuras composições/alterações.

Com estes exemplos, pretendemos mostrar a vantagem da reconfiguração dinâmica no contexto de padrões de estrutura e comportamento, quer pela possibilidade de adicionar novas tarefas aos padrões (reconfigurações da estrutura), quer pela vantagem de se poder alterar o padrão de comportamento que define as dependências entre as tarefas em tempo de execução.

5.2 Exemplo *DDDAS*

Este exemplo ilustra uma possível aplicação do nosso protótipo no contexto de sistemas aplicacionais orientados a dados dinâmicos *(Dynamic data driven applications systems (DDDAS))*[Dar04]. Em geral, o conceito *DDDAS* captura a necessidade de incorporar dados dinamicamente numa aplicação de simulação em execução, bem como a possibilidade de a própria aplicação parametrizar o modo como essa recolha de dados é feita. Nesta secção é ilustrado um exemplo que apresenta apenas algumas características desde tipo de aplicações, sendo que na próxima secção é apresentada uma extensão a este exemplo.

O exemplo escolhido é um caso particular de uma simulação no contexto de monitorização e análise de cheias urbanas (*Urban Flooding Analysis and Monitoring (UFAM))*.

O exemplo descreve alguns cenários possíveis para uma zona crítica que, recorrentemente, é sujeita a situações de cheias. Para um sistema *UFAM*, a informação meteorológica (e.g. valores de humidade ou precipitação) torna-se fulcral para a elaboração de dados estatísticos por forma a definir a probabilidade de cheias em locais historicamente mais afectados. Neste contexto, as autoridade locais das zonas problemáticas beneficiam da subscrição de serviços que disponibilizem informações meteorológicas das várias zonas criticas. Para mais, a forma como esses dados são entregues pode variar de acordo com a possibilidade de existir perigo de uma inundação ou não. Por exemplo, uma notificação periódica dos valores (de humidade, etc) é, à partida, suficiente numa situação de seca prolongada. Já no caso de os valores indicarem uma probabilidade significativa de ocorrência de chuva, é importante obter um fluxo continuo de dados recolhidos na zona em perigo.

Contudo, esses serviços meteorológicos podem ser oferecidos com um custo financeiro, tanto sobre o número de ligações para recolha de diferentes tipos de dados (humidade, precipitação, etc), como sobre a forma como esses dados são entregues (e.g. entrega tendo como base modelos de interacção publicador/subscritor, *streaming*, etc). Assim, o custo pode ter de ser tido em conta na escolha do modelo de interacção com o serviço, para além da possibilidade de se verificar uma inundação.

Estrutura Geral Como forma de representar um sistema *UFAM* com estas características, foi construído um *workflow* que permite reconfigurações dinâmicas de acordo com o

evoluir de um possível cenário de inundação numa zona historicamente crítica. O *workflow* definido captura, não só a recepção de dados da zona crítica, como também a sua disponibilização a diversos elementos da autoridade local, bem como a sua salvaguarda num repositório para posterior avaliação. Assim, o *workflow* base é composto por um padrão estrutural em estrela, em que o nó núcleo representa a tarefa responsável por obtenção de valores meteorológicos (humidade, precipitação, etc) da zona crítica; os nós satélites representam as tarefas que utilizam esses valores e para objectivos diferentes, como por exemplo, disponibilizando a informação numa *interface*, ou salvaguardando a informação numa base de dados para fins de análise de histórico, etc. Deste modo, a mesma informação (recolhida no núcleo da estrela) pode ser entregue a mais do que uma tarefa (os satélites) para processamento diversificado ou por questões de redundância (e.g. várias *"Data Storage"*).

Para além disso, o *workflow* de suporte é especificado tendo em consideração que, para a autoridade local existem três situações possíveis ao longo do ano: uma situação normal, por exemplo no verão em que os valores da humidade variam pouco e a precipitação é quase nula; uma situação de prevenção, por exemplo o início da estação de Inverno onde os valores são imprevisíveis; e uma situação de emergência, por exemplo quando ocorre chuva intensa (precipitação elevada), recorrente nessas zonas. Nas próximas sub-secções descreve-se a construção incremental do *workflow* de base e reconfigurações adicionais, de acordo com o evoluir da situação.

De referir que, apesar de não ser ilustrado neste exemplo, é possível adicionar um evento *Error Boundary Event* ao sub-processo. Este evento serve para quando ocorre erros/excepções no *sub-processo* a execução fluí para um caminho específico (por exemplo notificar o administrador do erro).

Situação normal Por forma a guardar o histórico dos valores recolhidos na zona crítica ao longo do ano, a autoridade local subscreve um serviço que disponibiliza os valores da humidade recolhidos na zona. O modelo de interacção por omissão é o publicador/-subscritor, dado que é suficiente para situações normais. Este modelo implica um custo menor associado ao serviço, dado que os valores recolhidos são comunicados esporadicamente (com uma taxa de entrega que poderá ser alterada, aumentando o seu custo). O modelo de interacção publicador/subscritor estará em uso tipicamente durante a estação do Verão. Na Primavera e Outono, poderá ser também definido como modelo de interacção por omissão, perante situações meteorológicas consideradas normais.

Em termos da composição do *workflow* propriamente dito, no cenário "normal", são definidas apenas duas tarefas, como ilustrado Figura 5.15. No núcleo da estrela encontra-se uma tarefa subscritora da fonte de dados (e.g. valores de humidade). Estes dados recolhidos são enviados para uma tarefa (satélite) que, garante a sua salvaguarda num repositório para análise posterior. Para esta fase foram criado *web services* de teste que geram valores aleatórios entre 1-35% durante um certo período de tempo (1 min, para teste); esses valores mudam para o intervalo 35-100% depois desse período. Pretende-se

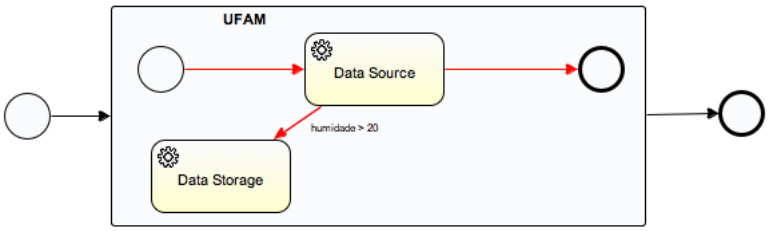


Figura 5.15: Exemplo *UFAM* (passo 1)

desta forma simular valores típicos de inverno e criar um cenário o mais realista possível. Como se observa na Figura 5.15, neste caso, a autoridade local subscreveu valores da humidade superiores a 20%. Um exemplo de uma amostra obtida pela tarefa publicadora e a tarefa subscritora é a seguinte:

Listing 5.1: Valores obtidos na tarefa fonte (publicadora)

```
Publicador valor humidade: 15%
Publicador valor humidade: 10%
Publicador valor humidade: 31%
Publicador valor humidade: 34%
Publicador valor humidade: 26%
Publicador valor humidade: 16%
Publicador valor humidade: 7%
Publicador valor humidade: 2%
Publicador valor humidade: 34%
Publicador valor humidade: 3%
Publicador valor humidade: 2%
Publicador valor humidade: 33%
Publicador valor humidade: 3%
Publicador valor humidade: 16%
Publicador valor humidade: 14%
Publicador valor humidade: 5%
Publicador valor humidade: 8%
Publicador valor humidade: 33%
```

Listing 5.2: Valores obtidos na tarefa satélite (subscritora)

```
Valor humidade obtido: 31%
Valor humidade obtido: 34%
Valor humidade obtido: 26%
Valor humidade obtido: 34%
Valor humidade obtido: 33%
Valor humidade obtido: 33%
```

Como se pode verificar pelos resultados obtidos, ocorre o comportamento publicador/subscritor, i.e., tendo a subscrição superior a 20.

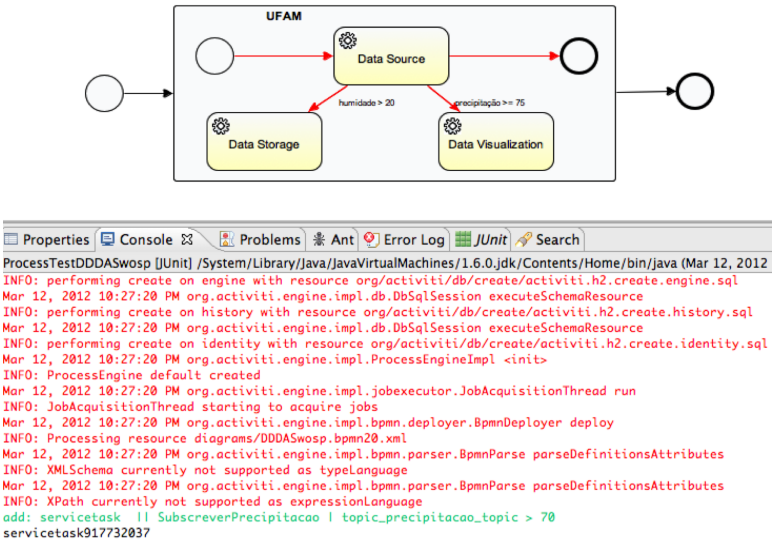


Figura 5.16: Exemplo *UFAM* (passo 2)

Situação de prevenção Supondo agora a execução do *workflow* já durante o período de Inverno, a humidade aumenta e a probabilidade de a precipitação a ser elevada também aumenta. A autoridade local define que o cenário é agora de "prevenção" e, dinamicamente, é feita a subscrição de outra fonte de dados, para recolha de valores de precipitação na mesma zona. A simulação desta situação é suportada com o acesso a um *web service* que gera valores aleatórios entre 55 e 85 mm, representando a média de densidade de precipitação numa cidade como Veneza [2], onde ocorrem imensos problemas com cheias. Os novos dados recolhidos são entregues a uma nova tarefa, também ela criada dinamicamente, correspondendo a mais um satélite da topologia estrela de base (ver topo da Figura 5.16). Esta nova tarefa corresponde a uma *interface* de utilizador (*UI*), onde os dados obtidos são usados para notificar membros da autoridade local, considerando o valor da densidade de precipitação subscrito. Neste exemplo, a nova tarefa subscreve valores de precipitação superiores a 70 mm, permitindo avaliar se a situação pode evoluir para uma situação de emergência. O comando de adição da tarefa é invocado em tempo de execução do *workflow*, resultando na sua reconfiguração dinâmica. Em particular, é adicionado um novo satélite, tal como descrito e, após invocação do botão de *update*, aparece a estrela já reconfigurada, como ilustrado na Figura 5.16. O botão *update* encontra-se na área de parametrizações do diagrama principal, descrito na secção 4.3.

A classe "SubscreverPrecipitacao", incluída no comando, serve para criar uma ligação TCP com um servidor, sendo a informação disponibilizada tal como ilustra a Figura 5.17.

Este é um exemplo da versatilidade do nosso protótipo: em tempo de execução é possível adicionar novas tarefas que trabalham sobre valores obtidos, permitindo a sua análise de formas diversas. Neste exemplo, e como ilustrado na Figura 5.16, os dados podem ser mostrados numa *UI* a utilizadores da autoridade local, mas podem também

[2] http://www.eldoradocountyweather.com/climate/europe/italy/Venice.html

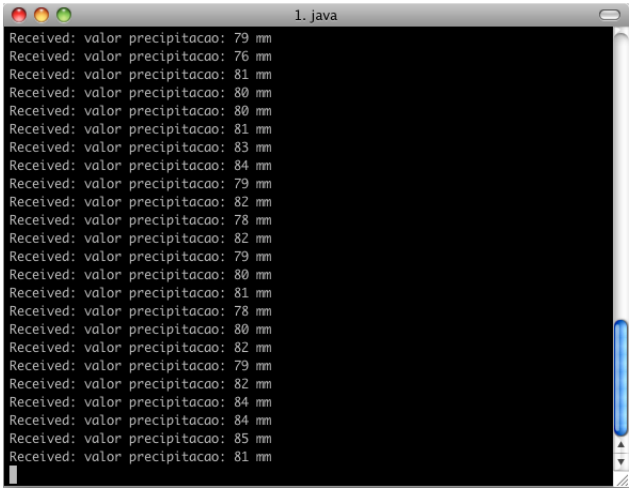


Figura 5.17: Exemplo *UFAM* (passo 3)

ser guardados num repositório. Estes dados podem ser analisados tanto em *"real-time"* como posteriormente.

Situação de emergência Suponhamos agora que os valores da precipitação estão bastante elevados para zonas sensíveis a episódios de inundações, tal como se pode verificar na Figura 5.17. Neste caso, a autoridade local necessita de receber, em tempo real, uma maior número de valores de precipitação de modo a ter um conhecimento mais preciso da situação. Este cenário implica que a situação mude para uma situação de "emergência".

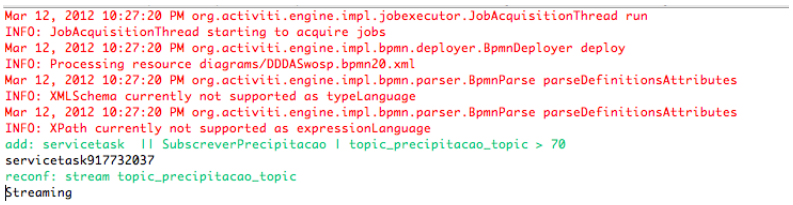


Figura 5.18: Exemplo *UFAM* (passo 4)

A melhor forma de, obter uma sequência de valores, em tempo quase-real, é reconfigurar a aplicação para um comportamento *Streaming*. Este padrão de comportamento é usado na interacção *"Data source"* (núcleo da estrela) com o *"Data storage"* e *"Data visualization"* (satélites), e é activado com o comando descrito na Figura 5.18. Note-se que para a entrega de um fluxo contínuo de dados, é necessário que o *"Data source"* esteja a interrogar continuamente o *web service* que providencia esses dados. Na secção 5.3 será descrita uma forma de evitar essa interrogação contínua, bastando para isso que a tarefa *"Data source"* seja cliente de uma sessão que garante um modelo de interacção *Streaming* com uma ou mais fontes de dados.

No caso deste exemplo, e para gerir custos, é escolhida unicamente a subscrição de dados em tempo quase-real para a precipitação.

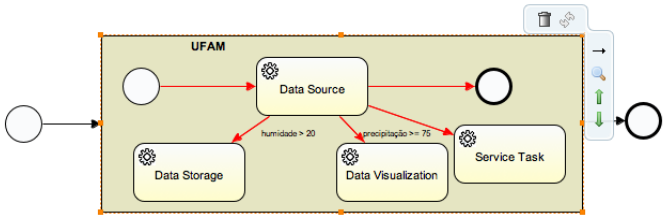


Figura 5.19: Exemplo *UFAM* (passo 5)

Um nova reconfiguração possível, é a adição de uma nova tarefa que subscreve a recepção de dados referente ao nível das águas, que tanto pode ser realizada em tempo dinâmico, como estático. A Figura 5.19 ilustra o caso de uma reconfiguração em tempo estático (i.e. em que a execução terá sido terminada pela autoridade local), bastando para isso pressionar o botão *increase*. A nova tarefa, tanto pode ser instanciada com um novo *"Data storage"*, como com uma instância de visualização de dados.

As reconfigurações apresentadas nas três situações descritas, para além de ajudarem a autoridade local a planear como actuar sobre zonas críticas, ajudam a reduzir custos em épocas em que é menor a probabilidade de ocorrência de inundações. Por exemplo no Verão só se acede a uma fonte de dados e com o modelo publicador/subscritor podendo este modelo ser alterado para *streaming* em situações críticas; em qualquer alturas, as tarefas consumidoras dos dados recolhidos na zona (os satélites da estrela) podem ser também alteradas.

A próxima sub-secção descreve novas configurações no contexto de aplicações *UFAM*.

5.2.1 Simulação para previsão de inundações

Como complemento da configuração inicial, descreve-se agora a criação de um *workflow* de suporte à simulação da evolução de possíveis cenários de inundação numa dada localidade, com base em dados recolhidos em tempo quase-real (*online data*).

Para isso é utilizado um padrão estrutural *pipeline* combinado com o comportamento produtor/consumidor, em que a primeira tarefa consiste na obtenção de dados de precipitação referentes a uma localidade particular. A segunda etapa do *pipeline* consiste numa aplicação de simulação que permite gerar cenários de possíveis inundações, face aos valores recolhidos no terreno (e recebidos no contexto da primeira etapa do *pipeline*). No exemplo criado a protecção civil pretende simular cenários de perigo de inundação na localidade de *Chioggia* (localidade de Veneza), tal como ilustrado na Figura 5.20.

Na primeira etapa, os dados meteorológicos são obtidos através de um *web service* que contém um método que recebe por argumento o nome da localidade e devolve os dados. O *web service* foi criado por nós para testes com geração de valores aleatórios. O valor máximo varia por localidade, por exemplo, para a localidade *Chioggia* a precipitação varia entre 0 e 99mm. A tarefa "simulator" por sua vez, recebe os valores da precipitação

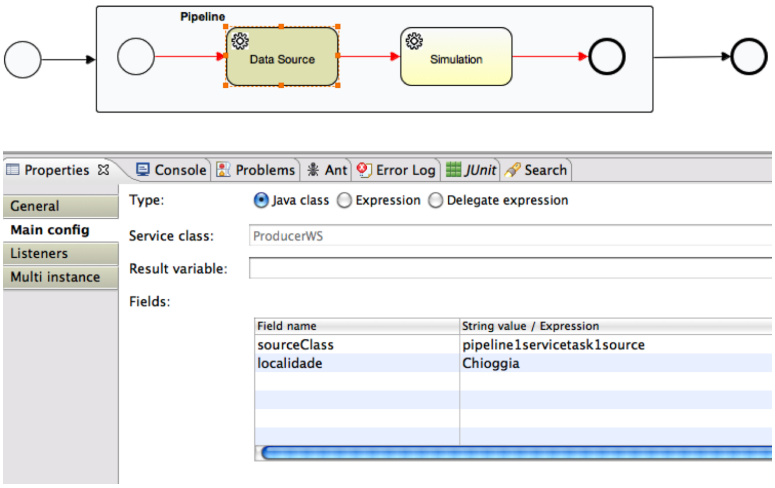


Figura 5.20: Exemplo *UFAM* (exemplo *pipeline*)

e, acedendo ao histórico da localidade, a simulação avalia a probabilidade de ocorrer inundações nesse contexto. Caso essa probabilidade seja significativa, gera avisos de situação de perigo, para essa localidade.

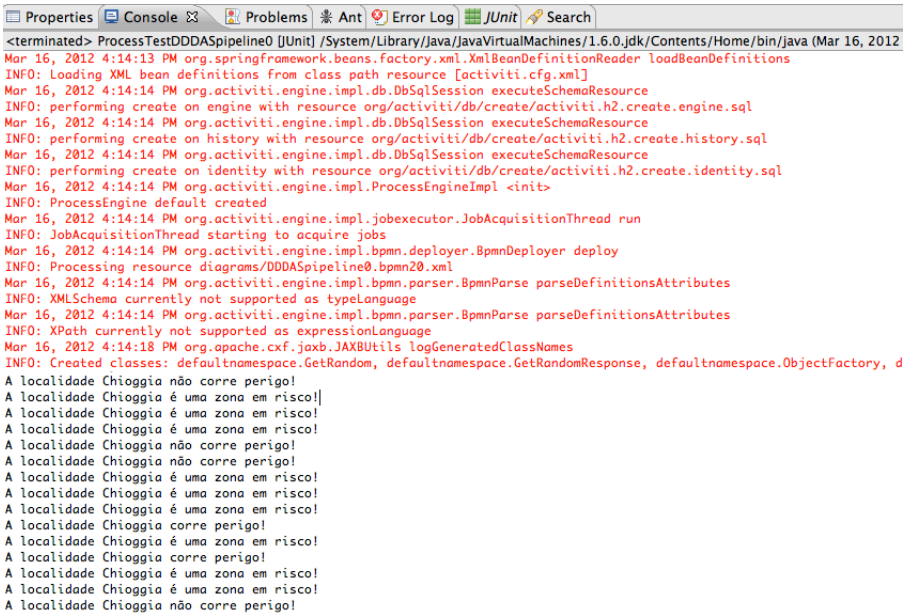


Figura 5.21: Exemplo *UFAM* (exemplo *output* da execução do *pipeline*)

Neste exemplo, a tarefa "simulator" avalia o valor da precipitação e retorna um *output* com possíveis cenários, por exemplo, caso esse valor seja entre 0 e 33 não existe perigo, caso seja entre 33 e 66 a zona está em risco, e entre 66 e 99 perigo é eminente. A Figura 5.21 ilustra um exemplo de execução. Por exemplo, a partir de uma determinada altura verifica-se que a localidade está em risco de ocorrer uma inundação. Neste caso, os serviços de protecção civil/autoridade local, podem ter necessidade que a simulação

seja mais precisa, porque uma única fonte de dados pode não ser suficiente para obter cenários o mais precisos e realistas possíveis. O chover muito não implica necessariamente casos de perigo. Assim é necessário que a informação geográfica da zona também seja considerada. Por exemplo, se existem barragens próximas da zona crítica, ou cursos de água em vias de transbordar os seus limites.

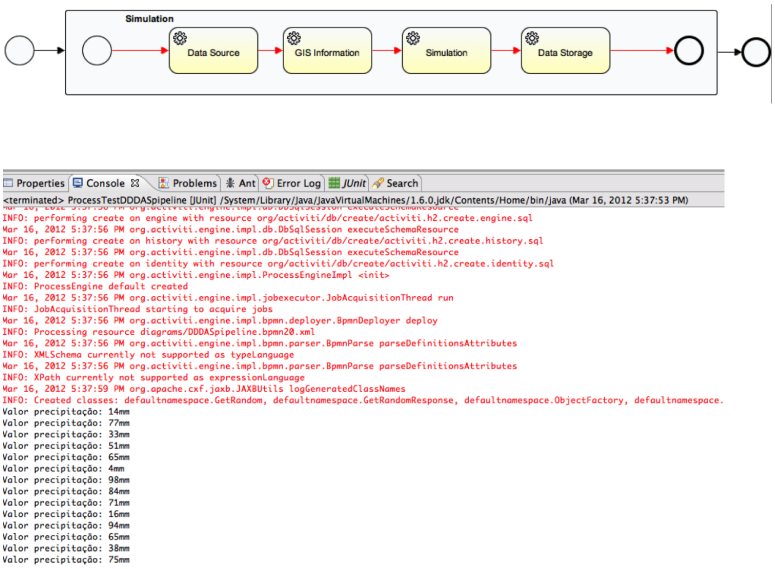


Figura 5.22: Exemplo *UFAM* (*pipeline* com nova fonte)

Adicionalmente, a protecção civil pretende que a informação gerada pelo simulador seja gravada numa base de dados, por forma a garantir a persistência dos dados para fins de histórico. Para o nosso exemplo essa tarefa guarda os dados num ficheiro.

Tal como mostra a Figura 5.22, o acesso a uma nova fonte de dados é realizada pela tarefa incluída na segunda etapa do *pipeline*, representando informação geográfica obtida num sistema *GIS*. No nosso exemplo, essa informação foi criada por nós, simulando possíveis condicionamentos para as diferentes localidades. Foi atribuído um peso a condições existentes no terreno, em termos da sua implicação num possível agravamento de uma situação de inundação. Por exemplo, a existência de rios com um elevado caudal, na zona afectada, aumenta significativamente a probabilidade de ocorrência de uma inundação, ou o seu agravamento, caso já exista inundação na zona em análise, pelo que tem um grande peso na formula subjacente à simulação. Outros factores podem ser tidos em consideração, aqui por exemplo a situação é mais grave numa planície do que numa zona montanhosa.

Na terceira etapa do *pipeline* encontra-se a aplicação de simulação (Figura 5.22), que verifica a densidade da precipitação bem como os condicionamentos na zona que possam agravar a situação, devolvendo cenários possíveis. No nosso exemplo, são considerados os seguintes condicionamentos: ”terreno acidentado” com peso médio, ”zona montanhosa” com peso baixo dado que está em análise o perigo de inundação. Os resultados

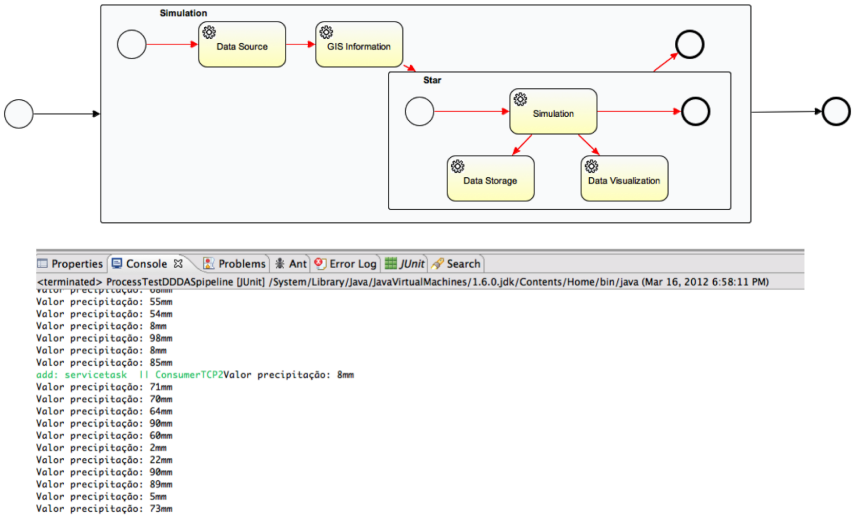


Figura 5.23: Exemplo *UFAM* (*pipeline* com novo consumidor e comando de reconfiguração)

obtidos são claramente diferentes visto que, apesar de ser uma zona com terreno acidentado, a informação de que é uma zona montanhosa ajuda a não existirem cenários de grande perigo de inundação. Esta nova formula de teste devolve valores entre 0 e 80 (não existe perigo), entre 80 e 130 (zona em risco) e entre 130 e 180 (perigo eminente):

Valor simulação: valor precipitação + média de pesos dos condicionamentos (são dados pesos numéricos, escala de 0 a 50)

No nosso exemplo "terreno acidentado" tem um peso de 25 e a "zona montanhosa" 10, logo a média é de 17,5. Os resultados como previsto são menos drásticos (lista 5.3) apesar das precipitações serem elevadas (ver Figura 5.22).

Listing 5.3: Valores gravados no ficheiro

```
1  Info Simulador: A localidade Chioggia nao corre perigo!
2  Info Simulador: A localidade Chioggia e uma zona em risco!
3  Info Simulador: A localidade Chioggia nao corre perigo!
4  Info Simulador: A localidade Chioggia nao corre perigo!
5  Info Simulador: A localidade Chioggia e uma zona em risco!
6  Info Simulador: A localidade Chioggia nao corre perigo!
7  Info Simulador: A localidade Chioggia e uma zona em risco!
8  Info Simulador: A localidade Chioggia e uma zona em risco!
9  Info Simulador: A localidade Chioggia e uma zona em risco!
10 Info Simulador: A localidade Chioggia nao corre perigo!
11 Info Simulador: A localidade Chioggia e uma zona em risco!
12 Info Simulador: A localidade Chioggia e uma zona em risco!
13 Info Simulador: A localidade Chioggia nao corre perigo!
14 Info Simulador: A localidade Chioggia e uma zona em risco!
```

Na última etapa do *pipeline* é guardada a informação gerada pela aplicação de simulação num repositório para análise posterior.

No contexto deste exemplo, uma das reconfigurações dinâmica possíveis é a adição de uma ou mais tarefas representando a visualização, em tempo real, dos dados produzidos pela simulação. Esta visualização é útil considerando que peritos no domínio de nacionalidades e em localizações diferentes, podem contribuir com os seus conhecimentos na avaliação de perigo da situação corrente. Assim, é feita uma reconfiguração tal que, as suas últimas etapas *"simulation"* e *"Data storage"* passam a fazer parte de uma estrela, nas posições de núcleo e satélite, respectivamente (Figura 5.23). Assim é possível adicionar dinamicamente novos satélites, representando consolas para peritos diferentes. Os mesmos dados, em vez de serem enviados apenas para a tarefa *"Data storage"*, são enviadas para todos os satélites da estrela.

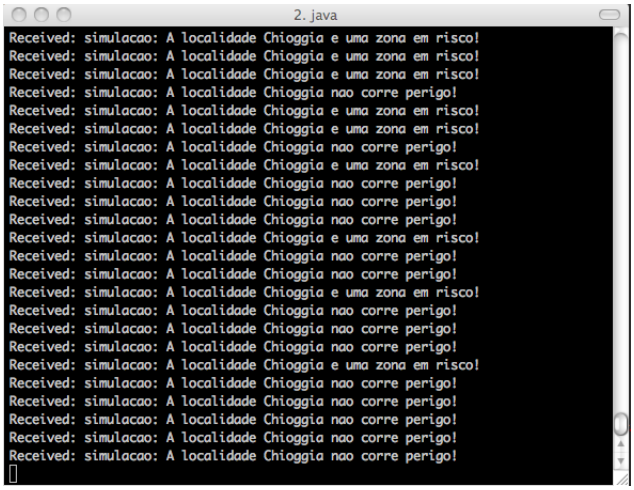


Figura 5.24: Exemplo *UFAM* (*Output* consola universidade)

Tal como no exemplo anterior, o administrador encarregue de gerir a execução deste *workflow*, executa o comando de adicionar uma tarefa para visualização dos dados numa consola. O *workflow* fica com a estruturação apresentada na Figura 5.23. Como se pode verificar a tarefa *"simulation"* é produtora para duas tarefas no contexto do padrão em estrela.

5.3 Integração Conceptual com Outros Sistemas de *Middleware*

Nesta secção descreve-se a integração conceptual deste trabalho com outros dois *middlewares* que permitem o acesso a serviços com estado no contexto de uma *sessão* e, através de modelos de interacção dinâmicos baseados no conceito de padrão de comportamento. Em ambos os casos, os padrões definidos correspondem aos mesmos padrões implementados nesta tese, tal como descrito no artigo [MCG12], sendo que no trabalho [Bap10] o *middleware* desenvolvido foi validado no contexto de redes de sensores sem fios, enquanto que o trabalho [Dom12] está ser adaptado para o contexto de *Cloud computing*.

A utilização da ferramenta Activiti estendida com padrões como *front end* para esses ambientes, pode ser vista sob duas dimensões. Por um lado, esta ferramenta de *workflow*

pode ser usada para parametrizar a recolha de dados e as reconfigurações dinâmicas possíveis nesses sistemas *middleware*. Por outro lado, dados produzidos nesse contexto, bem como reconfigurações dinâmicas que aí ocorram, podem ser usados para desencadear reconfigurações dinâmicas na ferramenta de *workflow*. A descrição da integração com os dois sistemas *middleware* é apenas conceptual, dado que não foi possível, no tempo desta tese, proceder à validação das acções necessárias à integração dos sistemas.

A próxima sub-secção descreve resumidamente o conceito base de *Sessão*, seguindo-se a descrição da arquitectura do sistema integrado que se pretende disponibilizar num futuro próximo. A sub-secção 5.2 descreve um exemplo da aplicabilidade desse sistema integrado.

5.3.1 O conceito de Sessão

O conceito *Sessão* representa o contexto da interacção entre um conjunto de utilizadores e um serviço, tendo como base modelos de interacção dinâmicos. A sua interface permite o acesso e a agregação de recursos com estado, bem como aos mecanismos de reconfiguração dinâmicas possíveis no contexto da sessão. Por exemplo, a possibilidade de se poder escolher, e alterar em tempo de execução, o modelo de interacção usado no contexto de uma sessão, permite obter fluxos de dados com diferentes qualidades de serviço. Assim, uma sessão inclui:

1. A identificação do serviço acedido, a identificação dos clientes pertencentes à sessão e o modelo de interacção usado, num dado momento. O modelo de interacção captura as dependências em termos de fluxo de dados e de controlo entre o serviço e os vários clientes, e é comum a todos eles.

2. Um identificador único usado por novos utilizadores quando se pretendem juntar à sessão. Existe a noção de *dono da sessão* que corresponde ao utilizador mais antigo. Apenas este pode, explicitamente, desencadear reconfigurações dinâmicas bem como terminar a sessão.

3. A definição do tempo de vida da sessão, sendo todos os clientes notificados quando esse tempo expira e, todos os dados da sessão destruídos. Sendo o tempo de vida ilimitado, cabe ao dono da sessão terminar a sessão.

4. Mecanismos de adaptação dinâmica com as seguintes características:

   a) Reconfiguração dinâmica dos modelos de interacção com base no contexto, o que pode incluir:

      i) O contexto do utilizador do serviço. Por exemplo uma autonomia reduzida no dispositivo móvel utilizado pelo cliente pode desencadear uma substituição do modelo de interacção para poupar energia, tal como de *streaming* para publicador/subscritor.

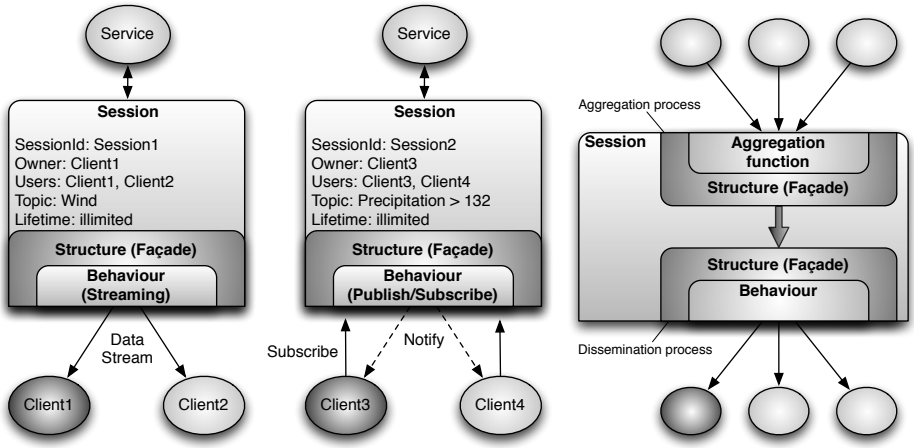


Figura 5.25: Exemplo de sessões: a) sessão com modelo de interacção *streaming*; b) sessão com modelo de interacção publicador/subscritor; c) sessão de agregação.

ii) O meio de interacção entre utilizador e serviço. Por exemplo, no caso de uma baixa qualidade de serviço da rede de comunicação, pode ser conveniente suspender um modelo interacção streaming enquanto essas condições persistirem, e usar ou um modelo publicador/subscritor ou cliente/servidor com interrogações esporádicas.

iii) O contexto do serviço. Por exemplo, o estado de um recurso representado pelo serviço pode desencadear a substituição do modelo de interacção, e.g. mudar de um modelo publicador/subscritor para *streaming* sempre que a temperatura detectada por uma rede sensores sem fios exceder um valor limite.

b) A adaptação pode resultar de um pedido do utilizador, ou ser despoletada automaticamente com base num conjunto de condições previamente definidas.

O suporte aos modelos de interacção dinâmicos no contexto de uma sessão, baseia-se no conceito de padrão tal como o que é proposto no contexto desta dissertação. Por um lado, padrões estruturais tais como *estrela*, *pipeline*, ou *façade* definem a visão estática da sessão em termos das interacções possíveis entre os clientes numa sessão e o serviço que é acedido. Por outro lado, padrões de comportamento como *produtor/consumidor*, *streaming*, *publicador/subscritor*, ou *cliente/servidor* definem a visão dinâmica da sessão – uma vez combinados com os padrões de estrutura, cada padrão de comportamento define o comportamento de cada elemento, bem como as suas dependências em termos de fluxos de dados e controlo, semelhante ao apresentado no nosso trabalho.

A Figura 5.25 mostra três exemplos de instâncias do conceito de Sessão, no domínio de uma aplicação de monitorização de inundações:

a) Sessão que oferece um *stream* de informação da velocidade e direcção do vento.

b) Sessão com modelo de interacção publicador/subscritor, sendo os clientes notificados

assim que o nível de precipitação aumenta para níveis perigosos (132 mm).

c) Sessão de agregação que permite combinar várias fontes de dados, e.g. características do vento e humidade, através de uma função de agregação parametrizável dinamicamente. A agregação é implementada com um *pipeline* definindo uma etapa de agregação seguida de uma etapa de disseminação, sendo cada etapa representada estruturalmente com um *Façade*. Por simplificação, o modelo de comportamento usado em ambas as etapas é o mesmo (e.g. *streaming*).

Tal como definido nesta dissertação, a divisão entre padrões de estrutura e comportamento, permite então

a) Combinar padrões de um modo flexível e reconfigurar, em tempo de execução, quer a estrutura quer o comportamento, com regras pré-definidas que obedecem à semântica de cada padrão.

b) A evolução/adaptação do sistema pode ser representada por uma sequência pré-definida de padrões, capturada por uma máquina de estados. Em [Bap10] é apresentada uma descrição detalhada desta máquina de estados.

5.3.2 Arquitectura do Sistema Integrado

Partindo do conceito de Sessão acima descrito, pretende-se usar o *Activiti* estendido com padrões tal que tarefas num *workflow* possam ser clientes de uma sessão particular. Assim, estas tarefas poderão parametrizar a recolha de dados no contexto da sessão, por exemplo, qual o modelo de interacção utilizado para essa recolha, bem como adicionar novas fontes de dados, caso seja necessário. Adicionalmente, essas tarefas poderão parametrizar ou desencadear a reconfiguração dessa sessão sendo, em consequência, notificadas das reconfigurações dinâmicas que ocorram nesse contexto.

Para mais, tornar-se-á possível definir um *workflow* de tarefas com um modelo de fluxo de dados e de controlo correspondente ao que está activo no contexto de uma sessão. Por exemplo, uma tarefa que seja cliente de uma sessão com um modelo de interacção *streaming* pode propagar esses dados para outras tarefas do *workflow*, no contexto de um padrão de estrutura, utilizando também o modelo de *streaming*. Caso o modelo de interacção, no contexto dessa sessão, seja alterado dinamicamente para um modelo publicador/subscritor, a captura dessa notificação permite desencadear a reconfiguração do fluxo de dados desse *sub-workflow* também para publicador/subscritor.

A abordagem de "reconfiguração por padrão" de um *workflow*, tal como proposto nesta tese, permitirá assim que fontes de dados externas desencadeiem, automaticamente, a adaptação dinâmica do modelo de fluxo de dados em partes de um *workflow*. Do mesmo modo, reconfigurações dinâmicas no contexto de um *workflow* poderão desencadear reconfiguração dinâmicas do lado de uma sessão particular. A disponibilização destas características de dinamismo pretende ser uma contribuição para os requisitos de

adaptação que são necessários tanto para os *workflows* da área de negócio, como da área científica, tal como discutido em [CRPN08].

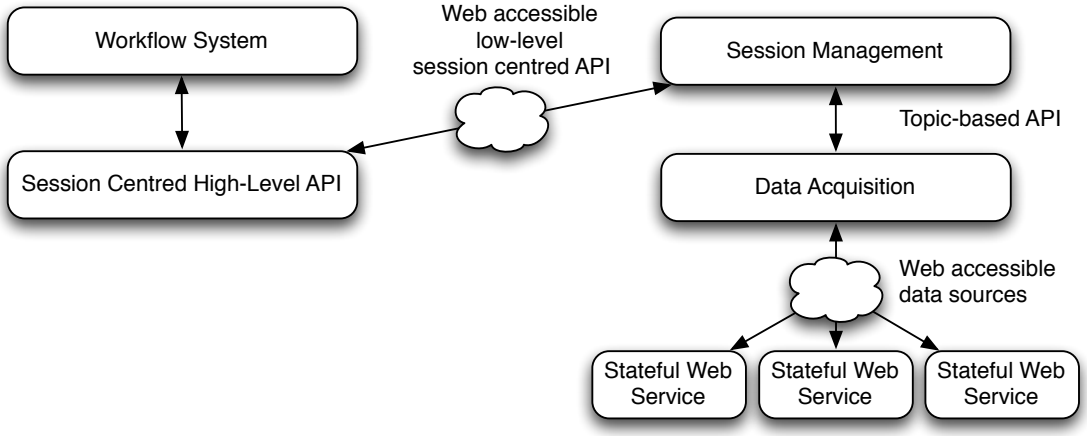


Figura 5.26: Arquitectura geral

A Figura 5.26 ilustra a arquitectura do sistema integrado pretendido, a qual corresponde a um modelo *multi-tier* separando, em camadas, os múltiplos interesses do sistema, i.e. apresentação, lógica e acesso a dados:

Data Acquisition Layer camada que interage com fontes de dados acessíveis através de serviços *web* com estado, disponibilizando uma *API* baseada em tópicos para essa interacção. As camadas superiores podem associar tópicos a essas fontes de dados, bem como a restrições sobre essas fontes de dados (e.g. dados da fonte que obedecem uma dada condição, como seja "temperatura > 40 graus centígrados").

Session Managment Layer camada que implementa a abstracção da sessão, oferecendo as ferramentas para criar, terminar, e gerir sessões (e.g. entrada e saída de clientes, tempo de vida da sessão, alterações dinâmicas ao modelo de interacção em uso, etc.). Como uma sessão pode incluir membros geograficamente dispersos, o serviço *web* é apresentado através de uma interface simples a ser usaao em *APIs* de linguagens de alto-nível.

Session-Centred High-Level API camada que disponibiliza uma interface de alto-nível para o conceito de sessão, de acordo com as características enumeradas acima.

Workflow System camada que representa o trabalho proposto nesta tese e, a partir da qual, é invocada a API de alto nível para o conceito de Sessão (e.g. definição de tarefas do *workflow* como clientes de uma sessão).

A próxima sub-secção complementa o exemplo dado na secção 5.2, no domínio dos sistemas aplicacionais orientados a dados dinâmicos (*Dynamic data driven applications systems (DDDAS)*), tirando agora partido deste sistema integrado que se pretende vir a disponibilizar.

5.3.3 Exemplo de Aplicação

Tal como descrito em [Dar04], o conceito *Dynamic Data Driven Applications and Systems (DDDAS) "implica a capacidade de incorporar dados dinamicamente numa aplicação de simulação em execução, sejam esses dados recolhidos em tempo (quase) real ou armazenados em repositórios, bem como, inversamente, a capacidade dessas aplicações controlarem dinamicamente os mecanismos de recolha desses dados"*. O resultado pretendido é melhorar quer a precisão das simulações, quer o seu tempo de execução. Tal pode ser conseguido a) com a selecção dinâmica de diferentes tipos dados; b) incorporando, em tempo de execução, modelos e módulos computacionais específicos do processamento desses dados; c) restringindo a área a ser observada, de acordo com o evoluir da situação; d) com a reutilização de dados já processados existentes em arquivo, e que correspondam a situações similares.

Dada a complexidade deste tipo de aplicações, os seus requisitos são variados, abrangendo diferentes domínios e necessidades computacionais em larga escala. Nesta secção é apresentado um exemplo particular de uma simulação neste contexto e que estende o que foi descrito na secção 5.2. De modo a ilustrar como o sistema integrado descrito poderá vir a contribuir para aplicações neste contexto, descrevem-se de seguida alguns dos seus requisitos (e que servem de base ao exemplo da secção 5.3.3.1):

a) **Fontes de dados diversificadas** O acesso e parametrização de informação recolhida por diversas fontes de dados, recolhida quer em tempo real ou quase real (*online data*) quer armazenada em repositórios ao longo do tempo, permite melhorar a precisão das simulações bem como melhorar o seu desempenho.

   Com o sistema integrado proposto, as tarefas podem ser clientes de uma sessão pelo que, podem parametrizar a recolha dos dados no contexto dessa sessão i) adicionando novas fontes de dados, ii) parametrizando a função de agregação, e iii) alterando dinamicamente o modelo de interacção, por exemplo passando de um modelo de notificação esporádica (publicador/subscritor) para um modelo de aquisição contínua de dados (*streaming*). Por outro lado, é possível adicionar novas tarefas a um *workflow* representando o acesso a repositórios para guardar os dados recolhidos e/ou já processados.

b) **Incorporação dinâmica de dados na simulação** Uma abordagem comum em sistemas DDDAS para incluir dados dinamicamente e de diferentes tipos, é organizar as simulações em módulos *"plug-and-play"*. Estes podem ser executados "a pedido" e escolhidos de acordo com o tipo dos dados a processar.

   No caso do sistema integrado, os padrões de estrutura podem ser utilizados para representar algumas formas de estruturação de uma simulação em módulos. Tal permite adicionar dinamicamente novos módulos a esses padrões, bem como parametrizar o modelo de interacção utilizado.

c) **Capacidade de processamento adicional em tempo de execução** No caso da adição dinâmica de novos módulos à simulação, pode ser necessário requerer capacidade de

processamento adicional para a infra-estrutura, de modo a executar toda a simulação em tempo útil.

Como o sistema integrado permite aceder a serviços computacionais (e.g. serviços *Grid* ou *cloud*), é possível requerer desses serviços mais recursos para melhorar o desempenho total. Do mesmo modo, pode ser feita a inclusão de serviços que ofereçam aplicações de suporte a uma área científica particular, adequadas ao tipo de dados que foram incluídos dinamicamente.

d) **Representar actividades realizadas por humanos** Dada a complexidade destes sistemas, torna-se necessário ter peritos de diferentes domínios como parte do sistema. Adicionalmente, nem sempre é possível automatizar todas as alterações dinâmicas a realizar, de modo a obter resultados mais precisos e em tempo útil. Assim, cabe a esses peritos, por exemplo, seleccionar que fontes de dados podem ser necessárias num determinado instante; que parte da zona crítica deve ser observada num determinado momento; que módulos científicos é necessário incluir dinamicamente na aplicação; identificar capacidades adicionais de processamento necessárias; controlar quem tem acesso aos vários tipos de informação; gerir agentes no terreno, no caso de monitorização de cenários de emergência, etc.

No caso do sistema integrado, por um lado é possível incluir tarefas representando actividades de humanos (tarefas oferecidas pelo *Activiti/BPMN*). Por outro, as reconfigurações no *workflow* podem ser desencadeadas a pedido por humanos, e.g. uma autoridade local responsável pela gestão de uma situação de emergência. Adicionalmente, alguns dos clientes de uma sessão podem representar agentes actuantes no terreno (e.g. bombeiros utilizando dispositivos móveis) e que, perante o agravamento de uma situação, podem desencadear reconfigurações dinâmicas no contexto da sessão. Como resultado, poderão ser também desencadeadas reconfigurações dinâmicas no contexto de um *workflow*.

Segue-se a descrição do exemplo propriamente dito, ilustrando como o sistema integrado a desenvolver pode contribuir para resolver alguns dos requisitos enunciados. A que referir que as figuras apresentadas no exemplo foram extraídas de um artigo já submetido e na referência [MCG12], pelo que os padrões de estrutura combinados com os padrões de comportamento (i.e. Produtor/Consumidor, *Streaming*, etc) não têm a mesma representação gráfica utilizada nas figuras anteriores (cor vermelha). Assim não existe diferenciação entre dependências do tipo fluxo de dados, das dependências do tipo fluxo de controlo.

5.3.3.1 Evolução de um Cenário Crítico de Inundação

Como é característico em zonas críticas que apresentem cenários recorrentes de inundações (e.g. como resultado de eventos súbitos de elevadas quantidades de precipitação), a área deve encontrar-se em contínua monitorização. Para tal, pode ser criada uma sessão

no contexto da qual é possível aceder, via *web*, a dados recolhidos por diferentes WSNs (e.g. valores de precipitação, humidade, direcção e velocidade do vento, níveis e velocidade da água na zona crítica, etc.)

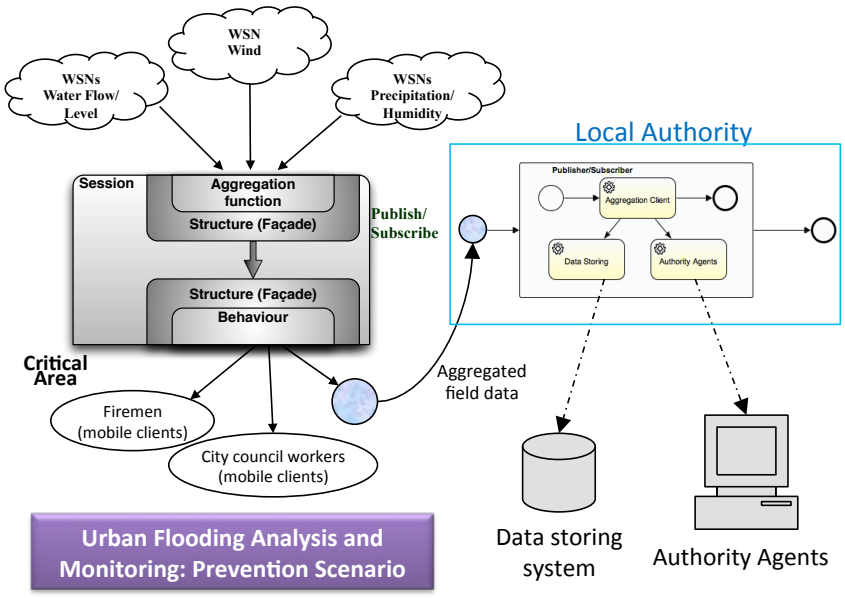
Inicialmente, numa situação de prevenção (e.g. em que as condições meteorológicas não indiquem possibilidade de precipitação), os dados recolhidos no contexto da sessão podem ser simplesmente valores de humidade na zona. Estes valores podem ser comunicados aos clientes da sessão apenas esporadicamente, usando um modelo publicador/-subscritor. Exemplos de clientes desta sessão são uma aplicação gerida pela autoridade local, e que é responsável por gerir situações de emergência (e.g. protecção civil), e ainda bombeiros responsáveis pela área crítica.

Para mais, a sessão pode estar já parametrizada para que, caso os níveis de humidade ultrapassem um valor de segurança, seja desencadeada automaticamente uma reconfiguração para uma sessão de agregação, adicionando por exemplo a recolha de dados de precipitação, tal como descrito em [Bap10]. Embora mantendo o modelo publicador/-subscritor, os clientes da sessão podem agora ter uma visão mais precisa das condições da área em observação.

Novos clientes podem entretanto juntar-se à sessão, e.g. trabalhadores da câmara municipal, passando deste modo a receber, através dos seus dispositivos móveis, a mesma informação que os restantes clientes, bem como quaisquer notificações que ocorram nesse contexto. O conceito de sessão permite assim partilhar a informação do contexto a novas entidades, de um modo simples. Caso ocorra um agravamento da situação, e de modo a obter informação mais precisa das condições meteorológicas locais, o dono da sessão pode adicionar dinamicamente novas fontes de dados, e.g. velocidade e direcção do vento, e nível e velocidade das águas. Esta reconfiguração dinâmica a pedido é notificada a todos o membros envolvidos, sendo os dados agregados disseminados por todos. Assume-se que estão pré-definidas as respostas, por parte dos vários clientes, aos vários eventos dinâmicos. Por exemplo, as equipas no terreno (e.g. bombeiros e trabalhadores da câmara) estão subordinadas à autoridade local, e seguem um protocolo de acção de emergência pré-definido.

Adicionalmente, os dados agregados podem ser gravados para posterior processamento, bem como ser observados por agentes autorizados. Esta situação de prevenção é ilustrada na Figura 5.27, onde a combinação de uma estrutura em estrela com o padrão de comportamento publicador/subscritor através do sistema *Activiti* estendido, permite propagar os dados recebidos no contexto da sessão às tarefas que representam o armazenamento dos dados e sua visualização.

Posteriormente, se a precipitação e o nível da água atingirem valores elevados, o dono da sessão pode requer uma reconfiguração dinâmica na sessão de agregação, e.g. mudando o modelo de interacção de publicador/subscritor para *streaming*. Todos os elementos da sessão são notificados desta alteração, tendo acesso a um fluxo contínuo dos




```
INFO: JobAcquisitionThread starting to acquire jobs
Dec 1, 2011 8:59:41 PM org.activiti.engine.impl.bpmn.deployer.BpmnDeployer deploy
INFO: Processing resource diagrams/figure2.bpmn20.xml
Dec 1, 2011 8:59:41 PM org.activiti.engine.impl.bpmn.parser.BpmnParse parseDefinitionsAttributes
INFO: XMLSchema currently not supported as typeLanguage
Dec 1, 2011 8:59:41 PM org.activiti.engine.impl.bpmn.parser.BpmnParse parseDefinitionsAttributes
INFO: XPath currently not supported as expressionLanguage
thread:star1servicetask3
reconf: stream
Streaming
consumers: 2
```

Figura 5.27: Situação de prevenção.

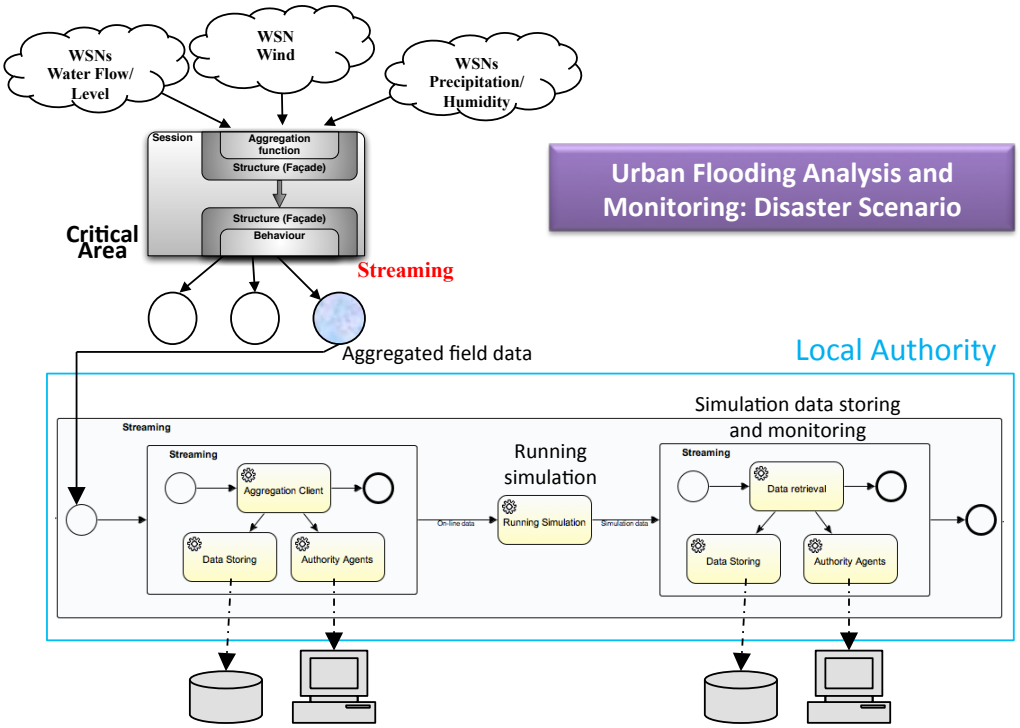


Figura 5.28: Situação de emergência.

dados agregados. Em consequência, o modelo de interacção no *workflow* pode ser alterado também para *streaming*, garantindo a propagação dos dados, com a mesma qualidade de serviço, para as tarefas de armazenamento e visualização dos dados (satélites do padrão estrela). Este pedido de reconfiguração dinâmica pode ser submetido através da linha de comandos do sistema *Activiti* estendido, sendo que estes comandos são processados no contexto do motor de execução do *workflow*.

Esta alteração é ilustrada na Figura 5.28 que apresenta também uma outra reconfiguração dinâmica do *workflow*, desencadeada a partir da linha de comandos do *Activiti*. Por um lado, uma aplicação de simulação e análise de inundações que processa o *stream* de dados recolhidos, é incluída como segunda etapa de um *pipeline*, sendo a primeira etapa a configuração em estrela descrita. Por outro lado, a terceira etapa do *pipeline* é um novo *sub-workflow* com topologia em estrela, onde os dados processados pela simulação são guardados para análise posterior e, podem ser observados por agentes da autoridade local. O modelo de interacção escolhido para as etapas do *pipeline*, bem como na topologia estrela, é novamente o modelo *streaming*, permitindo propagar o fluxo de dados para as várias tarefas.

Adicionalmente, a configuração hierárquica ilustrada na Figura 5.28 permite, caso seja necessário, reconfigurar individualmente cada padrão na hierarquia. Por exemplo, é simples adicionar novas tarefas à estrela na terceira etapa do *pipeline*, de modo a permitir que novos agentes da autoridade local acompanhem o evoluir da situação. Este tipo de reconfigurações dinâmicas segue o trabalho previamente apresentado em [GRC08,

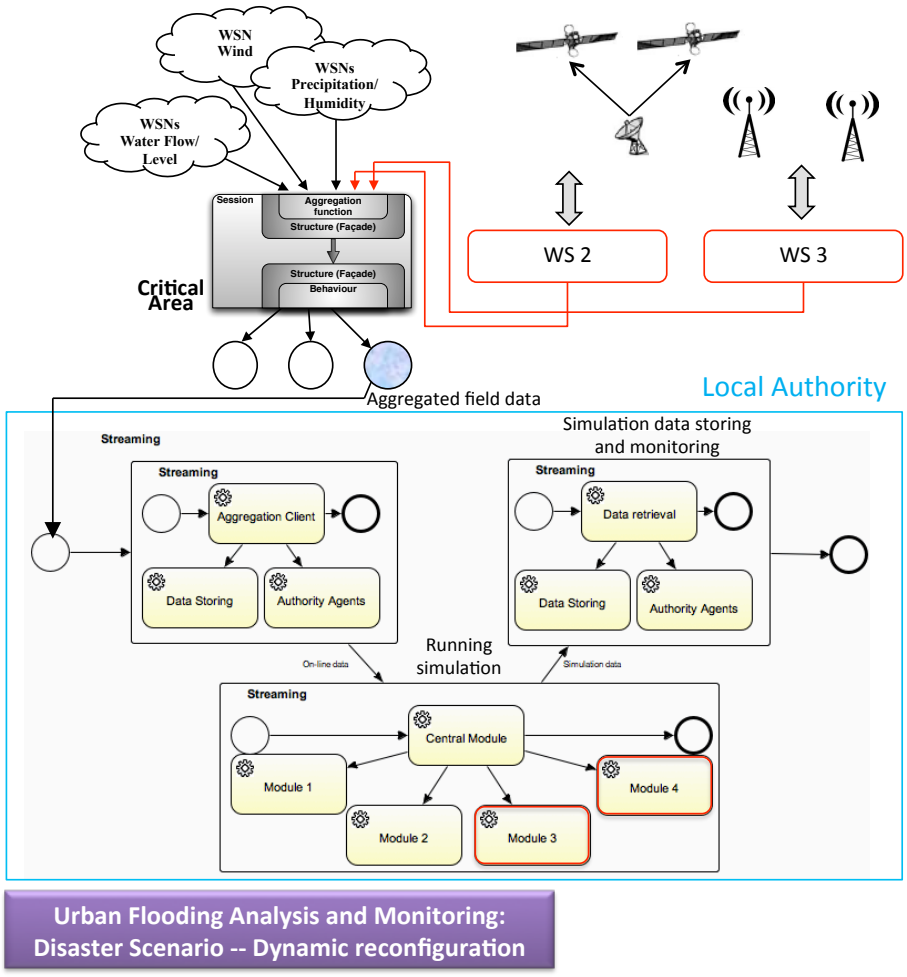


Figura 5.29: Aquisição dinâmica de dados de fontes de dados adicionais.

Gom07].

De modo similar, a Figura 5.29 mostra que é possível reconfigurar a segunda etapa do *pipeline* que representa a simulação em execução. Como descrito anteriormente, as aplicações no contexto de DDDAS têm de possibilitar a injecção de novos dados na simulação, o que pode ser conseguido se estas simulações estiverem estruturadas como um conjunto de módulos executáveis "a pedido".

Assim, a situação representada na segunda etapa do *pipeline* da Figura 5.29 corresponde a uma organização da simulação em módulos, tendo sido adicionados dinamicamente dois módulos para processar informação de duas novas fontes de dados. Como a Figura 5.29 mostra, foi desencadeada uma reconfiguração dinâmica da sessão de modo a incluir essas duas novas fontes de dados, e.g. para recolha de dados de humidade e vento em diferentes alturas do chão, bem como a recolha de dados de satélite das áreas sinistradas, representadas respectivamente por *WS 2* e *WS 3* na Figura 5.29.

6

Conclusões e Trabalho Futuro

A existência de aplicações distribuídas que interagem com múltiplos serviços, podendo esses serviços ser dependentes entre si, tem obrigado os sistemas computacionais a garantir algum dinamismo e automatismo na sua execução. O automatismo é garantido com o uso de *workflows*, os quais especificam um fluxo de trabalho automático que se rege pela execução de múltiplas tarefas. Neste contexto, surge o conceito de *workflow* de *web services* que oferece o automatismo necessário à interacção e interoperabilidade entre serviços, representando também as dependências entre eles.

No entanto, as ferramentas de suporte à especificação e execução de *workflows* são tradicionalmente separadas em dois domínios. Por um lado, existem ferramentas específicas da área de negócio que possibilitam a representação do fluxo de controlo das várias tarefas num processo. Por outro, existem ferramentas de *workflow* específicas das áreas de ciências e engenharia que representam a sua execução através de um fluxo de dados. Estas diferenças estabelecem requisitos próprios de cada domínio, dos quais são exemplos o foco no processamento de dados na área de ciências e engenharia ou a existência de tarefas humanas (*human tasks*), que possibilitam intervenção humana num processo de negócio. No entanto, com o evoluir dos sistemas estes requisitos tendem a ser necessários em ambos os domínios e não num domínio em particular.

Assim sendo, o objectivo geral desta tese consistiu na disponibilização de um protótipo que oferecesse mecanismos de composição de *workflows* de *web services* podendo as interacções entre tarefas ser baseadas em modelos de interacção mais ricos (e.g. Publicador/Subscritor). Esses mecanismos de composição e de interacção foram construídos sobre o conceito de padrão. Os modelos de interacção entre serviços são representados por padrões de comportamento, que, por sua vez, são parametrizados sobre padrões de estrutura (utilizados na composição dos serviços), sendo estes oferecidos sobre a forma

de *templates* instanciáveis (*"template-based patterns"*).

Outro objectivo focou o suporte de dinamismo tanto ao nível da composição das tarefas (inserir e remover tarefas na estrutura) como ao nível das interacções entre tarefas num *workflow* em tempo de execução. A necessidade da existência desse dinamismo nos *workflows* da área de ciência e engenharia é substanciada na secção 2.6, onde são apresentados vários cenários motivadores, nomeadamente no âmbito da gestão de recursos em ambiente distribuídos. A nossa abordagem às reconfigurações dinâmicas também se baseia no conceito de padrão, visto que, tal como para o desenvolvimento de sistemas, a utilização de padrões na adaptação dinâmica permite capturar e reutilizar situações recorrentes ou estratégias comuns. A sua implementação concreta aplica muitas das técnicas propostas em [CRPN08] (ver secção 2.6), como por exemplo, criação de tarefas abstractas, a possibilidade de alterar a estrutura (composição) tanto na fase de desenho como em tempo de execução.

O protótipo implementado atingiu os objectivos propostos através da extensão de uma ferramenta conhecida para a modelação de processos de negócio em *BPMN*, o *Activiti*.

6.1 Contribuição

As contribuições do trabalho realizado são:

1. Um *front-end* numa ferramenta de modelação de *workflows* na área de negócio (*BPMN 2.0*) com a possibilidade de utilizar padrões característicos da área de ciência e engenharia (i.e. padrões de fluxo de dados);
2. O uso de padrões como um meio para representar interacções entre vários *web services*;
3. Definição de uma arquitectura e implementação de um protótipo que incorpora:
   - Implementação de padrões identificados em 1.
   - Suporte à construção e composição de *Workflow* de *Web Services* com estruturação baseada em padrões
   - Mecanismos de reconfiguração dinâmica baseada nos padrões seleccionados.

Em relação as contribuições para a comunidade científica, parte deste trabalho foi incorporado no artigo [MCG12]. Neste momento estão em submissão dois artigos que focam trabalho realizado no âmbito desta tese: [FAP] e [MCG].

6.2 Análise Crítica

A disponibilização de padrões numa ferramenta de orquestração da área de negócio permite exemplos como o da gestão documental (secção 5.1). Esse exemplo ilustra um cenário em que o fluxo de execução tem de ser sequencial, isto é, o processo de "gestão de

projectos" tem de ser realizado por etapas. No entanto, para certas etapas desse processo, como por exemplo o caso dos sub-processos de *marcação de reunião* e *análise requisitos em falta*, é útil que a sua execução seja despoletada a partir de um fluxo de dados (execução dos produtores em simultâneo com a dos consumidores). Porém nesses sub-processos verifica-se que as tarefas representam actividades manuais/interacção com pessoas, e que o exemplo é executado no contexto de uma empresa, e não num ambiente "verdadeiramente distribuído", é por isso interessante ter esse tipo de padrões implementados numa perspectiva de orquestração e não de coreografia.

Apenas no caso de termos pessoas com telemóveis, em que actividade requer mesmo que eles se encontrem dispersos geograficamente, o que é o caso do exemplo DDDAS (corporações de bombeiros, por exemplo), ou em que se pretende coordenar os módulos da simulação, ou as outras tarefas em ambiente distribuído, de facto é mesmo relevante que se tratem de coreografias, e que o fluxo de dados nesses padrões não transitem pela ferramenta que faz a orquestração de todo o *workflow*.

De referir que algumas das funcionalidades que nos propusemos a realizar não foram implementadas: i) remover tarefas e alterar valores parametrizado (e.g. subscrição) em tempo real; ii) aplicar reconfigurações dinâmicas automáticas, pré-definidas pelo utilizador. A razão resume-se ao tempo despendido em encontrar uma ferramenta que fosse de encontro aos nossos requisitos (invocação de *web services*, *UI*, motor de execução), o que implicou que o processo de levantamento tecnológico fosse moroso. Para tal contribuiu também o facto de se preferir uma ferramenta na área de negócio, em vez de, por exemplo, na área de computação científica. Como estas ferramentas não disponibilizam um mecanismo de base para fluxo de dados entre tarefas, a implementação de padrões como Produtor/Consumidor, *Streaming*, etc, torna-se mais complexa. De facto, como o fluxo de execução está associado a dependências de fluxo de controlo e não de fluxo de dados, e como esse controlo está muito acoplado ao funcionamento do *Activiti Engine*, não foi simples alterar o seu funcionamento de modo a que os dados em trânsito desencadeiem a execução dinâmica da(s) tarefa(s) que os deve(m) processar. No entanto, problemas similares existem em ferramentas com fluxo de execução baseados em fluxo de dados (e.g. Triana [Tri]). Nestas ferramentas o fluxo de controlo está acoplado ao fluxo de dados, tornando-se também complicado disponibilizar padrões de coordenação baseados apenas em dependências de controlo que sejam independentes do fluxo de dados.

Grande parte do tempo disponível para a fase de elaboração desta dissertação foi despendido no desenvolvimento e implementação de soluções para o suporte de fluxo de dados na ferramenta *Activiti*, em particular no *Activiti Engine*. Este tempo deveu-se às várias limitações encontradas na ferramenta. Como descrito na secção 4.5, o facto do *Activiti Engine* ainda estar em desenvolvimento teve impacto ao nível das funcionalidades oferecidas, nomeadamente no que se refere à execução assíncrona de tarefas e de sub-processos, um requisito fundamental para a implementação de padrões de fluxo de dados.

Infelizmente, dado o tempo limitado, também não foi possível especificar os modelos

de interacção tratados nesta tese com recurso à notação BPMN 2.0, em particular *pools and lanes*. Tal como descrito no capítulo 3, a notação utilizada neste trabalho não segue assim a notação BPMN para esses modelos, mas pretendeu representar os modelos de fluxo de dados de um modo que é comum às ferramentas da área científica, e que ao nível mais abstracto da especificação de *workflow*, são semelhantes ao modos como são especificadas as dependências de fluxo de controlo no contexto da área de negócio.

No entanto, é nossa intenção definir uma notação que seja mais clara e permita distinguir o modelo fluxo de dados do modelo fluxo de controlo. Tal pode passar pelo uso das *pools and lanes*, ficando em aberto como podem ser especificadas reconfigurações dinâmicas de várias *lanes* dentro de uma *pool*.

6.3 Trabalho futuro

Pretende-se, como trabalho futuro, implementar os padrões de comportamento apresentados nesta tese como coreografias, à semelhança do que é feito no artigo [FGM11], com o conceito de "*workflow skeletons*" representado como um conjunto de "*proxies*" e cujos dados em trânsito não passam pela ferramenta de orquestração que gere a totalidade do *workflow* (no nosso caso, a ferramenta *Activiti*). A abordagem apresentada nesse artigo, embora interessante, não apresenta mecanismos de reconfiguração dinâmica como os apresentados neste trabalho. Nada é dito sobre a reconfiguração dinâmica de cada "*workflow skeleton*", algo que já foi abordado no trabalho [Gom07] no contexto da ferramenta *Triana*, nem sobre a possibilidade de substituir esses "*workflow skeletons*" no contexto do "*workflow skeleton*" de orquestração (por exemplo, substituir um padrão por outro), tal como foi descrito neste trabalho.

Um abordagem similar é também discutida em [SKD10], no sentido que, a partir de um *workflow* da área de negócio, existem tarefas que acedem a/representam *workflows* da área científica. Este trabalho, se por um lado permite a integração de diferentes *workflows* da área de ciência, orquestrados por um *workflow* da área científica, nada refere em termos de padrões de comportamento e mecanismos de reconfiguração dinâmica associados, tal como foi descrito neste trabalho.

Outra das acções a realizar como trabalho futuro, é precisamente tirar partido das novas funcionalidades oferecidas pelas versões mais recentes da ferramenta *Activiti*. Tal permitirá simplificar a implementação dos padrões e disponibilizar novos padrões de comportamento (e.g. *Master/Slave,* All-Pairs, Map-reduce, *etc*).

Adicionalmente, pretende-se adicionar um maior leque de padrões estruturais e de comportamento, tais como, *Façade*, *Adapter*, bem como completar todas as configurações possíveis para os padrões de estrutura e de comportamentos referidos nesta tese (e.g. alterar os papéis de produtor e consumidor no contexto de uma estrela, tal que seja também possível que o núcleo passe a ter um comportamento de consumidor e os satélites sejam os produtores).

Outra funcionalidade pretendida é possibilitar a geração de padrões de estrutura e

comportamento automaticamente através de *scripts*, bem como implementar reconfigurações dinâmicas automáticas, por exemplo, através de um conjunto de regras e/ou de uma máquina de estados tal como referido no artigo [MCG12].

Finalmente, será finalizada a integração da ferramenta de *workflows* com outros sistemas de *middleware* que disponibilizem o contexto de sessão (ou no âmbito de um máquina local, ou no contexto de *Cloud Computing*). Tal permitirá usar esta ferramenta de *workflow* como *front-end* para esses sistemas, definindo um ambiente integrado que, na nossa opinião, constitui uma contribuição interessante nos domínios de negócio e de ciência e engenharia.

Bibliografia

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, e Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, Berlin, 2004.

[Act] Activiti. Activiti BPMN platform, http://www.activiti.org/. http://www.activiti.org/.

[AIM10] Luigi Atzori, Antonio Iera, e Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, Outubro 2010.

[Arc] Service Architecture. Online articles. http://www.service-architecture.com/articles/index.html, keywords = SOA Tutorial,.

[Bap10] Adérito Baptista. Dynamic adaptation of interaction models for stateful web services, 2010.

[BDtH05] Alistair P. Barros, Marlon Dumas, e Arthur H. M. ter Hofstede. Service interaction patterns. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, e Francisco Curbera, editores, *Business Process Management*, volume 3649, pág. 302–318, 2005.

[BGP12] Adérito Baptista, M. Cecília Gomes, e Hervé Paulino. Session-based dynamic interaction models for stateful web services. In Mehdi Snene, editor, *Exploring Services Science - Third International Conference, IESS 2012, Geneva, Switzerland, February 15-17, 2012*, number 104 in Lecture Notes in Business Information Processing, pág. 29–43. Springer-Verlag, 02 2012.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, e Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[CPM06] William R. Cook, Sourabh Patwardhan, e Jayadev Misra. Workflow patterns in orc. In Paolo Ciancarini e Herbert Wiklicky, editores, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pág. 82–96. Springer, 2006.

[CRPN08] Manuel Caeiro-Rodriguez, Thierry Priol, e Zsolt Németh. Dynamicity in scientific workflows. Relatório Técnico TR-0162, Institute on Grid Information, Resource and Workflow Monitoring Services , CoreGRID - Network of Excellence, August 2008.

[dAea] Van der Aalst et al. Workflow patterns home page. `http://www.workflowpatterns.com/`, keywords = Workflow Patterns,.

[Dar04] Frederica Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In Marian Bubak, G. Dick van Albada, Peter M. A. Sloot, e Jack Dongarra, editores, *International Conference on Computational Science*, volume 3038 of *Lecture Notes in Computer Science*, pág. 662–669. Springer, 2004.

[Dom12] João Domingos. Dynamic interaction models for cloud computing service ('em fase de elaboração'), 2012.

[FAP] Cecília Gomes Filipe Araújo e Hervé Paulino. Reconfiguração dinâmica estruturada de workflows de serviços web. Em submissão ao INforum2012.

[FGM11] Tino Fleuren, Joachim Götze, e Paul Müller. Workflow skeletons: Increasing scalability of scientific workflows by combining orchestration and choreography. In Gianluigi Zavattaro, Ulf Schreier, e Cesare Pautasso, editores, *ECOWS*, pág. 99–106. IEEE, 2011.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Tese de Doutoramento, University of California, Irvine, Irvine, California, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GHS95] Dimitrios Georgakopoulos, Mark F. Hornick, e Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[GL02] Dinesh Ganesarajah e Emil Lupu. Workflow-based composition of webservices: A business model or a programming paradigm? In *Proceedings*

of the 6th International Enterprise Distributed Object Computing Conference, pág. 273–284, Washington, DC, USA, 2002. IEEE Computer Society.

[GM09] Claudio Guidi e Fabrizio Montesi. Reasoning about a service-oriented programming paradigm. In Maurice H. ter Beek, editor, *YR-SOC*, volume 2 of *EPTCS*, pág. 67–81, 2009.

[Gom07] Maria Cecília Gomes. *Pattern Operators for Grid Environments.* Tese de Doutoramento, 2007.

[Gra] Eclipse Graphiti. Graphiti - a graphical tooling infrastructure, http://www.eclipse.org/graphiti/.

[GRC08] Cecilia Gomes, Omer F. Rana, e Jose Cunha. Extending grid-based workflow tools with patterns/operators. *Int. J. High Perform. Comput. Appl.*, 22:301–318, August 2008.

[Hoh07] Gregor Hohpe. Conversation patterns: Interactions between loosely coupled services. 2007.

[Hol95] D. Hollingsworth. Workflow management coalition - the workflow reference model. Janeiro 1995.

[HTZD08] Ta'Id Holmes, Huy Tran, Uwe Zdun, e Schahram Dustdar. Modeling human aspects of business processes — a view-based, model-driven approach. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pág. 246–261, Berlin, Heidelberg, 2008. Springer-Verlag.

[Ibm04] Don Ferguson Ibm. (Ver. 1.1):1–14, 2004.

[iDEP] Plug in Development Environment (PDE). Plug-in development environment (pde), http://www.eclipse.org/pde/.

[JBoa] JBoss. Jbpm, jboss tool for bpmn, http://www.jboss.org/jbpm.

[JBob] JBoss. Riftsaw, jboss tool for bpel orchestration, http://www.jboss.org/riftsaw.

[JWT] Eclipse JWT. Eclipse jwt, java workflow tooling, http://eclipse.org/jwt/.

[KC03] Jeffrey O. Kephart e David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, 2003.

[KL08] Oliver Kopp e Frank Leymann. Choreography design using ws-bpel. *IEEE Data Eng. Bull.*, 31(3):31–34, 2008.

[KLL09] Ryan K.L. Ko, Stephen S.G. Lee, e Eng Wah Lee. Business process management (bpm) standards: A survey. *Business Process Management journal*, 15(5), 2009.

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, e Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[LF07] Bo Liu e Yushun Fan. Research on architecture and key technology for service-oriented workflow performance analysis. In Kevin Chen-Chuan Chang, Wei Wang, Lei Chen, Clarence A. Ellis, Ching-Hsien Hsu, Ah Chung Tsoi, e Haixun Wang, editores, *APWeb/WAIM Workshops*, volume 4537 of *Lecture Notes in Computer Science*, pág. 540–545. Springer, 2007.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[MAVDA07] Nataliya Mulyar, Lachlan Aldred, e Wil M. P. Van Der Aalst. The conceptualization of a configurable multi-party multi-message request-reply conversation. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pág. 735–753, Berlin, Heidelberg, 2007. Springer-Verlag.

[MC90] Thomas W. Malone e Kevin Crowston. What is coordination theory and how can it help design cooperative work systems? In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, CSCW '90, pág. 357–370, New York, NY, USA, 1990. ACM.

[MCG] Adérito Baptista Filipe Araújo Maria Cecília Gomes, Hervé Paulino. Middleware support for structured dynamic adaptation mechanisms. Under submission.

[MCG12] Adérito Baptista Filipe Araújo Maria Cecília Gomes, Hervé Paulino. Dynamic interaction models for web enabled wireless sensor networks. In *'Aceite para conferência MUE 2012'*. IEEE, 2012.

[MG09] Peter Mell e Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[MGRtH11] Sara Migliorini, Mauro Gambini, Marcello La Rosa, e Arthur H.M. ter Hofstede. Pattern-based evaluation of scientific workflow management systems. February 2011.

[OAS] OASIS. Oasis web services resource framework (wsrf) tc. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf,.

[ODtHvdA07] Chun Ouyang, Marlon Dumas, Arthur H.M. ter Hofstede, e Wil M.P. van der Aalst. Pattern-based translation of bpmn process models to bpel web services. *International Journal of Web Services Research (JWSR)*, 5(1):42–62, 2007.

[OMG] OMG. Object managment group documents, bpmn 2.0. standards http://www.omg.org/spec/bpmn/2.0/.

[oTaA] The University of Texas at Austin. Orc language project, http://orc.csres.utexas.edu/. http://orc.csres.utexas.edu/.

[Ple02] Charles Plesums. Introduction to workflow. In *Workflow Handbook*, pág. 19–38, 2002.

[Pri] Priberam. Dicionário priberam de língua portuguesa. http://www.priberam.pt/dlpo/, keywords = Dicionário,.

[Pro] Eclipse Modeling Framework Project. Eclipse modeling framework project (emf), http://www.eclipse.org/modeling/emf/.

[PS11] Hervé Paulino e João Ruivo Santos. A middleware framework for the web integration of sensor networks. In Gerard Parr e Philip Morrow, editores, *Sensor Systems and Software - Second International ICST Conference, S-Cube 2010, Miami, FL, USA, December 13-15, 2010, Revised Selected Papers*, number 57 in Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engi, pág. 75–90. Springer-Verlag, 08 2011. URL=http://asc.di.fct.unl.pt/ herve/papers/SenSer-S-Cube-2010.pdf.

[RHE04] Nick Russell, Arthur H. M. Ter Hofstede, e David Edmond. Workflow resource patterns. 2004.

[RR07] Leonard Richardson e Sam Ruby. *Restful web services*. O'Reilly, first edition, 2007.

[Ser] OASIS Technical Committees Web Services. Oasis technical committees. http://www.oasis-open.org/committees/tc_cat.php?cat=ws, keywords = OASIS Web Services Articles,.

[Sim05] Richard C Simpson. An xml representation for crew procedures. 2005.

[SKD10] Mirko Sonntag, Dimka Karastoyanova, e Ewa Deelman. Bridging the gap between business and scientific workflows: Humans in the loop of scientific workflows. In *Proceedings of the 2010 IEEE Sixth International*

Conference on e-Science, ESCIENCE '10, pág. 206–213, Washington, DC, USA, 2010. IEEE Computer Society.

[SOA] OASIS Technical Committees SOA. Oasis technical committees. http://www.oasis-open.org/committees/tc_cat.php?cat=soa, keywords = SOA articles,.

[Tri] Triana. Triana, http://www.trianacode.org/. http://www.trianacode.org/.

[vdAH03] W.M.P. van der Aalst e A. H. M. Ter Hofstede. Yawl: Yet another workflow language. *Information Systems*, 30:245–275, 2003.

[VDATHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, e A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14:5–51, July 2003.

[vdAvH02] Wil v. d. van der Aalst e Kees v. van Hee. Workflow Management: Models, Methods, and Systems (Cooperative Information Systems). Janeiro 2002.

[W3S] W3Schools. Web services tutorial. http://www.w3schools.com/webservices/default.asp, year = 2008, keywords = Web Services Tutorial,.

[WC99] Stephen A White e I B M Corp. Process modeling notations and workflow patterns. *Business*, 21(1999):1–25, 1999.

[WSC04] Web services choreography description language version 1.0, 2004.

[WvdAD+05] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede, e Nick Russell. Pattern-based analysis of bpmn - an extensive evaluation of the control-flow, the data and the resource perspectives. 2005.

[YAW] W.M.P van der Aalst YAWL. Yawl: Yet another workflow language http://www.yawlfoundation.org/.

[YB05] Jia Yu e Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34:44–49, September 2005.

[YGN09] Ustun Yildiz, Adnene Guabtni, e Anne H. H. Ngu. Business versus scientific workflows: A comparative study. In *SERVICES I*, pág. 340–343. IEEE Computer Society, 2009.

Anexos

Listing A.1: Exemplo geração *BPMN 2.0*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn" xmlns
  <process id="TestNewPubSub" name="TestNewPubSub">
    <documentation>Place documentation for the 'TestNewPubSub' process here.</documentation>
    <startEvent id="startevent1" name="Start"></startEvent>
    <endEvent id="endevent1" name="End"></endEvent>
    <subProcess id="star1" name="Star">
      <startEvent id="star1startevent2" name="Start"></startEvent>
      <endEvent id="star1endevent2" name="End"></endEvent>
      <serviceTask id="star1servicetask1" name="A" activiti:class="ValuesTemperature">
        <extensionElements>
          <activiti:field name="topic_temperature_topic">
            <activiti:string>topic_temperature_topic</activiti:string>
          </activiti:field>
        </extensionElements>
      </serviceTask>
      <sequenceFlow id="star1flow8" name="" sourceRef="star1startevent2" targetRef="star1servicetask1"></sequenceFlow>
      <sequenceFlow id="star1flow12" name="" sourceRef="star1servicetask1" targetRef="star1endevent2"></sequenceFlow>
      <serviceTask id="star1servicetask2" name="B" activiti:class="JavaTask"></serviceTask>
      <sequenceFlow id="star1flow13" name="" sourceRef="star1servicetask1" targetRef="star1servicetask2">
        <extensionElements>
          <activiti:field name="behavior">
            <activiti:string>Publish/Subscriber</activiti:string>
          </activiti:field>
          <activiti:field name="behaviorCondition">
            <activiti:expression><![CDATA[${topic_temperature_topic == 20}]]></activiti:expression>
          </activiti:field>
        </extensionElements>
      </sequenceFlow>
      <serviceTask id="star1servicetask3" name="C" activiti:class="JavaTask"></serviceTask>
      <sequenceFlow id="star1flow14" name="" sourceRef="star1servicetask1" targetRef="star1servicetask3">
        <extensionElements>
          <activiti:field name="behavior">
            <activiti:string>Publish/Subscriber</activiti:string>
          </activiti:field>
          <activiti:field name="behaviorCondition">
            <activiti:expression><![CDATA[${topic_temperature_topic > 20}]]></activiti:expression>
          </activiti:field>
        </extensionElements>
      </sequenceFlow>
      <serviceTask id="star1servicetask4" name="D" activiti:class="JavaTask"></serviceTask>
      <sequenceFlow id="star1flow15" name="" sourceRef="star1servicetask1" targetRef="star1servicetask4">
        <extensionElements>
          <activiti:field name="behavior">
            <activiti:string>Publish/Subscriber</activiti:string>
          </activiti:field>
          <activiti:field name="behaviorCondition">
            <activiti:expression><![CDATA[${topic_temperature_topic > 40}]]></activiti:expression>
          </activiti:field>
        </extensionElements>
```

Figura A.1: Diagrama de classes principal

Figura A.2: Diagrama de classes

```
    </sequenceFlow>
  </subProcess>
  <sequenceFlow id="flow6" name="" sourceRef="startevent1" targetRef="substar1"></sequenceFlow>
  <sequenceFlow id="flow7" name="" sourceRef="star1" targetRef="endevent1"></sequenceFlow>
  <subProcess id="substar1" name="Workflow1">
    <startEvent id="substar1startevent3" name="startWk1"></startEvent>
    <receiveTask id="substar1receivetask1" name="receiveWk1"></receiveTask>
    <parallelGateway id="substar1parallelgateway1" name="parallelWk1"></parallelGateway>
    <serviceTask id="substar1publisher_behavior" name="Publisher" activiti:class="ValuesTemperature">
      <extensionElements>
        <activiti:field name="topic_temperature_topic">
          <activiti:string>topic_temperature_topic</activiti:string>
        </activiti:field>
      </extensionElements>
    </serviceTask>
    <exclusiveGateway id="substar1exclusivegateway1" name="exclusiveWk1"></exclusiveGateway>
    <endEvent id="substar1endevent3" name="endWk1"></endEvent>
    <endEvent id="substar1endevent4" name="endWk2"></endEvent>
    <serviceTask id="substar1dispatcher_behavior" name="dispatcher_behavior" activiti:class="ValuesTemperature">
      <extensionElements>
        <activiti:field name="runBehavior">
          <activiti:string>substar1publisher_behavior</activiti:string>
        </activiti:field>
      </extensionElements>
    </serviceTask>
    <sequenceFlow id="substar1flow16" name="" sourceRef="substar1startevent3" targetRef="substar1publisher_behavior"></sequenceFlow>
    <sequenceFlow id="substar1flow17" name="" sourceRef="substar1publisher_behavior" targetRef="substar1parallelgateway1"></sequenceFlow>
    <sequenceFlow id="substar1flow18" name="" sourceRef="substar1parallelgateway1" targetRef="substar1receivetask1"></sequenceFlow>
    <sequenceFlow id="substar1flow19" name="" sourceRef="substar1parallelgateway1" targetRef="substar1dispatcher_behavior"></sequenceFlow>
    <sequenceFlow id="substar1flow20" name="" sourceRef="substar1receivetask1" targetRef="substar1exclusivegateway1"></sequenceFlow>
    <sequenceFlow id="substar1flow21" name="" sourceRef="substar1exclusivegateway1" targetRef="substar1endevent4">
      <conditionExpression xsi:type="tFormalExpression"><![CDATA[${!pubSubRunsubstar1}]]></conditionExpression>
    </sequenceFlow>
    <sequenceFlow id="substar1flow22" name="" sourceRef="substar1exclusivegateway1" targetRef="substar1publisher_behavior"></sequenceFlow>
    <sequenceFlow id="substar1flow23" name="" sourceRef="substar1dispatcher_behavior" targetRef="substar1endevent3"></sequenceFlow>
  </subProcess>
  <sequenceFlow id="flow24" name="" sourceRef="substar1" targetRef="endevent1"></sequenceFlow>
</process>
<bpmndi:BPMNDiagram id="BPMNDiagram_TestNewPubSub">
  <bpmndi:BPMNPlane bpmnElement="TestNewPubSub" id="BPMNPlane_TestNewPubSub">
    <bpmndi:BPMNShape bpmnElement="startevent1" id="BPMNShape_startevent1">
      <omgdc:Bounds height="35" width="35" x="156" y="110"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="endevent1" id="BPMNShape_endevent1">
      <omgdc:Bounds height="35" width="35" x="565" y="100"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1" id="BPMNShape_star1">
      <omgdc:Bounds height="165" width="410" x="190" y="207"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1startevent2" id="BPMNShape_startevent2">
      <omgdc:Bounds height="35" width="35" x="210" y="237"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1endevent2" id="BPMNShape_endevent2">
      <omgdc:Bounds height="35" width="35" x="540" y="237"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1servicetask1" id="BPMNShape_servicetask1">
      <omgdc:Bounds height="55" width="105" x="342" y="219"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1servicetask2" id="BPMNShape_servicetask2">
      <omgdc:Bounds height="55" width="105" x="220" y="307"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1servicetask3" id="BPMNShape_servicetask3">
      <omgdc:Bounds height="55" width="105" x="345" y="307"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="star1servicetask4" id="BPMNShape_servicetask4">
      <omgdc:Bounds height="55" width="105" x="470" y="307"></omgdc:Bounds>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNEdge bpmnElement="star1flow8" id="BPMNEdge_flow8">
      <omgdi:waypoint x="245" y="254"></omgdi:waypoint>
      <omgdi:waypoint x="342" y="246"></omgdi:waypoint>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="star1flow12" id="BPMNEdge_flow12">
      <omgdi:waypoint x="447" y="246"></omgdi:waypoint>
      <omgdi:waypoint x="540" y="254"></omgdi:waypoint>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="star1flow13" id="BPMNEdge_flow13">
      <omgdi:waypoint x="447" y="246"></omgdi:waypoint>
      <omgdi:waypoint x="220" y="334"></omgdi:waypoint>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="star1flow14" id="BPMNEdge_flow14">
      <omgdi:waypoint x="447" y="246"></omgdi:waypoint>
      <omgdi:waypoint x="345" y="334"></omgdi:waypoint>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="star1flow15" id="BPMNEdge_flow15">
      <omgdi:waypoint x="447" y="246"></omgdi:waypoint>
      <omgdi:waypoint x="470" y="334"></omgdi:waypoint>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="flow6" id="BPMNEdge_flow6">
      <omgdi:waypoint x="191" y="127"></omgdi:waypoint>
      <omgdi:waypoint x="190" y="289"></omgdi:waypoint>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="flow7" id="BPMNEdge_flow7">
```

```
143         <omgdi:waypoint x="600" y="289"></omgdi:waypoint>
144         <omgdi:waypoint x="565" y="117"></omgdi:waypoint>
145       </bpmndi:BPMNEdge>
146     </bpmndi:BPMNPlane>
147   </bpmndi:BPMNDiagram>
148 </definitions>
```

Listing A.2: *Schema XML* reconfigurações

```
1  <?xml version="1.0"?>
2  <xs:schema id="ReconfValidate" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
3    <xs:element name="reconfigurations">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element name="behavior" minOccurs="0" maxOccurs="unbounded">
7            <xs:complexType>
8              <xs:sequence>
9                <xs:element name="condition" type="xs:string" minOccurs="0" msdata:Ordinal="0" />
10             </xs:sequence>
11             <xs:attribute name="id" type="xs:string" />
12           </xs:complexType>
13         </xs:element>
14         <xs:element name="addtask" minOccurs="0" maxOccurs="unbounded">
15           <xs:complexType>
16             <xs:sequence>
17               <xs:element name="condition" type="xs:string" minOccurs="0" msdata:Ordinal="0" />
18               <xs:element name="javaClass" type="xs:string" minOccurs="0" msdata:Ordinal="2" />
19               <xs:element name="fields" minOccurs="0" maxOccurs="unbounded">
20                 <xs:complexType>
21                   <xs:sequence>
22                     <xs:element name="field" minOccurs="0" maxOccurs="unbounded">
23                       <xs:complexType>
24                         <xs:sequence>
25                           <xs:element name="name" type="xs:string" minOccurs="0" />
26                           <xs:element name="value" type="xs:string" minOccurs="0" />
27                         </xs:sequence>
28                       </xs:complexType>
29                     </xs:element>
30                   </xs:sequence>
31                 </xs:complexType>
32               </xs:element>
33             </xs:sequence>
34             <xs:attribute name="id" type="xs:string" />
35           </xs:complexType>
36         </xs:element>
37       </xs:sequence>
38       <xs:attribute name="id" type="xs:string" />
39     </xs:complexType>
40   </xs:element>
41 </xs:schema>
```

Execution Entity

***Start*:** método responsável por iniciar a execução do *workflow*;

***End*:** método responsável por terminar o *workflow* em execução;

***Take*:** método que verifica qual o próximo passo a tomar, i.e., associa a transacção e tarefa que ficam activas;

***All (takeAll)*:** Método invocado quando existem múltiplas tarefas quem têm de a ser a executadas no próximo passo do *workflow*, invocando para cada tarefa o método *Take*.

***Perform Operation*:** método que verifica se a tarefa activa é assíncrona ou não e direcciona a sua execução para o método respectivo, i.e., caso seja assíncrono invoca *Send Job* caso contrário invoca *Perform Operation*.

Command Context

***Start*:** método responsável por instanciar a execução da *ThreadLocal* responsável por atender os pedidos;

***Perform Operation*:** método que recebe o identificador da próxima operação, proveniente da *Execution Entity* ou de uma outra *Atomic Operation*, e direcciona para a *Atomic Operation* respectiva.

***Send Job*:** Caso a tarefa seja assíncrona essa é enviada para a *pool* de threads gerida pelo serviço *ManagementService* (secção 3.2.2).

Behavior

***Execute*:** método que executa o comportamento associado a tarefa.

***Leave*:** verifica qual o próximo passo a realizar e invoca o método *Perform Operation*.

***Perform Operation*:** Termina o comportamento e envia qual (ou quais) a(s) próxima(s) tarefa(s) a ser(em) executada(s)

***End*:** Termina o comportamento.

***Signal*:** Todas as tarefas têm um identificador na base de dados, com este método é possível executar/retomar o comportamento com *signal(id)*.